

Improving Instrumentation Speed via Buffering

Dan Upton
Kim Hazelwood
University of Virginia

Robert Cohn
Greg Lueck
Intel Corporation

Abstract

Dynamic binary instrumentation systems are useful tools for a wide range of tasks including program analysis, security policy enforcement, and architectural simulation. The overhead of such systems is a primary concern, as some tasks introduce as much as several orders of magnitude slowdown. A large portion of this overhead stems from both data collection and analysis. In this paper, we present a method to reduce overhead by decoupling data collection from analysis. We accomplish this by buffering the data for analysis in bulk. We implement buffering as an extension to Pin, a popular dynamic instrumentation system. For collecting a memory trace, we see an average improvement of nearly 4x compared to the best-known implementation of buffering using the existing Pin API.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Debugging aids, Tracing*; D.3.4 [Programming Languages]: Processors—*Run-time environments*

General Terms

Performance, Experimentation

Keywords

Instrumentation, program analysis tools

1. Introduction

An important part of developing and maintaining efficient hardware and software is understanding the run-time behavior of the system. Collecting information about an executing application can be an involved process, requiring the user to not only determine relevant characteristics to study but also to determine how to collect the results and verify their correctness. Additionally, collecting information about an application for which the user does not have source code can be difficult.

Dynamic binary instrumentation systems provide a layer of abstraction for collecting information about a program's

run-time behavior. Furthermore, they require only the application binary for profiling, allowing profiling of an application without needing source code access or requiring the application to be recompiled. Such systems have been used for a variety of tasks ranging from collecting an instruction mix or memory trace to architectural simulation or enforcing security policies.

The main drawback for dynamic binary instrumentation systems (DBIs) is the overhead involved. Without adding any instrumentation code, there is still some amount of extra code introduced simply by running the DBI. The system must generally perform some sort of recompilation of the hosted binary; beyond that, it must perform a number of management tasks, such as managing the recompiled code, maintaining control, and handling signals and system calls. Adding instrumentation further increases the overhead, possibly ranging as high as 1000x or greater slowdown for sophisticated, compute-intensive tasks. Reducing the overhead of the DBI itself has been studied extensively with various dynamic binary systems [2, 4, 6, 8, 10, 11] and is not addressed in this paper.

Profiling an application can be broken down into two main phases – collecting the data and analyzing it. Both parts are responsible for overhead: collecting data requires adding and executing additional instructions, such as calculating the effective address of a memory access; analyzing the data generally involves running some user-defined function. Strategies for collecting and processing data can also be broken down into two general classes, online analysis and decoupled analysis.

Several methods have been suggested for reducing the impact of data analysis. For instance, PiPA [15] sought to overlap analysis with application execution and data collection by collecting small chunks of data and processing them in one or more separate threads. Other systems [9, 13] have parallelized data collection in a separate thread while the unmodified application runs in its own thread.

In this paper, we focus on reducing the overhead of the first phase, data collection. As noted above, analysis can either be performed online or in a decoupled manner, collecting chunks of data and processing a full chunk at once. We implement a buffering system for Pin [8] to effectively collect data for decoupled analysis. While such buffering can be accomplished using the preexisting interface to Pin, an efficient implementation is complex and a simple implementation is inefficient. Using our buffering system, users can write a comparable amount of code to the simple implementation while achieving greater performance than the more complex implementation.

There are three main contributions in this paper. First, we add a new API to Pin specific to tools that collect data in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WBIA '09, Dec 12, New York City, NY

Copyright ©2009 ACM 978-1-60558-793-6/12/09 ...\$10.00.

buffers. The API allows the user to control how the buffers are defined, how they are filled, and how their memory is managed. The buffers are thread-private, meaning the user does not need to use sophisticated methods to separate collected data. While it is possible to write a tool that collects data in a buffer with the existing API, the additions make it simpler to write the tool and simpler for Pin to optimize the code. Second, we optimize the code generated to write to the buffer using a combination of scalar optimizations and register allocation. Third, we reduce the cost of detecting when the buffer is full by using a probabilistic algorithm that is backed up by a safe but slower method. We achieve on average nearly a 4x improvement over the best-known method for implementing a buffer using the existing API.

The rest of the paper is organized as follows. Section 2 presents background on dynamic binary instrumentation systems and on Pin specifically, including a short overview of the Pin instrumentation API. Section 3 describes our implementation in detail, including the additions to the API, options for code generation, and how we handle buffer overflow. Section 4 provides a performance evaluation of our system, comparing execution time on both single- and multi-threaded programs. Section 5 discusses related work. Finally, Section 6 presents future work and concludes.

2. Background

Before discussing implementation details of buffering in Pin [8], we first present an overview of dynamic binary instrumentation and related systems. We then provide a brief introduction to Pin and its instrumentation interface. We will also describe options for buffering data for decoupled analysis which already exist in Pin.

Dynamic binary instrumentation systems (DBIs) such as Pin [8] and Valgrind [10] take control of a hosted application and have the ability to inspect every instruction in the application before executing it. This allows DBIs to add, modify, or remove instructions from the dynamic execution stream. Specifically, they are commonly used for inserting user-defined instrumentation instructions before points of interest, such as memory operations, branches, or function calls. Such instrumentation has many uses for profiling an application, and has also been used for such tasks as enforcing security policies [7].

DBIs are similar to dynamic binary translators and dynamic binary optimizers. Dynamic binary translators such as IA-32EL [3], Boa [1], or Transmeta’s CMS [6] take a stream of instructions for one instruction set architecture (ISA) and translate them to the ISA of the current processor. Dynamic binary optimizers such as Dynamo [2] and DynamoRIO [4] seek to improve performance of hosted applications by targeting “hot” code for optimization and utilizing information available at run time. Most of these systems share implementation details as they have many of the same basic issues to solve, such as maintaining control (*e.g.*, capturing branches to make sure the application does not return to its original code). Additionally, many of these systems maintain a “software code cache” from which code is executed after translating, optimizing, or instrumenting once. This avoids the overhead of repeating work.

