

Scalable Support for Multithreaded Applications on Dynamic Binary Instrumentation Systems

Kim Hazelwood^{†,‡} Greg Lueck[‡] Robert Cohn[‡]

[†]University of Virginia [‡]Intel Corporation
www.pintool.org

Abstract

Dynamic binary instrumentation systems are used to inject or modify arbitrary instructions in existing binary applications; several such systems have been developed over the past decade. Much of the literature describing the internal architecture and performance of these systems has focused on executing single-threaded guest applications. In this paper, we discuss the specific design decisions necessary for supporting large, multithreaded applications on JIT-based dynamic instrumentation systems. While implementing a working solution for multithreading is straightforward, providing a system that scales in terms of memory and performance is much more intricate. We highlight the design decisions in the latest version of the Pin dynamic instrumentation system, including the just-in-time compiler, the emulator, and the code cache. The overall design strives to provide scalable performance and memory footprints on modern applications.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Code generation, Optimization, Run-time environments

General Terms Languages, Management, Measurement, Performance

Keywords scalability, multithreading, memory management, instrumentation

1. Introduction

The recent trend toward multicore architectures has led software developers to focus on ways to leverage multiple processing cores in their application software. One way to utilize multiple cores is to develop multithreaded (MT) applications. Despite the fact that MT programs are ubiquitous, many system designers still evaluate their systems with small, single-threaded (ST) applications. There are many factors contributing to the lack of analysis of systems with MT workloads. Simulation and analysis tools either emphasize or exclusively support single-threaded applications or are too slow to execute large MT programs. MT applications are inherently less deterministic than ST applications, complicating the evaluation methodology (Pereira et al. 2008). The disconnect between today's architectures and the applications supported by today's tools is particularly problematic as we move further away from single-core

machines, and the results acquired from single-threaded applications become even more irrelevant.

To remedy this disconnect, many simulation and analysis tools are adding support for multithreaded applications. Developers of these tools are learning that providing support for multithreaded applications is often straightforward, but providing robust and/or scalable support for multithreading tends to be much more of a challenge (Jaleel et al. 2008). Given recent trends toward dramatically increasing the number of cores in multicore processors, it is ever more critical for the development of scalable solutions to this and many other design challenges.

At the same time, dynamic binary instrumentation has emerged as an invaluable mechanism for analyzing and modifying software, and even simulating new and existing hardware. Unlike static instrumentation systems, dynamic instrumentation systems enable analysis of all executed instructions including shared libraries, dynamically generated code, and perhaps most importantly, applications for which source code is not available. One such dynamic binary instrumentation system that is widely used due to its user-friendly API and robust implementation is the Pin dynamic instrumentation system.

In this paper, we present and analyze various aspects of our design for robust support for multithreaded guest application execution on the Pin dynamic instrumentation system (Luk et al. 2005). After providing an overview of Pin in Section 2, we introduce the basic modifications necessary for supporting multithreaded applications in Section 3. Next, we delve into our approaches for supporting signals in Section 4. Section 5 focuses on the code cache and presents our trace construction policy that balances memory and performance overheads, and our generational cache flushing policy that allows us to avoid synchronizing cache flushes across all threads. Section 6 then evaluates the resulting memory and performance scalability of the system. Finally, Section 7 presents related work and Section 8 concludes.

2. Pin's High-Level Architecture

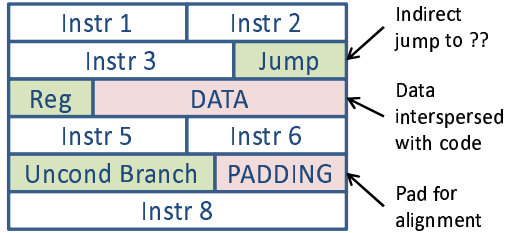
Before delving into the design aspects that target Pin's support for multithreading, we first provide a high-level view of Pin's internal architecture. We highlight the features that are discussed in more detail when we focus multithreading support.

At a very high level, Pin is a tool that allows users to modify existing binary applications with an easy-to-use, cross-platform instrumentation API. The user simply writes a short, plug-in C++ program (called a *Pintool*) that defines *where* the new code should be inserted, *what* code to insert, and *when* to notify the user of various events such as thread creation (i.e. callbacks). The rest is handled automatically and transparently by Pin, which operates on IA32, Intel[®] 64, Intel[®] Itanium, and ARM. Pin operates at runtime, since it is impossible to find and modify all of the instruc-

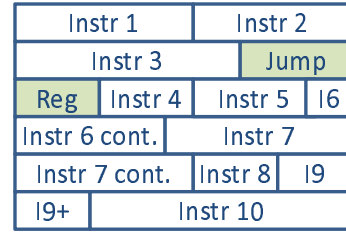
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to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

ISMM'09 June 19-20, 2009, Dublin, Ireland.

Copyright © 2009 ACM 978-1-60558-347-1/09/06...\$5.00



(a) Dynamic x86 Decoding Example



(b) Static x86 Decoding Example

Figure 1. The code-discovery problem that arises from variable instruction lengths, mixed code and data, and indirect branches. On the left, we show the correct interpretation of the binary as determined by a dynamic instrumentation system. On the right, we show an incorrect interpretation as seen by a static instrumentation system (Smith and Nair 2005).

tions in a program before execution. (The *code-discovery problem* prevents robust static modification of IA32 binaries, as data may be interspersed between variable-length application instructions, complicating decoding (Harris and Miller 2005). See Figure 1 for a demonstration.) Pin and the Pintool are invoked along with the application at the command line. Pin launches and injects itself into an application then uses the Pintool to dictate how to modify the application.

