Analyzing Dynamic Binary Instrumentation Overhead

Gang-Ryung Uh Robert Cohn† Bharadwaj Yadavalli† Ramesh Peri† Ravi Ayyagari
BOISE STATE UNIVERSITY INTEL†
{uh,rkayyagari}@cs.boisestate.edu {robert.s.cohn,s.bharadwaj.yadavalli,ramesh.v.peri}@intel.com

Abstract

Robust and powerful software instrumentation tools are essential for dynamic program analysis tasks such as profiling, performance evaluation, and bug detection. *Dynamic binary instrumentation* (DBI) is a general purpose technique that eases the development of program analysis tools by facilitating automatic low-level instrumentation. DBI-based program analysis can introduce high overhead and it is crucial for tool writers to minimize the cost. Analyzing the performance of instrumentation tools is challenging because most systems use a *just-in-time* compiler (JIT) to dynamically generate code. In this paper, we describe our method for analyzing the performance of instrumentation tools. The instrumented code is itself *instrumented* with basic block counters. We implement the profiler in Pin and use it to analyze the behavior of simple and complex instrumentation tools. The analysis yields several unexpected results about the dynamic behavior of instrumented programs. By examining these results, we often find effective solutions to improve performance.

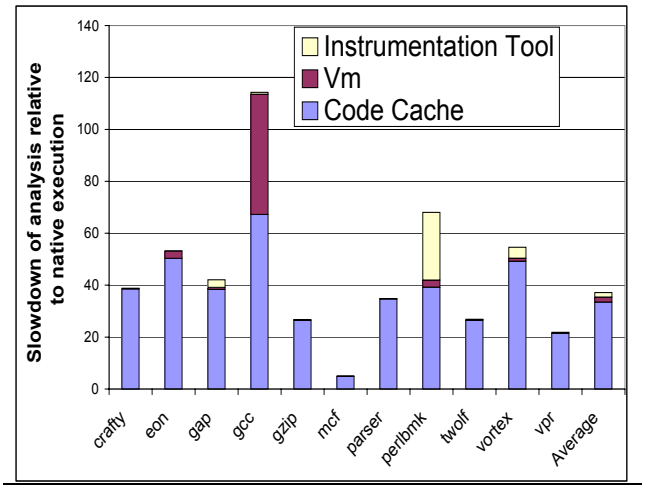

Figure 1. Slowdown for Memory Analysis normalized to original application time

1. Introduction

Dynamic binary instrumentation (DBI) is a popular technique to analyze the runtime behavior of software. DynamoRIO [7], Valgrind [6], and Pin [5] are widely used DBI systems that provide a general API to facilitate the development of instrumentation tools. Users of these systems have written tools that simulate caches [12], detect memory allocation errors [6], detect security violations [13], model system performance [14], etc. Since the usability of a DBI-based program analysis tool depends heavily on its speed, tool developers are particularly interested in improving the performance of instrumentation [11].

DBI-based instrumentation can be slow for complex program analysis. As a concrete example, Figure 1 shows the slowdown when using a representative Pin-based [5] program analysis tool on SPECint programs [8]. This tool is used internally at Intel on production applications to analyze memory reference behavior. It performs sophisticated memory analysis for every *load*, *store*, *call*, and *return*. As indicated in the figure, the average slowdown for SPECint programs is 38 times execution of the uninstrumented binary. The slowness of the Pin-based tool limits the size of the workload we can use to exercise the instrumented application.

The common techniques for understanding the performance of applications cannot easily be applied to DBI-based program analysis tools. This is because Pin and all the other widely used DBI systems employ a *just-in-time* compiler (JIT). For best performance, the JIT intermingles dynamically generated code for the original application, inserted instrumentation code and other book-keeping code for the DBI. To illustrate this point, we used *VTune* [10] to profile the execution of programs instrumented with the memory reference analyzer. We analyzed the data to break down the time into 3 categories: *instrumentation tool* (**IT**), *code cache* (**CC**), and *virtual machine* (**VM**).

The IT time includes the on-line processing of data collected by instrumentation. The CC time is all of the dynamically generated code, which includes the original application code and the parts of the on-line analysis that the JIT was able to inline. The VM time is the JIT, or time to dynamically generate and optimize code. As show in Figure 1, 5% of the time is spent in VM, 4.6% in IT, and 88% of the time in CC on average.

The VM and IT are conventional binaries and can be analyzed with standard profilers such as VTune. However, the CC time is the most important for this instrumentation tool because it dominates the total execution time. Unfortunately, it cannot easily be examined with a profiler because it has no symbol information. Furthermore, generated code basic blocks are a mix of original application and instrumentation code. It is difficult to know whether the CC time is high because the instrumentation tool inserted a lot of extra work or because the DBI system has generated inefficient code. This becomes more difficult for instrumentation tools with complex analysis and DBI optimizations like inlining and register allocation.