In this paper, we describe an implementation of buffering in Pin. Pin is a DBI developed at Intel that consists of a virtual machine (VM), a software code cache as described above, and an instrumentation interface. The instrumenta-

tion interface provides a large API for writing user-defined programs, called PinTools, which describe where to add instrumentation and how to process the collected data. Pin’s API allows most tools to be written once and then compiled and run on any supported OS (Linux, Windows, MacOS, or FreeBSD) and architecture (IA-32, Intel 64, IA-64, or ARM). When instrumentation code is added, a just-in-time compiler (JIT) in the VM manages isolating program state from VM and tool state, including saving and restoring registers as necessary. The JIT also performs register allocation and inlines analysis code when possible.

Pin’s instrumentation API provides several ways to add instrumentation code, such as before an instruction or before a basic block, using functions such as `INS_InsertCall` or `BBL_InsertCall` respectively. More sophisticated instrumentation is also possible, such as “if/then” instrumentation, where a lightweight boolean call is added first and more involved instrumentation is only executed if the boolean function evaluates to `true`. The user may also specify where to insert instrumentation using `IPOINT` directives, such as `IPOINT_AFTER`, used to place analysis code after the instruction under analysis.

There are two basic methods for analyzing the data collected by adding instrumentation code, online or decoupled. Online analysis can be very flexible. User-defined arguments, such as branch target or memory operation addresses, are passed to a user-defined routine that analyzes the data. However, the flexibility comes at a cost. The user-defined routine may overwrite registers, and these registers may be live in the application when the routine is called. Thus, Pin inserts register saves and restores to prevent the registers from being overwritten. To reduce the number of saves and restores, Pin does extensive optimizations. Pin may also inline analysis routines; however, this code duplication can negatively impact cache performance, and if the code is infrequently used, the additional time spent in the JIT may outweigh any benefits. Decoupled analysis – collecting a large amount of data to be processed at once – can lessen or avoid both of these sources of overhead, because a specialized buffer fill routine can have a smaller register and instruction footprint than generalized analysis routines.

Buffering data for decoupled analysis is possible in Pin using the existing API. However, as will be described in more detail later in Section 4, a straightforward implementation of buffering is inefficient. An efficient implementation is possible but is more involved and difficult to get right – for instance, the user must be aware of how to use the “if/then” instrumentation, ensure the buffer fill code is inlined, and avoid modifying the `eflags` register on IA-32 and Intel 64 architectures. The buffering implementation described in this paper manages these details transparently to the user, allowing them to write a tool in a straightforward manner while achieving the performance of the more involved implementation.

3. Implementation

In this section, we will discuss the various facets of implementing buffering of data for decoupled analysis in Pin. First, we describe the user interface, implemented as additions to the Pin instrumentation API. Next, we describe the code generation for filling the buffers, including options for store instructions. Finally, we discuss handling the buffer overflow condition and allowing the user to process the data.

```

#define NUMPAGES 1000
struct MEMREF{
  ADDRINT addr;
}

VOID* BufferFull(BUFFER_ID id, THREADID tid,
  const CONTEXT * ctxt, VOID *buf,
  unsigned numElt, VOID *v){
  // process buffer as necessary
  return buf;
}

VOID Ins(INS ins, VOID* v){
  if(INS_IsMemoryRead(ins)){
    INS_InsertFillBuffer(ins, IPOINT_BEFORE,
      IARG_MEMORYREAD_EA,
      offsetof(struct MEMREF, addr),
      IARG_END);
  }
}

int main(){
  PIN_Init(argc, argv);
  BUFFER_ID id=PIN_DefineBuffer(sizeof(MEMREF),
    NUMPAGES, BufferFull, 0);
  INS_AddInstrumentFunction(Ins, 0);
  PIN_StartProgram();
  return 0;
}

```

Figure 1: Sample PinTool using the buffering API described in this paper.

3.1 Buffering User Interface

The buffering API can largely be broken into three classes: describing the buffer, filling the buffer, and managing the buffer memory. Figure 1 shows a sample PinTool demonstrating the use of this API to collect a trace of memory load addresses. Managing buffer memory is not shown in the sample tool.

Describing the buffer: We first provide a new call, `PIN_DefineTraceBuffer`, which takes as parameters the size of an individual record, the total size of the buffer, a callback function, and an optional pointer (such as to a struct or object). In the general use case, an individual record will be a struct or an object, although it is possible to store a single value without wrapping it in a struct or class. The record size parameter is used during code generation to determine by how much to increment the pointer within the buffer. The total size of the buffer is given as a number of pages. Depending on the performance of a particular tool and the relative amounts of work collecting and analyzing the data, the user may choose a different size for the buffer. The callback function is invoked any time the buffer overflows; details for handling overflow will be discussed later. Additionally, the callback function is invoked on thread exit, which allows the user to drain the buffer of any values inserted between the last overflow and the exit.

The callback function is called with a pointer to the base of the overflowed buffer, the number of elements, information about the application state including the thread ID and architectural context, and the optional pointer from `PIN_DefineTraceBuffer`. The specified buffer size is allocated for each thread, in a thread-private manner, so the thread ID allows the user to analyze each thread separately

or record information about thread interleaving.

The user may define multiple buffer types; for example, the user may create one buffer type to collect a memory trace, and a second buffer type to collect a branch trace. We differentiate these using a new ID type. The ID of a buffer is also delivered to the user in the callback. This allows the user to consolidate their callback functions; it is also used in the memory management functions described later, if the user wishes to manage the buffer in the callback.