Pin provides support for two general techniques for runtime modification of binary applications – just-in-time (JIT) compilation and probing. In a *JIT-based approach*, a JIT compiler modifies and recompiles small chunks of binary instructions immediately prior to executing them. Modified instructions are stored in a software code cache where they are executed in lieu of the original application instructions. Software code caches allow the code regions to be modified once and then reused for the remainder of program execution, helping to amortize the costs of compilation.

In a *probe-based approach*, the original binary is modified in place (Hunt and Brubacher 1999; Buck and Hollingsworth 2000). The system overwrites original application instructions with branches to instrumentation. Pin’s probe-based approach has near zero execution and memory overhead, but has a more restrictive API, mainly limiting tools to interposing wrapper routines for global functions while the JIT-based approach allows fine-grained instrumentation down to the instruction level.

Most tools use the JIT-based instrumentation, since it comes with fewer limitations regarding how and where you can instrument the code. Therefore, we focus exclusively on the JIT-based implementation in this paper.

The architecture of the JIT-based version of Pin is shown in Figure 2. Note that all threads in the application run in the same Pin instance and some portions of Pin and the Pintool run serially while others run in parallel across all threads. In general, we serialize the infrequent code paths (such as the Pin JIT compiler) for simplicity of design, while allowing the frequent paths (such as executing application code) to run in parallel.

3. Basic Multithreading Support

To support multithreading, we need a solution that is scalable in a number of dimensions. Clearly, applications that benefit from parallelism must achieve similar speedups when run under Pin. Although efficient parallel programs usually limit the number of threads to the number of physical cores, it is also important that Pin perform well when applications create a large number of concurrent threads or repeatedly create and destroy threads. We describe the low-level Linux kernel interface for threads and explain how Pin manages threads.

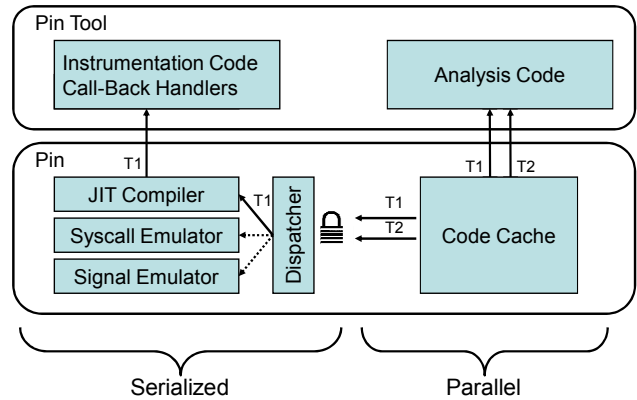


Figure 2. The architecture of Pin’s JIT-based instrumentation system. Pin runs in a user space process, where all threads share a single instance of Pin. Infrequent code paths are serialized to protect key data structures, but frequent code paths are allowed to run in parallel.

Programs are invoked by a `fork` system call, which creates a process, followed by an `exec` system call, which loads a program and starts execution. A process always starts with a single thread of control. When a program executes a `clone` system call, the kernel creates another thread (called a kernel thread) and it appears as though both the original thread and the newly created thread return from the `clone` system call. Threads typically end by executing an `exit` system call, which can kill one or all threads.

Pin intercepts all system calls. When it observes an application `clone` system call, it does the clone on behalf of the application, and allows the parent thread to resume execution in the code cache. Pin increments the number of threads and allocates a thread-local data structure for the new thread. Pin uses a callback to notify the Pintool when new threads are created. Finally, the new thread resumes executing in the code cache.

Pin also intercepts `exit` system calls. It notifies the Pintool that a thread is exiting and adds the system thread ID to the *pending dead* list. If this is the only thread or the exit requests to kill all threads, then Pin also notifies the Pintool that the program is exiting. Finally, it executes the `exit` system call on behalf of the thread, destroying the kernel thread. To simplify the bookkeeping, we do not immediately release any of the Pin-allocated data structures until after the kernel has eliminated the thread. When a new thread is created, we query the OS for each thread ID in the *pending dead* list, and reclaim the memory.

	Static	Dynamic
Direct	74 %	86 %
Indirect	4 %	3 %
Call	15 %	6 %
Return	7 %	6 %
Syscall	< 1%	< 1%

Figure 3. The breakdown in static and dynamic branch types, across the SPEC OMP 2001 benchmarks. The indirect branches and returns must be handled specially in order to scale well.

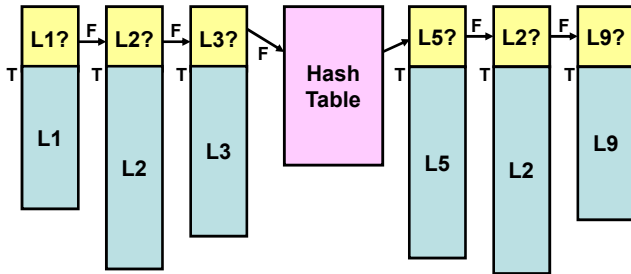


Figure 4. Indirect branch resolution. Each trace contains a check that determines whether to fall into the trace or to continue down the chain of predictions.