This paper describes a technique for more detailed analysis of time spent in CC. Section 2 describes Pin, the instrumentation system we analyzed. Section 3 presents our analysis technique for the CC time, which instruments the instrumented code itself. Section 4 discusses alternatives for analyzing CC time. Experimental results with our analysis technique are presented in Section 5, followed by conclusions in Section 6.

2. Overview of Pin

The goal of Pin [5] is to provide an instrumentation platform for building a wide variety of program analysis tools for multiple architectures. The Pin API makes it possible to write tools (called Pintools) that reside in the same address space as an executing application and observe all the architectural state of a process, such as the contents of registers, memory, and control flow. It uses a model similar to ATOM [4], where the user adds procedures (called analysis routines in ATOM) to the application process, and writes instrumentation routines to determine where to place calls to analysis routines. The arguments to analysis routines can be architectural state or constants. Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite application registers and application memory.

Pin's call-based model of instrumentation is simpler than other tools where the user inserts instrumentation by adding and deleting statements in an intermediate language. However, it is equally powerful in its ability to observe architectural state and it frees the user from the need to understand the idiosyncrasies of an instruction set or learn an intermediate language. The inserted code may overwrite scratch registers or condition codes; Pin automatically saves and restores state around calls so these side effects do not alter the original application behavior. The Pin model makes it possible to write efficient and architecture-independent instrumentation tools, regardless of whether the instruction set is RISC, CISC, or VLIW.

The call-based model makes it easier to write tools, but can potentially lead to poor performance. A combination of inlining, register re-allocation, and other optimizations attempt to make Pin's procedure call-based model as efficient as lower-level instrumentation models. This paper examines how well it achieves that goal.

To provide some background for analysis of instrumentation overhead, we describe how Pin instruments code. Instrumentation is performed by a just-in-time (JIT) compiler. The input to this compiler is not bytecode, however, but a native executable. Pin intercepts the execution of the first instruction of the application and generates ("compiles") new code for the straight-line code sequence (trace) starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time the JIT fetches some code, the Pintool has the opportunity to instrument it before it is translated for execution. To improve performance, the translated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions.

A trace is a straight-line sequence of instructions which terminates at: (i) an unconditional control transfer (branch, call, or return), (ii) a pre-defined number of conditional control transfers, or (iii) a pre-defined number of instructions. In addition to the last exit, a trace may have multiple side-exits (the conditional control transfers). Each exit initially branches to a stub, which re-directs the control to the VM. The VM determines the target address (which is statically unknown for indirect control transfers), generates a new trace for the target if it has not been generated before, and resumes the execution at the target trace. The original trace exit is back patched to branch to the newly generated trace to avoid entering the VM the second time it is executed.

3. Analyzing Overhead Via Self Instrumentation

To accurately analyze the overhead in the CC time, fine-grained profiling needs to be performed. While it is easy to collect a profile of the original application, it is more difficult to analyze the code that executes in the CC. To collect a profile, we chose to modify the JIT to instrument the code that it generates with basic block counters.

When the JIT generates code, every instruction is annotated with its source. The most coarse categorization is *application*, *instrumentation*, or *DBI*. When code generation is complete, the JIT allocates the C structure in Figure 2 for each basic block. Using the annotations, it initializes the fields of the structure with the static count of instructions of each class. It also inserts code in the basic block to increment the basic block execution count (`_blk`) in a thread-safe manner.

After execution of the instrumented application terminates, Pin outputs a summary of dynamic counts for each instruction class, as well as list of all basic block execution counts and instruction count classification. Figure 3 shows

```
typedef struct {
   UINT64 _blk;   // block execution frequency
   UINT16 _appl;  // number of application instructions
   UINT16 _instr; // number of instrumentation instructions
   UINT16 _DBI;   // number of DBI generated instructions
   …              // further breakdowns of _instr and _DBI
   …
   UINT32 _cacheAddr;  // CC address for the basic block
} BASIC_BLOCK_INST_CLASS;
```