Buffer filling: The new API elements for filling a buffer are parallel to many of the existing instrumentation calls as described in Section 2. For instance, instead of `INS_InsertCall`, present in the existing API, we provide `INS_InsertFillBuffer` for inserting a buffer fill around an instruction. As with `InsertCall` functions, there are different versions for instrumenting with respect to an instruction, basic block, routine, etc., as well as a version for if/then instrumentation (*i.e.*, the user can opt to only insert data to the buffer if a certain condition is true). The `InsertFillBuffer` functions also support the various `IPOINT` directives, which instruct Pin where to insert instrumentation code with respect to the instruction or basic block being instrumented.

There are two primary differences between the `InsertCall` and `InsertFillBuffer` functions. First, rather than providing a pointer to an analysis function, the user provides the ID of the buffer they would like to fill. This is used internally during code generation to find the size of the record to advance the buffer pointer. Second, where the user would provide arguments to an analysis call, they must also provide the offset into the record. This is necessary for generating the fill code to make sure the correct values go to the correct places in a struct; otherwise, the user may get nonsensical values when they process the buffer later. It has the additional benefit of allowing the user to specify the fields for a particular `InsertFillBuffer` call in the order that makes the most sense at that time; furthermore, they may only fill the parts of the record they care about. For instance, rather than having two separate buffer types for memory and branch traces as described previously, the user could define their record to store information about both, along with a flag; by requiring them to specify the offset for the fields they care about, they need not write code in their tool to fill unimportant fields. Likewise, when Pin generates the actual code to fill the buffer, it does not have to generate fill code for fields the user does not care about for that particular instruction, which reduces both the instruction count and memory traffic.

Managing buffer memory: For each buffer type, one physical buffer is allocated automatically when a thread begins execution. In addition to defining multiple buffer types, the user may request allocation of multiple buffers for any given buffer type. This allows the user to have a second buffer, or an arbitrary number of buffers, so one or more processing threads may work on analyzing the data while the main application thread(s) continue to run and collect data. This multi-buffering can be used with a technique like PiPA [15] as well to break data up into smaller chunks to be processed further by other threads.

We provide two API calls for explicitly managing buffers, `PIN_AllocateBuffer` and `PIN_DeallocateBuffer`. To allocate a new buffer, the user only needs to provide the ID for

the type. For deallocation, the user needs to give both the ID and address of the base of the buffer. Pin maintains a list of the memory associated with each buffer type, so it will not free memory from an address that does not match the base of an appropriate buffer. If the user has ever allocated their own buffers, Pin can not safely automatically free the memory, so the user is responsible for deallocating buffers to avoid memory leaks.

The user explicitly manages which buffer to write after an overflow by returning a pointer from the callback. In the single-buffer case, the user will just return the pointer to the buffer they received in the callback. For multiple buffers, the user may either allocate a new buffer each time and free the processed buffer at the end of the processing thread, or they may allocate some number of buffers in advance and rotate through them. Which of these is ideal depends on the processing task: if processing takes considerably longer than filling, the application may stall while waiting for an empty buffer; however, creating too many processing threads and extra buffers can quickly exhaust memory and execution contexts, degrading overall performance.

3.2 Code Generation

As the application is running, Pin recompiles the original application binary to include the specified instrumentation. When analyzing data online, this basically involves generating code to set up arguments for the analysis function in registers and on the stack, then either inlining the analysis code or generating a call to the analysis function. Inlining analysis functions is not always possible, and can have a negative impact on performance. Additional time is required to recompile and allocate registers for the analysis function, and a larger, frequently inlined function can reduce instruction cache locality. Alternately, calling the analysis function can be slow, as it requires saving and restoring at least part of the application state.

When generating code to buffer data, we can largely mitigate both of those issues. We guarantee the buffer fill code is inlined, which avoids the slower call to an analysis function. Furthermore, we have a known small number of instructions required for filling the buffer, reducing the impact on code size, compile time, and instruction cache effects.

We store the pointer to the current location in the buffer in a virtual register, where the set of virtual registers available in Pin is larger than the set of architected registers. Depending on the current mapping from virtual to physical registers, we may first need to generate a spill and fill to get the buffer pointer into a physical register; this is handled automatically by Pin’s register allocator. The basic approach is to move the analysis argument – effective address of a load, branch target, etc. – into a register, and then move the argument from that register to the specified offset from the buffer pointer. After filling all relevant fields of a record, we advance the buffer pointer by the size of one record. One simple optimization we make here is to use an `lea`, rather than an `add`, to advance the pointer. This avoids modifying the `eflags` register, which could affect program execution, or having to save and restore it, which has been demonstrated in the past to be slow.

We show a sample of this code in Figure 2, taken from instrumentation code generating a trace consisting of the program counter, the effective address, the size of the reference, and a flag indicating load or store. In the code, `r14`

```

mov r8, 0x0 ; flag: load
mov qword ptr [r14+0x14], r8 ; fill argument 3
mov r8, 0x8 ; size of reference
mov dword ptr [r14+0x10], r8d ; fill argument 2
lea r8, ptr [rsp-0x8] ; EA of reference
mov qword ptr [r14+0x8], r8 ; fill argument 1
mov r8, 0x3668c01147 ; original PC
mov qword ptr [r14], r8 ; fill argument 0
lea r14, ptr [r14+0x18] ; advance pointer

```

Figure 2: Sample buffer fill code. Here, `r14` stores the current buffer pointer, and `r8` holds analysis arguments.

Fill Method	Bandwidth (GB/s)
gmemset	2.38
serial	1.58
frame	1.58
serialnti	2.43
fragmenti	2.40

Table 1: Fill bandwidth for combinations of fill instruction and buffer pointer advancement scheme. Higher is better.

has already been filled with the value of the current buffer pointer, and `r8` (or `r8d`, the lower 32 bits of `r8`) has been used to store each of the analysis arguments to be added to the buffer. As shown, we only need two instructions per analysis argument plus one to advance the buffer pointer.