To ensure that multithreaded programs continue to scale when run under Pin, we avoid mechanisms that insert extra synchronization in the steady state of the application. For the most part, the thread only needs to access the Pin thread-local storage and no extra synchronization is required. However, some resources are shared among threads and access must be coordinated. We took extra care to allow frequently accessed code or data to be updated by one thread without stopping other threads.

3.1 Concurrent Instruction Rewrite

Pin’s just-in-time compiler stores instructions in the code cache, which is shared by all threads. JIT compilation is typically triggered by a branch to a previously untranslated part of the application. The branch goes to a stub, which invokes the compiler. After the new code is generated, Pin overwrites the branch to go directly to the new code. When translated code is invalidated, Pin must rewrite branches that target invalidated code to point back to a stub. The branch patching process must occur concurrently with other threads executing the same instructions, which can be problematic.

To solve this challenge, Pin patches the first two bytes of a branch with a temporary *self branch* instruction. Next, it patches the remainder of the bytes of the branch with the latter (n-2) bytes of the new instruction. Finally, Pin overwrites the first two bytes of the branch with the correct bytes for the new instruction. The three step process ensures that all threads either (a) see the old branch instruction, (b) spin on a self branch, or (c) see the new instruction (Sundaresan et al. 2006). However, Intel64 instructions are variable length and can start on any alignment boundary, which places restrictions on the method for rewriting. First, the 2-byte self branch must be written atomically. Intel64 memory ordering requires that the self branch begin on an even address (Intel 2007). Pin inserts a padding nop if it is required to align the beginning of the original branch instruction. Second, other processes must see the stores for all the updates in program order. The Intel64 memory ordering manual states, “Loads are not reordered with other loads and stores are not reordered with other stores”, which guarantees this property for data writes and reads. However, this rule does not hold in our case because the read is an instruction fetch. To

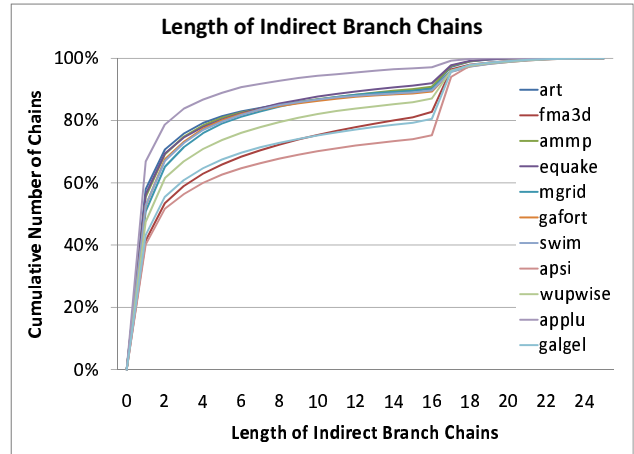


Figure 5. The cumulative length of indirect branch prediction chains for the SPEC OMP 2001 benchmarks. The max chain length is set at 16, while most of the chains are fairly short.

ensure that the stores are seen in program order for instruction fetch, Pin does not allow a patchable branch to cross a cache line boundary. This is accomplished by inserting padding nops before the instruction.

3.2 Concurrent Indirect Table Update

Since Pin executes a cached, modified copy of application instructions, it must map all of the original instructions to their cached copies at run time. For efficiency, it performs this mapping on a trace granularity since a hash table entry for every single instruction does not scale (and is unnecessary). Whenever a branch is encountered, Pin simply looks up the cached address of the branch target. It performs this lookup once per static instance of direct branch, which scales nicely since the lookup is expensive – it causes a context switch from the application state back into the translator to access the hash table without modifying the application’s register or memory state.

Indirect branches and return instructions can have a different destination each time they execute, so the code cache address of the branch target must be calculated every time (Agesen 2006; Sridhar et al. 2006; Hiser et al. 2007). A simple solution is to pay the performance penalty of context switching back to the translator on every indirect branch or return. However, a large percentage of branches are either indirect or returns (see Figure 3). Instead, Pin takes a multi-tiered approach depending on the number of indirect branch targets it encounters as shown in Figure 4. If an indirect branch usually targets a small set of targets, Pin generates code that compares the current target against the predicted targets (Luk et al. 2005). This allows the comparison code to be inlined into the application code, saving the overhead of a context switch. Figure 4 illustrates the sequence of comparisons of an indirect branch target against labels L1, L2, and L3. If any of these match, the comparison falls through to the code for L1, L2, and L3 respectively. The lengths of the chains built while executing SPEC OMP 2001 are shown in Figure 5. Procedure returns can have 1,000’s of targets, and it would be impractical and slow to generate a sequence of checks for each one. Therefore, Pin falls back to using a hash table to map the target to the code cache, as shown in Figure 4 when the comparison against L3 fails. The hash table is indexed by the original branch target address, and it returns a corresponding code cache address. Collisions in the hash table are resolved by using

another chain of comparisons against a predicted target, as shown in the right side of Figure 4.

Aside from the performance and memory optimizations above, new design challenges arise when dealing with multithreaded applications. The targets of an indirect branch are discovered incrementally throughout the entire execution of a program. The *chain* of indirect branch predictions is built incrementally and the hash table is updated while the program is running. Indirect branches are frequent and performance critical; requiring threads to get a lock before each branch would be inefficient. Adding new links into the chain uses the branch patching mechanism described in Section 3.1, making it safe to add links while other threads are walking the chain. To allow the hash table to be updated concurrent with other threads reading it, Pin only stores a single code cache address per source address. A thread unconditionally jumps to the code cache address without checking whether it is a match for the current target address. Since it jumps to code that checks whether the target is correct; it does not matter if another thread is updating the table to point to a different target while another thread is reading it.