Figure 2. C data structure for basic block execution count

```
                DYNAMIC INSTRUCTION COUNT FROM CODE CACHE
================================================================
INSTRUCTION TYPE CLASS                        COUNT          WEIGHT
================================================================
OVERALL CLASS                          785589321905         100.00%
I.   Application                        39818585108           5.07%
II.  Pintool                           304149463234          38.72%
III. Pin Generated                     441621273563          56.22%

PINTOOL:Memory analysis tool           304149463234         100.00%
1.  Analysis bridge                    226439280124          74.45%
2.  Inlined analysis                    77710183110          25.55%
3.  Signal location                     10987805042           3.61%

PIN GENERATED                          441621273563         100.00%
1.  Register spill                     173389899663          39.26%
2.  Register reload                    183873237979          41.64%
3.  Eflag spill                         16190813906           3.67%
4.  Eflag reload                        16068459784           3.64%
5.  Branch resolution                    4282765198           0.97%
….
….
                    PER BASIC BLOCK DYNAMIC PROFILE REPORT
================================================================
LEGEND
(I)  CACHE ADDR - BBL's entry point in Code Cache
(II) BLK        - BBL's dynamic block execution count
      1. AL         - number of application INSs in BBL
      2. PN         - number of pin generated INSs in BBL
      3. PNT        - number of pintool INSs in BBL
================================================================
CACHE ADDR:0x04a400332   BLK:664865390   AL:    0 PN:   10 PNT:    7
CACHE ADDR:0x04a4002f4   BLK:664865390   AL:    0 PN:    9 PNT:   10
CACHE ADDR:0x04a4001a6   BLK:664128799   AL:    3 PN:    3 PNT:    1
CACHE ADDR:0x04a400169   BLK:664128799   AL:    0 PN:    7 PNT:    4
CACHE ADDR:0x04a400142   BLK:664128799   AL:    1 PN:    3 PNT:    2
CACHE ADDR:0x04a40039e   BLK:562728003   AL:    0 PN:   11 PNT:    7
…
…
```

Figure 3. Detailed dynamic instruction count in the CC for Gzip program with memory analysis tool

the dynamic instruction count for gzip, as collected by Pin's CC profiler. From this information, we can determine if the CC instruction count is spent in instrumentation, DBI overhead or application. For the DBI and instrumentation categories, there is a finer-grained breakdown of different types of overhead. It is also possible to determine the hottest traces generated by the JIT and examine their overhead.

In this paper, our technique uses instrumentation to collect instruction count. However, this technique can be easily extended to account for other architectural features. For example, we can include a model of a cache in the analysis of time spent in CC.

4. Related Work

The HDTrans [1] DBI system estimates the overhead introduced from the JIT by dumping and reloading its translation cache and associated metadata. Similarly, Pin [5] estimates the overhead from the JIT by reading the real-time clock. However, to the best of our knowledge neither of these techniques can produce detailed information of the time spent in the CC.

Event or timer based sampling is an alternative to using instrumentation. Sampling has the potential to measure cost in time instead of instruction count. Since some instructions are more expensive than others, this is potentially more useful in diagnosing problems. However, we want to attribute a cost to individual instructions because out JIT intermingles instructions from the application, DBI bookkeeping, and instrumentation at a fine grain. Sampling is typically unable to attribute time at this level of detail.

Sampling can also be used to estimate the frequency of execution of basic blocks. When used this way, it is not fundamentally different than using instrumentation to count basic block execution. It is potentially faster than instrumentation, but we found that the overhead of instrumentation is tolerable. Instrumentation makes it simpler to correlate the collected counts with the annotations. Instrumentation is also simple to implement since we are already have a JIT to insert the counters.

Another alternative is to use the DBI to instrument itself by running the DBI on top of the DBI. This has the potential to provide complete visibility of VM, IT, and CC with a single tool. While it is possible for Pin to instrument itself, it executes slower than our approach to collect equivalent data. It also has the same problem as sampling in correlating the raw counts collected with the instruction annotations.

5. Experimental Results

In this section, we use the profiler to analyze the performance of applications running under pin with no instrumentation, basic block counting instrumentation, and a complex memory analysis tool. We show that different types of instrumentation have different performance related problems.

5.1 Experimental Setup

The experimental results reported in this section were collected on an 2.8 GHz Xeon^{TM} IA32 machine running Linux 2.4.21. The following software configuration was used to collect detailed dynamic instruction count in the software code cache (CC).

1. The dynamic profiling technique described in Section 3 was implemented within Pin.
2. Dynamically-linked SPECint 2000 binaries compiled with gcc 3.3.2 with -O3 were used to obtain dynamic profiles.
3. The Pin-based tools were built using gcc 3.3.2 with -O3 on Linux 2.4.21.