We considered two dimensions for code generation: which store instruction to generate, and how often to advance the buffer pointer. In terms of how to actually store the buffer, we considered using `mov` as shown in Figure 2, or `movnti`, a non-temporal store instruction from the SSE instruction set. The non-temporal instructions essentially tell the processor the data being written will not be accessed soon, removing the need to load the cache line if it is not already present in the first-level data cache. Depending on the specific microarchitecture, the line may or may not be loaded into the last-level cache.

There are two options for advancing the buffer pointer: advance each fill, or advance once per basic block or trace. Advancing once per fill is very straightforward and means the effective offsets are the same for each fill. However, it can have a larger overhead in terms of instructions, depending on the number of buffer fills per basic block or trace. Advancing once per basic block requires keeping track of the number of fills executed so far, so the offsets can be shifted appropriately. For instance, if the record is 16 bytes and the pointer is advanced once per basic block, the second fill in a basic block must have all the offsets adjusted by 16 bytes, the third fill by 32 bytes, and so on. Additionally, if the basic policy is to advance once per trace, all branches that exit early from the trace must be modified to advance the buffer pointer an appropriate amount before branching.

Our main concern with respect to the code generated for filling the buffer and advancing the pointer was to ensure buffer filling is not bandwidth-limited – if the memory bandwidth is saturated, no other changes will improve performance. We used a microbenchmark to test the maximum fill bandwidth for several configurations; this comparison is

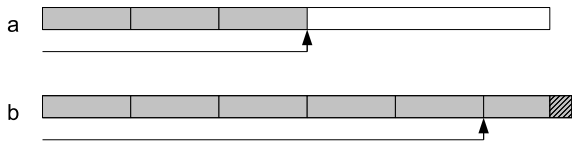


Figure 3: Demonstration of buffering. In (a), the buffer is partially full and the buffer pointer points to empty space. In (b), the write at the buffer pointer overflows the buffer, triggering an overflow.

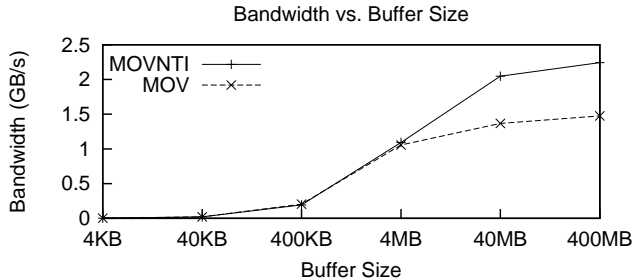


Figure 4: Bandwidth with respect to buffer size on an ideal application. For small buffers, frequent overflows limit performance.

shown in Table 1. The bandwidth is approximated as the average over 15 executions of filling a 40MB block of memory, using the time stamp counter to measure time. We use `memset` from `glibc`, a highly-tuned implementation of filling memory, as a baseline. We compare filling with either `mov` or `movnti`, and either advancing the pointer after each fill (“serial”) or once every four fills (“frame”). From the graph, we see `movnti` gives performance roughly equal to `memset`, whereas `mov` only reaches approximately 65% of the maximum. Furthermore, the two methods for advancing the pointer are equivalent.

Based on this, we initially chose to use `movnti` with the “serial” advancing scheme as our baseline. However, as we will detail further in Section 4, performance degrades on real applications. This is due to `movnti` performing best when moving full, aligned, cache lines, which is an assumption broken by interleaving application stores or even non-aligned buffer fills.

3.3 Handling Overflow

The final major component of the buffering implementation is handling overflow. For some instrumentation tasks, applications, and buffer sizes, the initial chunk of memory allocated for the buffer will hold the entire application’s data, and the user’s callback only needs to be called on termination. However, on most interesting cases, the user’s instrumentation routines will generate many buffers of data.

Our basic strategy is to allocate an additional page at the end of the buffer, without read or write permissions, to act as a guard page. When the buffer is full, the next attempt to write into the buffer will land on this guard page and generate a signal for a segmentation fault. Figure 3 depicts this case. For transparency, Pin must catch all signals and either reconstruct the original application state or report the signal as an internal fault. We add an extra case to the signal handler for segmentation faults; if Pin is unable to

find an original application instruction matching the faulting instruction in the code cache, we check the address of the write against the guard page ranges for currently-allocated buffers. If the write falls in a guard page, we can easily obtain the necessary information for calling back to the user, including the ID of the overflowing buffer and the number of records stored in it. In this case, the number of records is simply the size of the buffer divided by the specified record size.

The primary issue with this scheme is the overhead of handling the signal, including both catching the signal and reconstructing the state. Reconstructing the state essentially requires recompiling the current trace, including all instrumentation, which becomes a significant source of overhead if done frequently. This can be mitigated by using a larger buffer, which reduces frequency of overflows but which scales poorly when multiple threads are involved. Figure 4 shows the bandwidth for filling a buffer with both `mov` and `movnti` at several sizes, on an idealized application where the only memory traffic is from filling the buffer. In this case, wall-clock performance closely follows bandwidth performance; the primary point to note is how quickly performance drops off as the buffer size decreases.

To mitigate this, we implemented multiple “high-water mark” checks for the buffer. This basically means we check at various points, preferably infrequently, to see how close to overflowing each buffer is. If the buffer is within a certain distance of the end, we force an early overflow without needing to recompile to reconstruct the state.

We implement this in two ways. The first is to check if the buffer is past the high-water mark every time the VM is entered, which happens for various reasons such as compiling new code or handling a system call. This method works well in some cases, avoiding as many as nearly 95% of signal handling overflows, but avoids less than 3% on average. This is due primarily to the fact that in the ideal case, which is realized on some applications, all of the application code is compiled early and most of the execution is spent executing from the software code cache.