3.3 Serial Translation

Pin performs just-in-time translation of a few basic blocks of application code at a time. When a thread exits a translation block and the target address has not already been translated, the thread enters the JIT, which generates a few more basic blocks of application code, and execution resumes in the newly generated code. The JIT is guarded by a single lock, so translation can become a serialization point. However, once programs reach the steady state very little time is spent in translation, typically less than 5%. For most programs, concurrent translation would provide a small benefit and it did not merit the complexity required to support it.

3.4 Serial System Calls

As described earlier, Pin must intercept certain system calls. These include system calls for creating and destroying threads, allocating memory, and manipulating signals. During system call interception, the thread enters the Pin translator which allows Pin to perform the system call on behalf of the application. Depending on the system call, this execution path may need to touch some global data. For simplicity, we protect it with a single lock. As a consequence, there is a serialization point in the path that executes system calls. The system call itself is executed without holding a lock. The Linux kernel executes some system calls without any serialization, so Pin is potentially worse. We did not expect concurrent execution of system calls to be beneficial for the applications we are targeting, but are investigating the benefits of eliminating the lock in the future.

3.5 Thread-Local Storage

Pin inserts instrumentation code into existing binary applications without cooperation from a static compiler. Therefore, Pin may need to generate code to spill application register values in order to free up registers for use by the instrumentation code, while preserving the original application computation. Compilers typically spill values on the stack, but Pin cannot write values to the application's stack without potentially changing the behavior of the program. Similarly, calls to instrumentation in the user's Pintool require values to be read and written to the stack, but using the application stack could change the behavior of the program.

Instead, Pin steals a general-purpose register to point to thread-local storage (TLS) and rewrites all application uses of the register. All spills and fills are relative to the pointer. Before calling instrumentation, Pin sets the stack pointer register to a location in the TLS. Management of TLS is straightforward. When the application

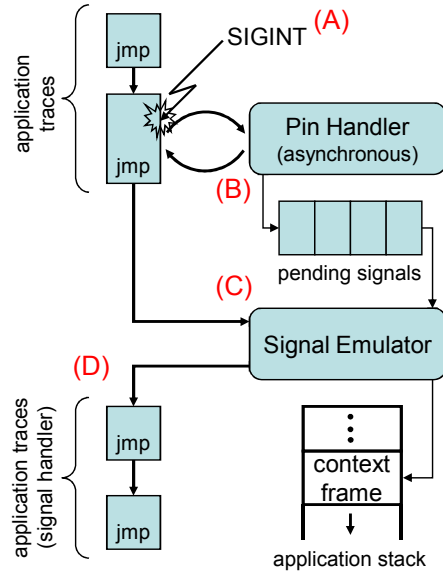


Figure 6. When Pin receives an asynchronous signal, it places the signal on a pending queue and waits for the end of the current trace. When the trace completes, Pin emulates a signal context frame on the application's stack and starts compiling the application's signal handler.

begins execution, the TLS register points to an initial TLS. Then, as threads are created or destroyed, TLS areas are allocated and freed.

Other systems have used immediate addressing in conjunction with thread-private caches (Bruening et al. 2003), but this solution does not scale when there are a large number of threads (Bruening et al. 2006). We also considered using IA-32 segmentation by stealing a segment register instead of a general-purpose register to address TLS. We opted to use a general-purpose register to allow the same method to be used across multiple architectures and operating systems. Experiments show that stealing a general-purpose register costs 10% on IA-32 and 5% on Intel64 compared to immediate addressing for single-threaded programs.

3.6 Writing Thread-Safe Instrumentation Tools

Calls to instrumentation are executed by the application threads. The Pintool author is responsible for making the instrumentation thread-safe so it can be applied to a multithreaded program. Writing a thread-safe instrumentation routine is similar to writing a traditional thread-safe routine in a multithreaded program. References to shared data must be synchronized with other threads using locks. Pin also provides APIs for allocating and addressing thread-local storage.

It is usually straightforward to make a parallel program correct by adding locks. However, a highly contended lock serializes execution and leads to poor CPU utilization. Because application threads execute instrumentation, a highly contended lock in a Pintool serializes the execution of the application. The serialization increases the overhead of the Pintool when compared to uninstrumented execution of the application, and may drastically alter the behavior of a parallel program.

Pintool authors must employ standard parallel programming techniques for avoiding excessive serialization. They should use thread-local storage to avoid the need to lock global storage. Instead of a single monolithic lock for a data structure, they should use fine-grained locks. Pintool data structures must be carefully laid out to avoid false sharing.

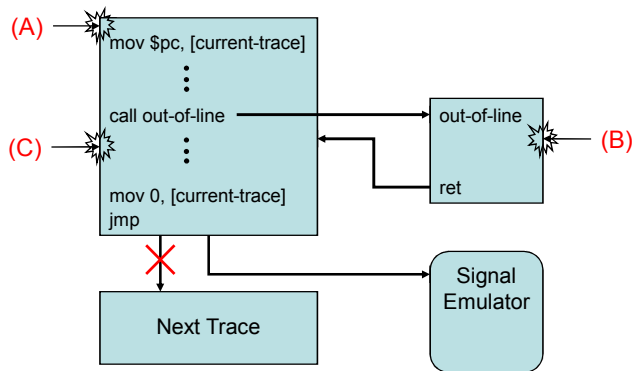


Figure 7. Pin uses *current trace* markers to help identify the current trace when a signal arrives asynchronously.