5.2 No Instrumentation

As a baseline, we first examine the performance of running applications with Pin and no instrumentation. This illustrates

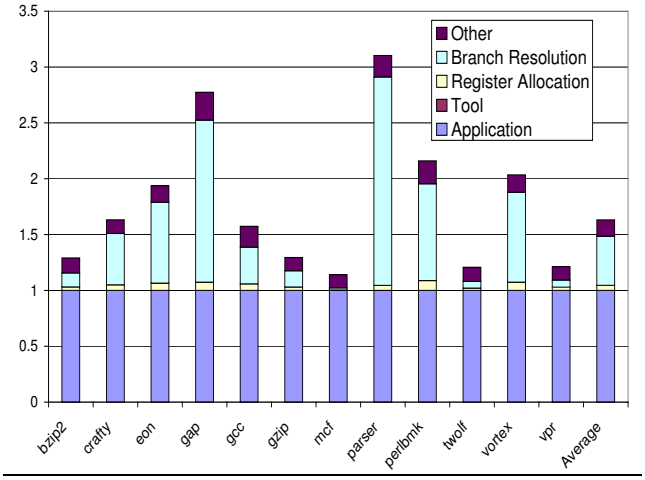

Figure 4. Breakdown of code cache instrumentation normalized to application instruction count with no instrumentation

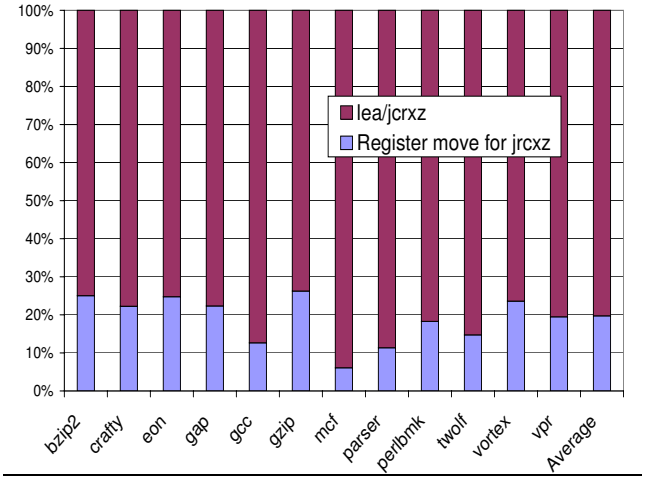

Figure 5. Breakdown of instruction count for branch resolution by cause

the overheads inherent in a DBI. Figure 4 shows the classification of instructions generated in CC while running Pin on SPECint2000. The instruction count values are normalized to the original application instruction count. On average, Pin increases the instruction count over native application execution by 60%.

The categories shown in Figure 4 are as follows. *Application* instructions are the instructions from the original instrumented application. *Tool* are instrumentation instructions. This value is 0 in Figure 4 because no instrumentation is inserted. *Register Allocation* comprises loads and stores inserted by Pin's register allocator. Pin may need to insert a load or store when it needs a free register for instrumentation or as part of the DBI translation. *Branch resolution* code is the code that Pin generates to translate an indirect branch target address to a code cache address [2]. IP relative branches are translated at compile-time and incur no overhead. *Other* includes all unclassified instructions. Figure 4 shows that branch resolution is responsible for the most overhead in the CC when no instrumentation is used.

For efficient branch resolution, Pin uses the same `lea/jrcxz`[1] idiom as DynamoRIO [7, 3]. The sequence does not modify the conditional flags, which avoids an expensive save and restore of the `eflags` register. However, the `jrcxz` instruction requires an operand to be in the `ecx` register, which may require the jit to move values. A more detailed breakdown of the components of branch resolution can be found in Figure 5.

On average, 20% of the instructions in branch resolution are register-register moves that copy the target address register to a temporary register, and the rest are the `lea/jrcxz` sequence. The move is generated when the target of an indirect branch is contained in `ecx`. When it became apparent that this was costly, we examined the code generation and found that the move could be avoided by making the register allocator more flexible and the overhead was eliminated.

[1] Jump if RCX register is 0.

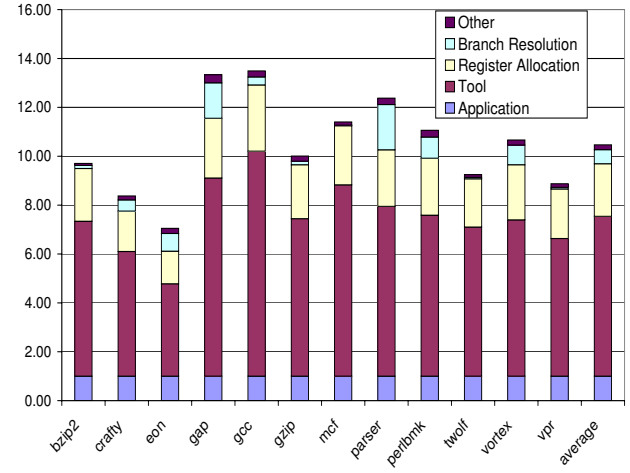

Figure 6. Breakdown of instruction count for simple basic block instrumentation, normalized to application instruction count

5.3 Basic Block Instrumentation

A typical instrumentation tool performs an action once per basic block. Examples include control flow profilers and instruction tracers. To understand the performance of such a tool, we analyze an instrumentation tool that inserts calls to an empty function once per basic block. Pin normally inlines calls to simple instrumentation. To make it possible to measure the overhead of the instrumentation call, we disable Pin's inliner for this example. The results are in Figure 6. On average, this tool increases the instruction execution count by 900%.