The second implementation is periodic if/then instrumentation. As described in Section 2, we can execute a small boolean function quickly and execute a more heavyweight function if the boolean function evaluates to true. In this case, our “if” function is whether the buffer is above the high water mark, and the “then” function mimics an overflow. Because these “if” calls generally happen more frequently than VM entries, this method generally leads to more early overflows than the VM entry check.

Pathological cases exist, such as where an instrumented instruction could appear in a loop nest and not be monitored by an if/then. This would lead to all overflows being handled by catching a signal. One way to attempt to avoid this case is to recompile any trace which causes a full overflow and explicitly insert the if/then instrumentation. We will discuss this further in Section 4.

3.4 Summary

In this section, we described the implementation details of buffering in Pin in three major parts: the user interface, code generation, and handling buffer overflow. The user interface for buffering is implemented as additions to Pin’s instrumentation API, allowing the user to define multiple buffer types, insert buffer fill instructions in a variety

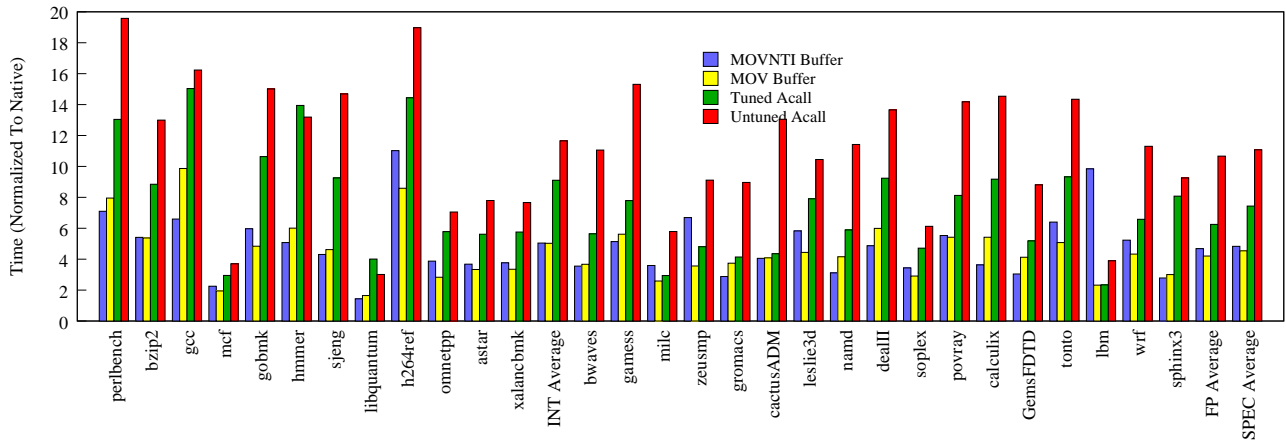


Figure 5: Runtime comparison of buffering implementations, comparing the implementation described in this paper with each instruction type to two analysis call-based implementations. For either fill instruction, our implementation performs better than analysis call-based implementations.

of manners, and manage memory for filling and processing buffers in parallel. These additions generally parallel the existing Pin API. Fill code is generated as pairs of instructions to calculate an analysis argument into a register and then store it into the buffer. These fill instructions must then be followed by an instruction to advance the buffer pointer. We discussed choices for buffer fill instruction and buffer pointer advancement scheme. Based on bandwidth experiments, we use as a baseline filling the buffer with the SSE `movnti` instruction and advance the pointer with an `lea` after each record. Finally, we described our method for dealing with overflow, catching a signal due to writing into a guard page at the end of the buffer, and two methods to avoid the overhead associated with signal handling.

4. Performance Evaluation

In this section we will present performance results for our buffering API. We first present a comparison of run times for basic implementations based on both fill instructions compared to unoptimized and optimized analysis call-based implementations. We then compare the effects of our overflow-handling optimizations.

The results were collected on a 1.6GHz dual-processor Intel Core2 Quad with 8GB of RAM running 64-bit CentOS 4.7. We report results using two benchmark suites, SPEC-CPU and SPEC-COMP. For SPEC-CPU, we used the reference inputs to SPEC2006 v1.1, compiled with `gcc 4.1.2` with optimization `-O2 -fPIC`. For SPEC-COMP, we used the medium inputs for SPEC-COMP 2001, compiled with `icc 10.1.013` with optimization `-O3`. SPEC-COMP benchmarks were run with eight threads.

Measuring buffering performance only makes sense in the context of using it in a PinTool. Our comparisons are based on a PinTool that collects a memory trace of effective addresses of loads and stores – that is, each record is a single 64-bit value. Since we only care about the performance of the buffer, no analysis is performed on the data.

4.1 Basic Implementation

We first demonstrate the performance of our buffering implementation without the modifications to improve overflow

handling. Figure 5 compares the overhead of collecting a memory trace, as described in the beginning of the section, using four methods. The first two, “MOVNTI Buffer” and “MOV Buffer,” use the buffering API described in this paper with different fill instructions. The last two, “Tuned Acall” and “Untuned Acall” are implementations using the pre-existing Pin API. As a point of reference, the two versions using our API require approximately 75 lines of user code, the untuned analysis call requires approximately 50 lines, and the tuned analysis call implementation requires roughly 500 lines. The tuned analysis call implementation is included with the Pin kit as the `memtrace` tool and uses `if/then` instrumentation before each fill to determine whether the buffer will overflow on the current fill. If the buffer would overflow, it first processes the buffer and resets the pointer before continuing. In this case, the “if” call can be inlined. The untuned analysis call implementation performs the overflow check in the same function as the buffer fill and can not be inlined.