4. Signal Support

Unix signals present a challenge to Pin. One might think that Pin can simply let the application handle signals natively, but this presents three problems. First, the application’s signal handler escapes from Pin’s control and is executed without the JIT compiler. Second, if the application’s handler looks at the interrupted register context, it sees the interrupted Pin state, not the pure application state. Finally, the signal might interrupt the Pintool’s instrumentation and cause reentrancy problems.

For these reasons, Pin intercepts the application’s signal handlers and emulates signal delivery. Figure 6 illustrates that when a signal arrives (A), Pin catches the signal and places it on a queue of pending signals (B). Execution then resumes until any tool instrumentation completes and we reach the end of the next code cache trace. At the end of the trace, control jumps to the Pin Signal Emulator (C), which builds an emulated signal context frame using the values of the application’s registers. Finally, the Signal Emulator starts translating code at the PC of the application’s handler (D).

One critical feature of the algorithm is ensuring that the Pin VM notices that signals are pending at the end of the next trace. This is important for timely delivery of signals, but it is also a correctness issue since signals cannot be delayed across certain system calls. One option is to insert checking code at the end of each trace to look for pending signals. However, this adds overhead in the common case when there are no pending signals. Instead, we patch – or *unlink* – the outbound branch of the currently-executing trace to return control to the VM. The VM checks for pending signals as part of its normal processing.

4.1 Identifying the Currently-Executing Trace

The protocol for identifying the trace to unlink has some interesting performance considerations. In our initial implementation, we simply unlinked all traces whenever Pin received a signal. This worked correctly, but scaled very poorly for large applications, where there are tens of thousands of traces or more. Instead, we want to unlink just a single trace. If Pin receives a signal while executing in a trace, the solution is simple. Pin can use the PC from the interrupted context to find the trace to unlink.

However, there are other situations when signals may arrive, such as during the execution of an out-of-line analysis routine (which may be called from a number of different cached traces). Conceptually, we need to find the trace that ultimately called this out-of-line routine. Walking the stack is unreliable, especially in highly-optimized code. Instead, Pin inserts *current trace* markers in traces that have out-of-line calls, as shown in Figure 7. The marker simply writes the PC of the trace to a thread-private location, so

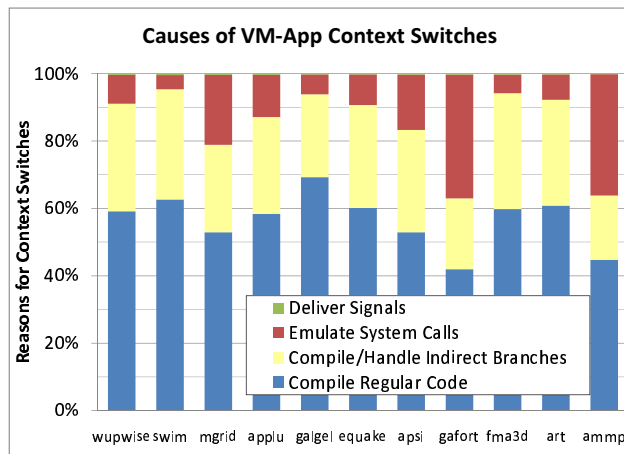


Figure 8. A breakdown of run-time reasons for entering the VM for the SPEC OMP 2001 benchmarks. Each entry and exit requires a context switch.

the marker code is very small and fast. Furthermore, the marker code is only needed for traces with out-of-line calls, so the relative overhead can be negligible. Figure 7 shows three possible points when a signal might interrupt a trace. If a signal arrives at (A), we cannot use the *current trace* marker because it has not yet been set up. Instead, we use the interrupted PC to find the current trace. If a signal arrives at (B), the PC is in an out-of-line analysis call, so it does not tell us the trace to unlink. Instead, we use the *current trace* marker to find this trace. Finally, if a signal arrives at (C), we could use either method to find the trace.

4.2 Optimizing Entry and Exit to the VM

Pin enters the VM to perform three basic services: 1) to compile new code, 2) to deliver pending signals, 3) to emulate system calls. The breakdown of the reasons for entering the VM is shown in Figure 8. The time to actually enter and leave the VM is not critical for the case of compiling code, because the compiler itself is relatively slow and the frequency of new code decreases as the application reaches steady state. The frequency of signal delivery and system call emulation, though, remains steady even after the application’s startup phase. Moreover, the cost of delivering a signal or emulating a system call is low, so the overhead of entering and leaving the VM can be a significant fraction.

One fruitful optimization involves the process for leaving the VM. Pin needs to generate a single-use code fragment each time it leaves the VM, which helps restore the register state when switching from the VM to the application context. Previously, we used our JIT compiler to generate these instructions on each exit, and this dominated the time needed to emulate a signal’s delivery. We optimized this solution by using a pre-compiled code template for exit code fragment. Each trip through the VM now just fills in a few fields in this template instead of compiling it from scratch, allowing our solution to better scale to a large number of signals and system calls.