The biggest component is now *Tool*. When inserting a call to instrumentation, a wrapper function must be created. It allocates a stack frame, saves registers on the stack, calls the instrumentation, restores registers, and deallocates a stack frame. It is typically 34 instructions long, and adds a high fixed cost if the instrumentation does something simple like increment a counter. From Figure 6, we see that its dynamic cumulative instruction count is 6.25 times the original application instruction count, which is consistent with a 5 instruction average basic block length. We discuss the overhead of the wrapper in Section 5.5.1 and show in Section 5.4 how inlining of calls to instrumentation can dramatically reduce this overhead.

The next biggest component is *Register Allocation*. Inspection shows that much of this overhead is to swap stack pointers. To avoid polluting the stack of the application with instrumentation values, pin executes instrumentation on a private stack. In the worst case, pin must do the following for every call to instrumentation: save the application stack pointer, load the instrumentation stack pointer, call the instrumentation, and restore the application stack pointer. Pin uses a register allocator to manage the stack pointer, often avoiding unnecessary stack pointer swaps. Still, the overhead is high and we continue to investigate.

Note that branch resolution is now a relatively small overhead compared to the instrumentation related overhead. This demonstrates that while dynamic instrumentation systems are similar in architecture to binary translators and dynamic optimizers, the challenging problems for good performance can be quite different. Reducing the branch resolution overhead to 0 would not have a significant effect on performance in this scenario.

5.4 Inlined Basic Block Instrumentation

In the next example, we analyze the overhead of simple basic block instrumentation that can be inlined. Figure 7 shows the breakdown of overhead. Inlining reduces the instruction execution count from 900% to 140%.

When instrumentation is simple we can eliminate the wrapper and inline the instrumentation directly into the application code. Inlining reduces the tool contribution to overhead from from 625% to 37%. Most of the overhead in the wrapper is to save and restore registers that are potentially modified by the call to instrumentation. Inlining makes it possible for the register allocator to manage the registers and most of the saves and restores are found to be either unnecessary because the instrumentation does not touch them or redundant because the register had already been saved and not touched since a previous call to instrumentation.

5.5 Complex Instrumentation

The previous examples analyze the behavior for simple instrumentation. We next investigate the overhead for a more realistic complex instrumentation tool. Figure 8 shows the detailed overhead for the memory analysis tool described in

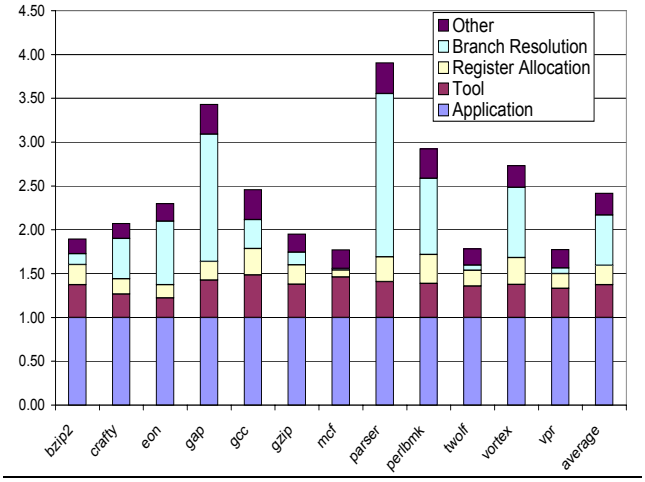

Figure 7. Breakdown of instruction count for simple inlined basic block instrumentation, normalized to application instruction count

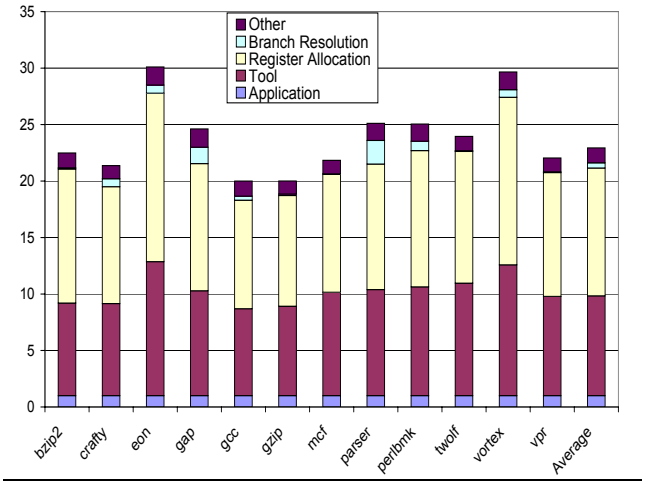

Figure 8. Breakdown of instruction count for memory analysis tool, normalized to application instruction count.