As mentioned previously in Section 3.2, although performance is better in the ideal case when filling the buffer with `movnti`, on SPEC applications buffer filling with `mov` is faster on average and provides a 2.6x speedup compared to `memtrace`. However, it is worth noting that using `movnti` provides the lowest overhead and highest speedup compared to `memtrace` on a single benchmark: a 43% overhead and nearly a 3x speedup on `462.libquantum`. The untuned analysis call implementation also performs better than `memtrace` on this benchmark, so it is possible the memory access patterns for `libquantum` are a pathological case for `memtrace`.

In many of the cases where the `mov` buffer is faster than the `movnti` buffer, the difference is relatively small. However, on `470.lbm`, we see both the `mov` buffer and `memtrace` have roughly a 2.3x overhead while the `movnti` buffer incurs nearly a 10x overhead. After investigation using hardware performance counters, we discovered this was due to how non-temporal stores update memory. Non-temporal stores rely on write-combining full cache lines to improve performance; interleaving non-aligned stores, either from Pin or the application, limits write-combining and thus performance.

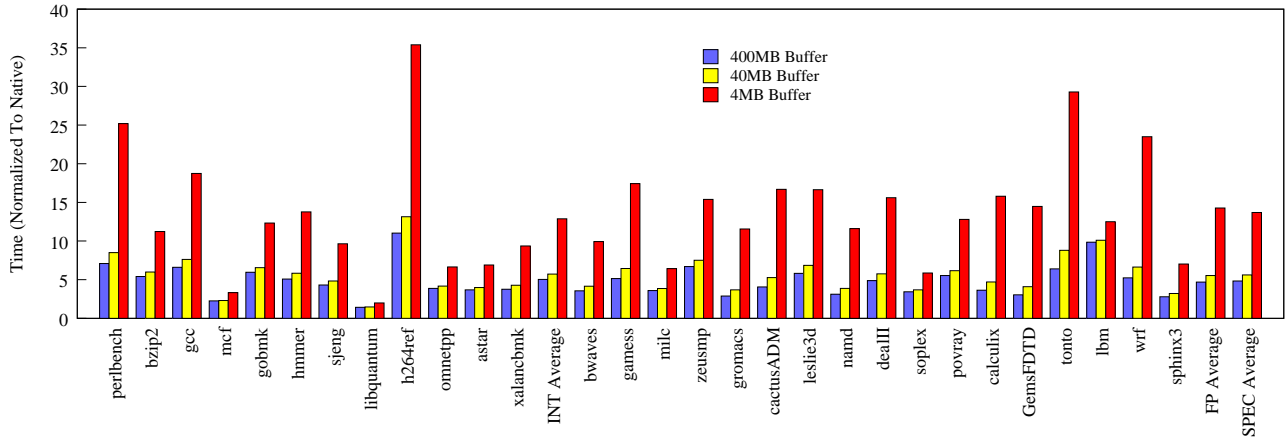


Figure 6: Overhead of a movnti-based buffer with different buffer sizes. Performance degrades slightly when decreasing the buffer size from 400MB to 40MB, but degrades more noticeably when decreasing from 40MB to 4MB.

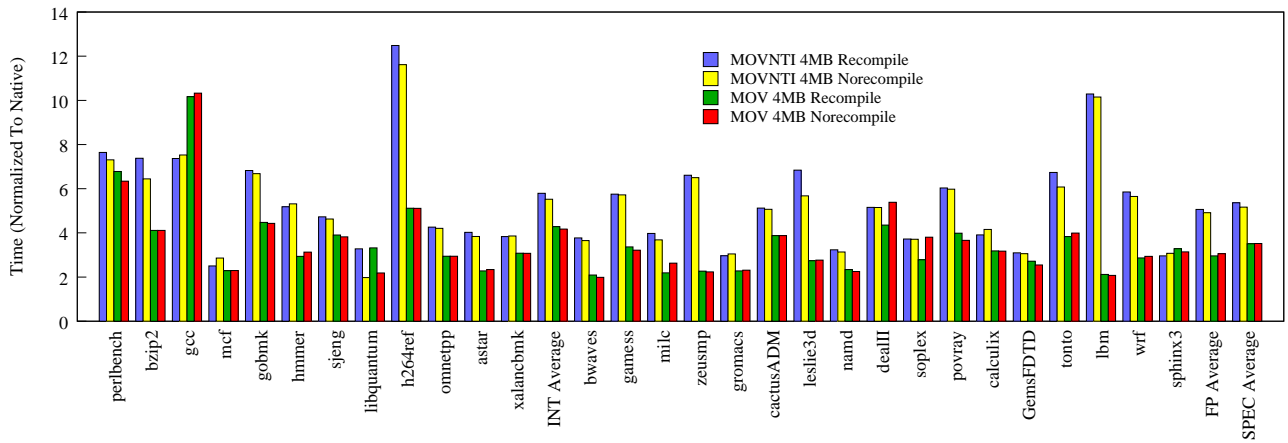


Figure 7: Buffering overhead after applying early overflow optimizations, triggering an overflow if the buffer is more than 50% full at check time. Small buffers now perform better than larger buffers.

We now compare the effect of buffer size on overhead. As mentioned in Section 3.3, memory bandwidth on an ideal application drops off rapidly as the buffer size is decreased. Figure 6 compares the run time of filling a buffer with `movnti` with buffer sizes of 4MB, 40MB, and 400MB. The results follow what one might expect from the bandwidth comparison of Figure 4; there is a large increase in performance from 4MB to 40MB, and less of an increase from 40MB to 400MB. This allows the user to make a tradeoff between a large buffer for increased performance on applications with few threads, or smaller buffers for many-threaded applications or applications where memory issues could arise.

4.2 Optimizing Overflow

Although the user has the option of increasing performance by increasing the buffer size, this is not possible in all cases. For instance, large buffers do not scale to applications with many threads; furthermore, they may lead to thrashing in memory. Since the increased frequency of overflows is the primary difference in behavior with varying buffer sizes, we will now discuss the performance benefits of modifying overflow behavior as described in Section 3.3.