5. The Software Code Cache

Most dynamic binary instrumentation systems employ a software code cache to store previous translated code and amortize the overhead of compilation. The code cache consists of a directory that contains a mapping from original to translated instructions, a set of data structures that keep track of any patched branches in the code cache, and a code area that contains instrumented, translated

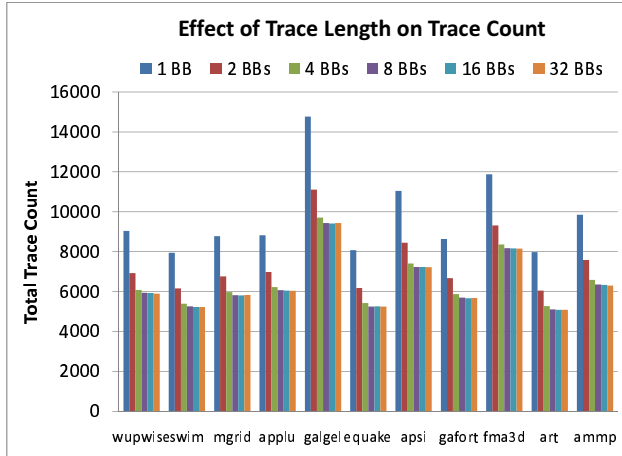


Figure 9. The impact of limiting cached traces to a given number of basic blocks for SPEC OMP 2001. As we move from shorter to longer traces, the number of generated traces declines.

code as well as auxiliary code to maintain control of execution. The maintenance of the various code cache structures is critical for both correctness and performance (Hazelwood and Smith 2006).

5.1 Scalable Trace Generation

The traces that Pin stores in its code cache consist of a sequence of contiguous basic blocks. Traces are formed when the first basic block executes, therefore all subsequent blocks are speculative. A memory-performance trade-off exists when considering the number of basic blocks to include in a trace. The speculative nature of traces means that various basic blocks that never execute may be compiled and included in a trace. However, when speculation is correct, there are fewer context switches into the VM and the compilation time is well spent.

All of the off-trace paths in the code cache require *exit stubs* to maintain control and return execution to the VM for on-the-fly compilation of the target. Interestingly, when we limit traces to a single basic block, we end up with many more exit stubs (and hence a higher memory footprint) than when we allow multi-block traces. This is because a single basic block may end in a conditional branch, which has two potential targets and thus requires two exit stubs. However, forming a contiguous trace would only require one exit stub since the fall through path would remain on trace, obviating the need for us to catch the otherwise off-trace path (Guha et al. 2007).

Figures 9 and 10 demonstrate the impact of the trace length on the number of traces generated and the overall memory consumption. We see that while the trace count decreases when we allow longer traces, the memory consumption increases for a number of reasons. The predominant cause is *tail duplication* - a given basic block may appear in a large number of different traces. Meanwhile, the overall runtime performance (not shown) remains fairly stable, with the extreme trace limit values having higher overhead than a limit of 2-16 basic blocks. Combining each of these factors, we determined that the most scalable trace limit from both a memory footprint and performance standpoint is approximately three basic blocks, which is the default for Pin.

5.2 Bounding Code Caches

Many dynamic binary modifiers, such as DynamoRIO and Strata, allow their software code caches to grow without bound, as their design highlights performance at the expense of memory footprint.

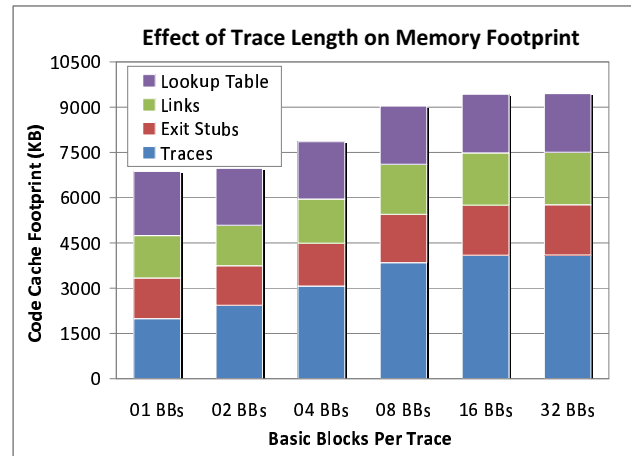


Figure 10. The memory impact of limiting cached traces (for the galgel benchmark). As we move from shorter to longer traces, the overall memory footprint increases.

However, when scaling to large applications, we found that unbounded code caches are beneficial neither from the performance nor the memory footprint perspective. In fact, even the simple, single-threaded application *perlbench* from SPEC2006 exhibits a speedup when the cache is bounded, as shown in Figure 11. In the case of *perlbench*, the code cache would have grown to about 3.5 MB (depending on the input) had we allowed it to grow without bound. Yet this is the worst-performing configuration of those we tried. We saw similar trends on larger enterprise applications that exhibit little code reuse. Intuitively, those applications have no need to maintain the translated code sequences that are a part of an initialization sequence, and meanwhile, those sequences take up valuable memory and hardware cache resources. Therefore, Pin's code cache is bounded by default, and we are investigating adaptive schemes that allow us to recognize and remove only the stale code when approaching the size limit of the code cache.

5.3 Generational Code Cache Eviction

Aside from bounding the size of the code cache, there are numerous reasons that Pin must remove translated code from the code cache. Dynamically unloaded code will need to be invalidated as other application code may be loaded at the same address. Users may request invalidation of cached code to remove instrumentation or to re-optimize regions of code. Finally, self-modifying code must also trigger invalidation and regeneration of the corresponding cached code.