Figure 1. While the example in the previous section only inserts code once per basic block with instrumentation that can be inlined, this tool potentially instruments every load and store. It uses a combination of instrumentation that is simple enough to inline and more complex instrumentation that cannot be inlined.

As before, the instruction count values are normalized to the original application instruction count. Unlike any of the previous examples, this tool spends a large amount of its instruction count in register allocator overhead. We investigate the performance of the two major categories: *Register Allocation* and *Tool*.

5.5.1 Register Allocation

Figure 8 shows that 58% of the code cache instruction count for the memory analysis tool was contributed by register al-

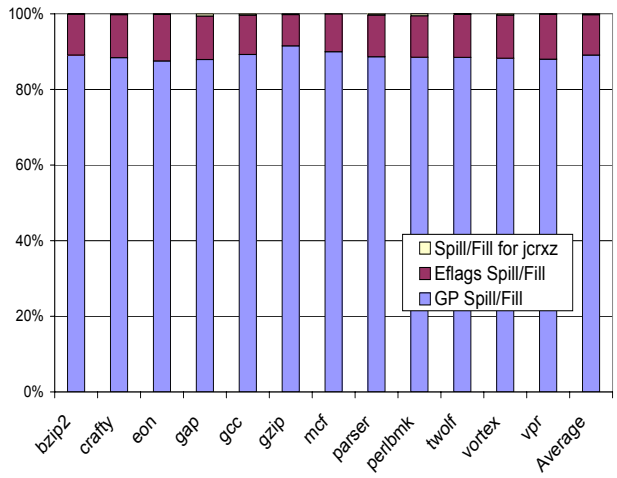

Figure 9. Breakdown of register allocation overhead with memory analysis tool

location. This is 11 times the instruction count of the original application.

Pin frequently needs extra registers during JIT compilation. For example, the code for the branch resolution needs three free registers. When Pin inlines an analysis function into original application code, or sets up a stack frame to make a call to an analysis function, the JIT must ensure that the call does not overwrite any scratch registers that may be in use by the application. Rather then obtaining extra registers in an ad-hoc way, Pin re-allocates registers for each trace, using linear-scan register allocation [9].

Since Pin's allocator does interprocedural allocation by incrementally discovering the flow graph during execution, the high register allocation overhead in Figure 8 was a surprise. To analyze this overhead, we collected a finer-grained dynamic instruction count breakdown for instructions inserted by the the register allocator. Figure 9 illustrates some interesting results.

First, spills and fills for the `eflags` register are relatively small. This contradicted our expectations. When pin inserts extra instruction for instrumentation, it must ensure that those instructions do not overwrite a live application value in the `eflags` register. Nearly every arithmetic operation in the IA-32 instruction set overwrites `eflags`. This is a difficult problem for general purpose instrumentation where it is not possible to hand craft the inserted code to avoid modifying `eflags`. Spilling the `eflags` register is more expensive than other registers. On IA-32, most registers can be spilled with a single instruction, but the `eflags` register requires 4 operations. The only way to save the entire `eflags` register is to push it on the stack[2]. Since we do not want to pollute the application stack with values from instrumentation, we must swap stack pointers so the push operation will write the `eflags` directly to its home in memory

[2] `lahf` only saves 8 bits of the `eflags`

```
1  pushfd          # save eflags
2  push    0x246
3  popfd           # put a safe value in eflags
4  push    esi     # save scratch registers
5  push    edi
6  push    eax
7  mov     esi,esp # save stack pointer
8  lea     ecx,[edx*0x4+0x8054e60]
9  push    ecx     # pass effective address of
                   # instrumented instruction to
                   # instrumentation
10 mov     [ebx+0x50],edx   # spill virtual ebx
11 mov     [ebx+0x60],ecx   # spill virtual edx
12 mov     [ebx+0x1000],esi # spill virtual framep
13 mov     edi,[ebx+0x10]   # fill virtual edi
14 mov     esi,[ebx+0x20]   # fill virtual esi
15 mov     edx,[ebx+0x60]   # fill virtual edx
16 mov     ecx,[ebx+0x70]   # fill virtual ecx
17 call    RecordEffectiveAddress # call instrumentation
18 mov     edx,eax   # save return value
19 mov     ecx,[ebx+0x1000] # fill virtual framep
20 mov     esp,ecx   # restore stack pointer
21 pop     eax       # restore scratch registers
22 mov     [ebx+0x10],edi   # spill virtual edi
23 pop     edi
24 mov     [ebx+0x20],esi   # spill virtual esi
25 pop     esi
26 popfd             # restore eflags
27 mov     [ebx+0x70],edi   # spill virtual ecx
28 mov     [ebx+0xf60],edx  # spill virtual retval
29 mov     ecx,[ebx+0x20]   # fill virtual esi
30 mov     edx,[ebx+0x50]   # fill virtual ebx
31 ret
```