Recall that we implemented multiple methods for causing an early overflow to avoid handling a signal and reconstructing the state, including checking when entering the VM, checking with periodically-inserted if/then instrumentation, and potentially recompiling traces which still overflow due to a segmentation fault. The if/then instrumentation is effectively the same as that described for the analysis call-based `memtrace`, although we automatically add the checks rather than the user explicitly inserting them.

There are a number of parameters which can be varied for these methods, including how full the buffer can grow before an early overflow is desirable (the “high-water mark”), how often to insert if/then calls, whether the time spent recompiling the trace is worthwhile, and whether both the VM-entry check and the if/then check work well together. We omit a full exploration of these parameters but summarize here. For nearly all variations of the parameters, noticeably better performance was obtained using a 4MB buffer. Although the best high-water mark and if/then frequency varied across benchmarks, the best case on average was with the high-water mark set at 50% and inserting if/then checks every 50 fills. Using these parameters along with an additional high-water check every time the VM is entered, we vary the fill instruction and whether to recompile a faulting trace with the if/then instrumentation.

Based on those initial studies varying the parameters, we selected four cases to perform a larger comparison on. Figure 7 compares the four combinations of remaining parameters, fill instruction and recompilation. There are two interesting points to take away from the comparison. First, as in Figure 5, the `mov`-based buffer performs better on average than the `movnti`-based buffer, but the gap is much more pronounced. Still, there are cases where `mov` performs worse, which generally parallel the performance prior to the overflow optimizations. The second interesting point is that although recompilation of faulting traces occasionally improves performance, in general it has a negative or negligible impact. Since the intent behind recompiling faulting traces was to instrument loops which may otherwise miss being checked, a possible cause for slowdown here is that making the check every time through the loop may have a

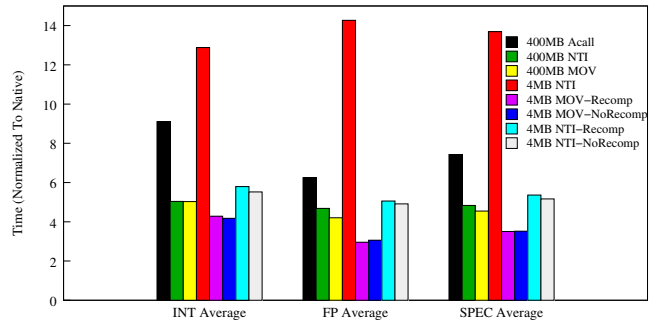


Figure 8: Summary of overhead for various buffering implementations. Initially, large buffers performed well but small buffers performed poorly. Optimizing overflow significantly improves small-buffer performance.

higher total overhead than the overhead of dealing with the overflow. A more sophisticated method that allows tuning how often to perform the check may improve performance.

Figure 8 summarizes the improvements of optimizing overflow handling. Bars 2-4 show the buffering API prior to optimizing overflow. While the large buffer leads to a 2.6x improvement over the analysis call implementation, the small buffer has much poorer performance. After optimizing overflow, the small `mov` buffer achieves nearly a 4x improvement over the base. The smaller `movnti` buffer performs better after optimization, but still has a higher overhead than the 400MB buffer.

4.3 Multithreaded Applications

We now consider the performance on multithreaded benchmarks. Figure 9 shows a performance comparison between the 400MB `movnti`- and `mov`-based buffers, along with the tuned analysis call implementation. The overflow optimizations were not used here; the optimized version will be discussed later. In contrast with the single-threaded case, the `movnti`-based buffer performs better on average with these benchmarks. As discussed previously, `movnti` has a better potential for performance, dependent upon memory use characteristics of the analyzed program, so this difference simply demonstrates a difference in memory use from the single-threaded benchmark suite.

Figure 10 compares the performance using the `movnti` buffer at buffer sizes of 400MB, 40MB, and 4MB. As before, the decrease in size leads to a small decrease in performance at 40MB and a larger decrease in performance at 4MB. The difference here is much more pronounced due to the multithreaded nature of the applications being analyzed. Any time an overflow happens, the signal handler prepares to search for the relevant code section and buffer information. This requires taking a lock on the VM, to prevent issues such as another thread deleting internal data structures needed by the signal handler. The threads in OpenMP programs will generally be performing the same actions and thus will overflow at roughly the same time, making overflow handling act as a serialization point.

Figure 11 compares the performance on multithreaded applications after applying the best optimization from the single-threaded case – a high-water mark set at 50%, with checks set every 50 fills and also on VM entry. As with the

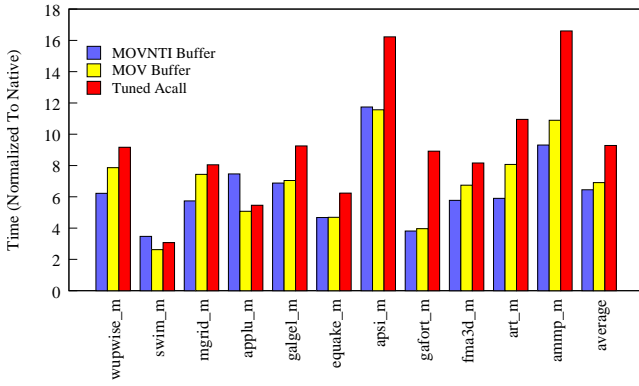


Figure 9: Buffering overhead on multithreaded benchmarks. Write-combining effects cause movnti to perform better here.