One fundamental observation is that code cache maintenance becomes much more complex when executing a multithreaded application. For instance, a complete flush of the the code cache is no longer possible. The problem is each thread is in one of two states - executing code in the code cache, or generating new code in the VM. In fact, many threads will be stalled in one of these two states. Pin may not delete a trace that another thread is currently executing or stalled in, but it is also extremely difficult to determine where each thread is executing without employing very expensive techniques like having every thread log every trace it enters (unlike the situation with signals, where we can use the thread context and we rarely need to augment traces with a current trace marker). While an easy alternative is to stall the flush until all threads are executing VM code (and thus not executing cached code), this stall can take an indefinite amount of time, and meanwhile threads are not able to make forward progress.

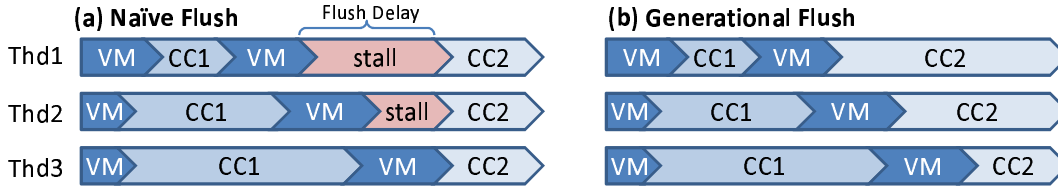


Figure 12. Timeline depicting a naïve code cache flush and our generational flush. The naïve implementation stalls until all threads return to the VM, while the generational implementation makes forward progress as it waits for threads to return to the VM.

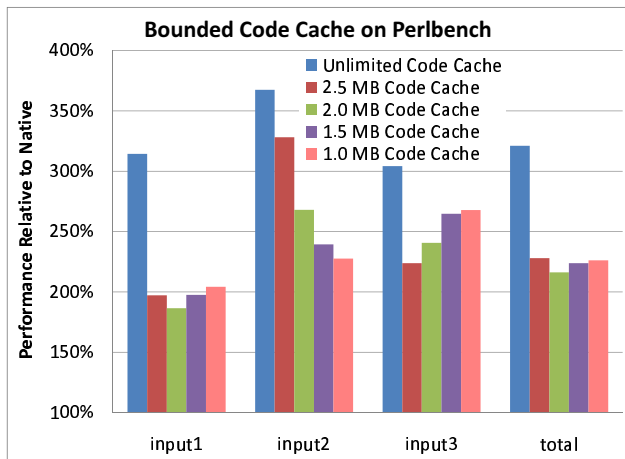


Figure 11. The performance effect of limiting the code cache size for perlbench from SPECint2006 (relative to native). We see that bounding the code cache improves overall wall-clock performance.

Our solution is to employ a generational code cache deletion algorithm, demonstrated in Figure 12. The code cache is partitioned into multiple blocks, and each block is marked with a generation number. Initially, all threads execute in generation 1. When a code cache flush needs to occur, we simply clear the cache lookup table (so that no threads enter the cache blocks for the current generation), advance the generation counter, and create a new cache block tagged with the new cache generation. As each thread enters the VM, it will only find the new traces present in the new generation, since the old traces will have been removed from the look-up table, and thus the thread will never re-enter code from the old generation. Threads may also move forward with generating new code for the new generation. Finally, when the last thread leaves the old generation, we flush the cache blocks for that generation. This scheme allows threads to continue to generate and execute new code while other threads are potentially stalled and/or in the process of leaving the old generation.

5.4 Thread-Shared Code Caches

It is fairly straightforward to implement a code cache in such a way that each thread has its own code cache. This approach avoids many of the synchronization and deletion headaches that arise when multiple threads share a cache. For instance, the task of deleting a single trace from the code becomes fairly complicated to do efficiently, since the system must ensure that no other threads are executing or stalled within the candidate trace. In addition, patching a branch within a trace so that it will jump directly to another trace must be done in an atomic manner, such that other threads do not inadvertently execute partially-modified code.

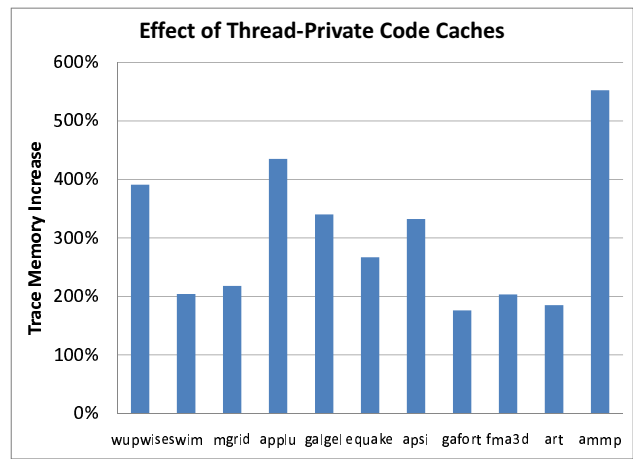


Figure 13. Code expansion resulting from thread-private code caches. The SPEC OMP 2001 benchmarks were run as 8 application threads.