Figure 10. Gzip hottest basic blocks - with memory analysis tool

and then restore the stack pointer. Pin performs aggressive analysis to avoid spilling/filling the `eflags` and it appears that it is successful in keeping the spill overhead low.

Second, the majority of the overhead (around 90%) is incurred from register spills and reloads of general purpose registers. To help isolate the cause, our profiler produces additional information about the hottest traces generated by the JIT and their individual overhead. For the Gzip program with the memory analysis tool, Figure 3 in Section 3 shows that the two hottest basic blocks start at 0x04a400332 and 0x04a4002f4. They both execute 664 million times and contain 0 instructions from the original application.

The annotated assembly listing for these 2 blocks are in Figure 10. It is a wrapper for a call to instrumentation. In this case, the tool has requested that Pin insert a call to RecordEffectiveAddress, passing it the data address of an application instruction that references memory.

Our primary concern was the high spills and fills of registers. These are all annotated with spill or fill in the figure. When generating code, Pin treats the original instruction register operands as virtual registers, and can bind them to any physical register or spill them to memory [5]. At the time that the wrapper in Figure 10 is called, application register `ebx` is bound to `edx`, `edx` is in `esi`, `esi` is in `ecx`, and

`edi` is in memory. When RecordEffectiveAddress is called, we want application `edi` to be in physical `edi`, application `esi` in physical `esi`, etc. We call this an identity mapping. The instructions in lines 10-16 move registers to the proper place before the call to RecordEffectiveAddress. When we return from the wrapper, the bindings of the registers must be the same as when we entered, so we have to do more spills and fills at lines 27-30 to put everything back. This wrapper function or a similar wrapper is called for most instrumented memory instructions, which means that it is repeatedly called from the same trace. If we bias register allocation in the calling trace to keep the identity mapping of bindings (i.e. application `edx` in physical `edx`), then the calling trace becomes slightly longer, but much of the spills and fills in the wrapper are eliminated for a net performance win.

A similar problem exists for the `eflags` register handling. The wrapper contains 4 instructions (lines 1-3, 26) to save and restore `eflags`. By moving this from the wrapper to the trace that calls it, it is likely that it only needs to be done once per trace (or less) instead of every time the trace calls the wrapper. The same situation applies for the saving and restoring of the scratch registers (lines 4-6 and 21-25).

Our original motivation in the design was to make the application trace as small as possible and move the work to a wrapper where it can be shared. This approach favors reduced code size over performance. After studying the results of the profiler, we now believe that the extra instruction count this entails is not worth the reduced code size. Ideally, the hot code would have all of the work of the wrapper inlined for maximum performance, and infrequently executed code could rely on the work being done in the wrapper to reduce the size of the generated code for the trace.

It is interesting to note that the register allocation overhead for wrapper routines is much smaller when simple instrumentation is examined. See Figure 6 and Figure 7 for comparison. The memory analysis tool makes heavy use of Pin's inliner, which is much more likely to result in non-identity mapping for register bindings, triggering the high register allocator overhead in the wrapper routines. It was totally unexpected that inlining makes non-inlined calls to instrumentation slower. This demonstrates the point that it is not sufficient to look at simple instrumentation tasks like basic block profiling or memory trace collection when evaluating the performance of a system.

5.5.2 Tool

Figure 8 shows that 38% of the code cache instruction count for the memory analysis tool was contributed by the Pin tool. This is 9 times the instruction count of the original application. We divide the time further in Figure 11.

Most of the tool instruction count comes from the wrapper routine; an example is shown in Figure 10. Note that the *Wrapper* category in Figure 11 only accounts for the instructions that are not marked fill or spill (fills and spills are charged as a register allocation overhead). As we discussed in Section 5.5.1, some of the overhead of the wrapper can be amortized by moving it into the trace.