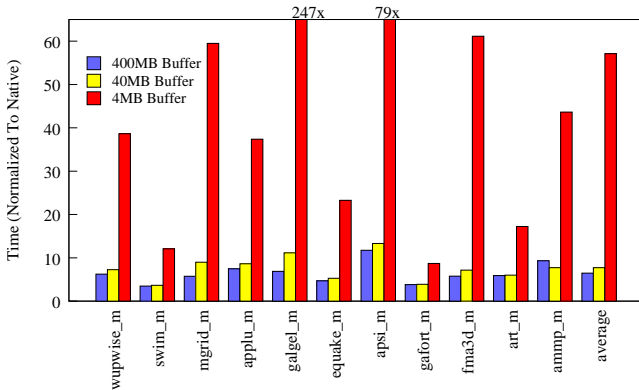


Figure 10: Buffering overhead with respect to size on multithreaded applications. Performance degradation of smaller buffers is more pronounced in the multithreaded case due to lock contention.

single-threaded case, performance is generally greatly improved, in many cases better than with the large buffer. On `art_m`, lock contention is still an issue.

4.4 Summary

In this section we have presented performance results for our buffering implementation. Our basic implementation had a roughly 2.6x improvement over the previous best-known implementation. However, performance rapidly degraded as the buffer size decreased, due to overhead associated with processing buffer overflows. After implementing several optimizations, we achieved nearly a 4x speedup over the best-known implementation even with a much smaller buffer. On multithreaded applications, we achieved roughly a 3.4x improvement after optimizations.

5. Related Work

The work presented in this paper shares with many other works the goal of reducing overhead of instrumentation and analysis, orthogonally to the overhead of the dynamic instrumentation system itself. Wallace and Hazelwood presented SuperPin [13], which parallelized analysis in slices

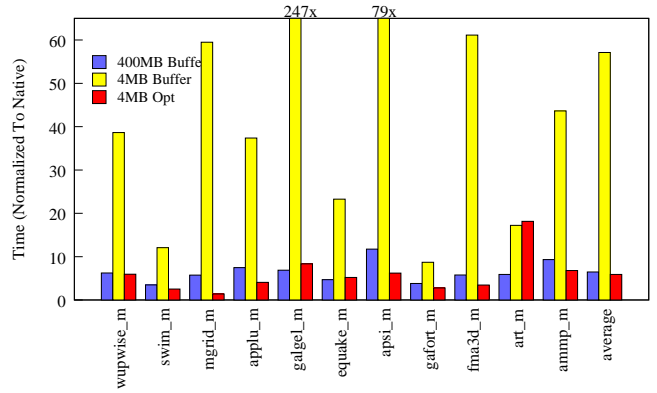


Figure 11: Comparison of buffering overhead with optimizations. As with the single-threaded case, early overflow optimizations generally greatly improve performance of small buffers.

with the unaltered application. Moseley *et al.* presented shadow profiling [9], which parallelized sampling with an unaltered application to reduce the overhead of profile-guided optimization. Systems such as ADORE [5] and Trident [14] reduce the overhead of profiling for dynamic optimization using performance monitoring hardware.

The work most similar to ours is PiPA, presented by Zhao *et al.* [15]. PiPA is a technique for reducing the overhead of analysis by collecting data in a lightweight manner and passing it to one or more external threads for processing. They described as their lightweight collection method filling a small buffer with data, then passing that to a thread which could then either process the data or further distribute the data amongst other threads. The buffer they described is similar to `mementrace` as described in Section 4. Shen and Shaw [12] also used a PinTool-based buffer using similar techniques. Our work provides a more thorough exploration of the design space and automates management tasks which must be performed explicitly by the user in their implementations.

6. Conclusions and Future Work

We have presented an implementation of buffering data for decoupled analysis in Pin. Buffering information about a program of interest can be performed in a lightweight manner to limit instrumentation overhead, and analysis in bulk reduces the overhead of switching from the context of the application to the context of the analysis routines. We described in detail our implementation, including additions to Pin’s instrumentation API, code generation, and handling overflow. We then characterized the performance of the system, beginning with our unoptimized implementation which achieved on average a 2.6x improvement over the previous best-known method. After optimizing overflow handling, we were able to decrease the buffer size while improving our performance to nearly a 4x improvement over the previous best-known method.

Two potential avenues of future work are improving the performance of invalidation and recompiling traces and avoiding lock contention. Performance of invalidation and recompilation may be improved by using a more sophisticated

method to check for early overflow in loops, such as increasing check frequency depending on whether the loop tends to lead to overflow. Avoiding lock contention requires decreasing the size of critical sections or avoiding the use of thread-shared data structures. These and other improvements can make buffering for decoupled analysis an even more useful tool for studying program behavior.

7. References

- [1] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, Dec 2000.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Programming language design and implementation*, pages 1–12, Vancouver, BC, 2000.
- [3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *36th Symposium on Microarchitecture*, San Diego, CA, 2003.
- [4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, CA, Mar 2003.
- [5] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *1st Symposium on Code generation and optimization*, pages 79–90, San Francisco, CA, 2003.
- [6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *1st Symposium on Code generation and optimization*, pages 15–24, San Francisco, CA, Mar 2003.
- [7] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, Aug 2002.
- [8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation*, pages 190–200, Chicago, IL, 2005.
- [9] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *5th Symposium on Code Generation and Optimization*, pages 198–208, San Jose, CA, 2007.
- [10] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.
- [11] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Sofa. Reconfigurable and retargetable software dynamic translation. In *Code Generation and Optimization*, pages 36–47, San Francisco, CA, Mar 2003.
- [12] X. Shen and J. Shaw. Scalable implementation of efficient locality approximation. In *21st Workshop on Languages and Compilers for Parallel Computing*, Edmonton, AB, July 2008.
- [13] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Symposium on Code Generation and Optimization*, pages 209–220, San Jose, CA, 2007.
- [14] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *14th Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, St. Louis, MO, 2005.
- [15] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: pipelined profiling and analysis on multi-core systems. In *Code generation and optimization*, pages 185–194, Boston, MA, 2008.