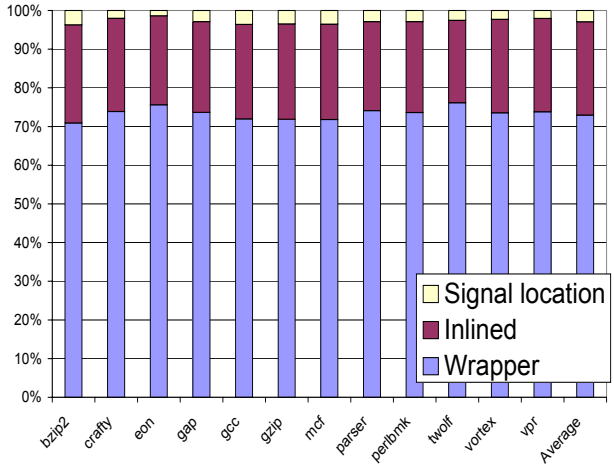

Figure 11. Breakdown of tool overhead with memory analysis instrumentation

The next category, *Inlined*, includes the instrumentation that is inserted inline into the trace of application instructions. It is either the code to call the wrapper or the entire body of the instrumentation if it is small enough for Pin to inline. The authors of the memory analysis tool carefully optimized it to take advantage of Pin's ability to inline. We believed that the time breakdown in Figure 1 indicated that they were successful. If the application was spending time in non-inlined instrumentation, then we would expect to see a significant amount of time in the *Instrumentation Tool* category. However, Figure 11 shows that it spends more time in the wrapper, which is used to call non-inline instrumentation. We believe that this is because the overhead of the wrapper is large, and it is used to call a non-inlined function in the tool with a relatively short path length. This indicates that more investigation is needed to enable more inlining so we may totally eliminate the wrapper overhead.

The final category is *Signal location*. Pin's emulation of Unix signals requires it to do some bookkeeping before every call to non-inlined instrumentation. The measurements confirm that this is not currently a significant overhead.

6. Summary and Conclusions

Dynamic binary instrumentation systems provide a powerful and robust method for analyzing the dynamic behavior of programs. The performance of instrumentation tools is critical to their success. However, it is very difficult to diagnose performance problems because of the interaction of original application, instrumentation and DBI code in a system where code is dynamically generated and optimized. We describe a self-instrumentation based method for obtaining fine grained profiles of instrumentation systems. Our system

makes it easier to identify the source of overhead and track it down to its root cause.

We use the profiler to identify performance opportunities in Pin. We investigate a variety of scenarios: no instrumentation, basic block instrumentation, and complex memory instruction instrumentation. We find that most of the overhead when no instrumentation is used is in indirect branch resolution. However, the overheads in a complex instrumentation tool are much higher and branch resolution does not have a significant impact on performance. As the instrumentation becomes more complex, the overhead introduced by register allocation becomes higher and appears to be comparable to the work for instrumentation. We are investigating a different strategy in the management of register state which we hope will make it possible to reduce much of the register allocation overhead. The results also indicate that there is some opportunity is making Pin's inliner more aggressive.

Finally, we conclude that optimizations for instrumentation can have complex interactions. To truly understand the performance of a general purpose instrumentation system it is necessary to look at usages that accurately reflect what real users do.

References

[1] S. Sridhar, J. Shapiro and P. Bungale HDTrans: An Open Source, Low-Level Dynamic Instrumentation System In *Proceedings of the Second International Conference on Virtual Execution Environments*, June, 2006.

[2] O. Agesen Binary Translation of Returns In *Proceedings of the Workshop on Binary Instrumentation and Applications*, October, 2006.

[3] Intel. IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M, March, 2006.

[4] A. Srivastava, A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196-205, 1994.

[5] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June, 2005.

[6] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verification* (http://valgrind.kde.org). 2003.

[7] D.L. Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation.
PhD thesis, M.I.T.(http://www.cag.lcs.mit.edu/dynamorio), September, 2004.

[8] J.L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. Manipulation. In *IEEE Computer*, 33(7):28-35, July, 2000.

[9] M. Poletto and V. Sarkar. Linear scan register allocation. In *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept, 1999.

[10] Intel. VTune^{TM} Analyzers. http://www.intel.com/cd/software /products/asmo-na/eng/vtune/index.htm

[11] X. Gao, M. Laurenzano, B. Simon, and A. Snavely Reducing Overheads for Acquiring Dynamic Traces In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC05)*, Oct. 2005, Austin

[12] A. Jaleel, M. Mattina, and B. Jacob Last-level cache (LLC) performance of data-mining workloads on a CMP–A case study of parallel bioinformatics workloads In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA 2006)*, pp. 88-98. Austin TX, February 2006.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe Secure Execution Via Program Shepherding In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, August 5, 2002.

[14] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*, December, 2004