Despite the complexity of thread-shared code caches, researchers have demonstrated (Bruening et al. 2006) that thread-private code caches do not scale beyond a very small number of threads. Therefore, it is worth the development effort to employ a single, shared code cache across all threads. For the applications where several threads perform similar tasks, the shared cache contains all of the common code, and thus the memory footprint is smaller, and more scalable with the number of threads. Figure 13 demonstrates the code expansion that occurs when the SPEC OMP 2001 benchmarks are executed using thread-private code caches and eight application threads. The figure shows that the code can expand well over 500% if we allow each thread to have its own private cache. This memory scalability issue justifies the use of a shared code cache across threads.

6. Overall Memory and Performance Scalability

Now that we have presented some of the fundamental issues that must be handled in order to provide robust support for multithreaded applications, we will discuss the memory and performance scalability of the SPEC OMP 2001 multithreaded benchmarks executing under the control of Pin. We varied the number of threads from 1 to 8 by using the OMP_NUM_THREADS environment variable. The complete reference inputs (medium) were executed to completion in our evaluation. We used Pin 2.6 for Intel64, release 24110, on an 8-core Xeon 1.6 GHz with 8 GB RAM running CentOS 4.7. All performance results are wall-clock time measurements reported by the official runspec benchmark launching script. Memory footprints were gathered using the mem_usage Pin-tool that comes with the 24110 Pin distribution. Figure 14 shows the speedup for SPEC OMP run without Pin.

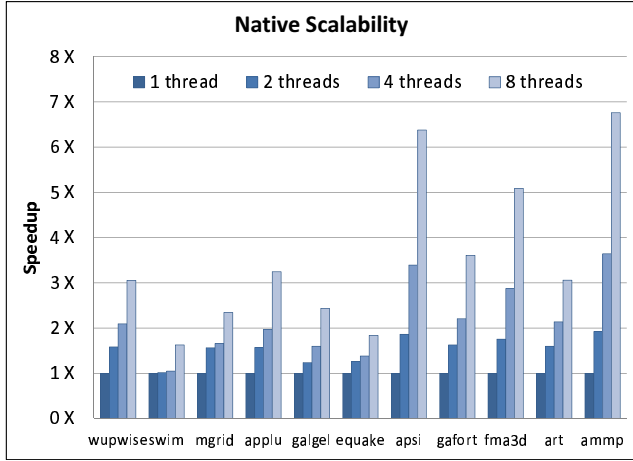


Figure 14. Speedup resulting from increasing the native thread count for SPEC OMP 2001 (run without Pin).

We evaluated the memory and performance scalability of two scenarios: (1) Pin executing a multithreaded application without user-inserted instrumentation, and (2) Pin executing a multithreaded application with frequent instrumentation calls. The former case demonstrates the fundamental scalability of the Pin engine without confusing the Pin engine overhead with the user-inserted instrumentation overhead. The latter case demonstrates the scalability of the instrumentation handling routines.

6.1 Translating MT Applications without Instrumentation

As Figure 14 shows, the performance of SPEC OMP steadily improves as we increase the number of threads until the thread count matches the number of processor cores. Instrumentation adds overhead, but we hope to see a similar speedup curve when executing applications under the control of Pin. We also want to verify that the memory footprint of Pin only increases by a small amount as the thread count increases.

Figures 15 and 16 show the runtime and memory scalability we observed when running the SPEC OMP benchmarks under the control of Pin without adding any instrumentation. In Figure 15, we normalize the time to execute under Pin for 1, 2, 4, and 8 threads to the time for native execution for 1, 2, 4, and 8 threads, respectively. Execution time under Pin scales similar to native execution, indicating that Pin does not serialize an otherwise parallel program. As demonstrated in Figure 16, the memory consumed by Pin’s code cache only increases slightly as the number of threads increases.

6.2 Translating MT Applications with Instrumentation

We also confirmed that the performance and memory scalability of Pin remains consistent when we move to more realistic instrumentation scenarios. We evaluated the memory and performance scalability while executing Pin with three plug-in instrumentation tools: `inscount2_mt`, `MemTrace`, and `CMP$im`.

The `inscount2_mt` Pintool is shipped as part of the standard Pin distribution, and counts the number of instructions executed by each thread in an MT application. It instruments every basic block with code to increment a thread-private counter. Pin inlines and optimizes the instrumentation code into the cached, modified copy of the application instructions. While counting instructions is not in itself interesting, it allows us to measure the overhead of more complex profiling tools without including work not directly related to instrumentation.

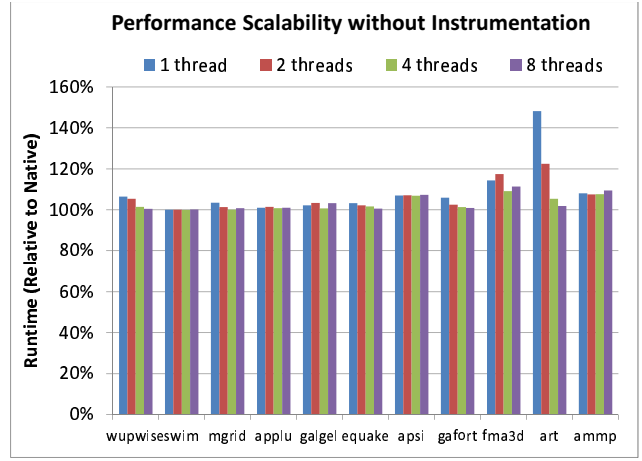


Figure 15. The runtime scalability of Pin without instrumentation on SPEC OMP 2001.

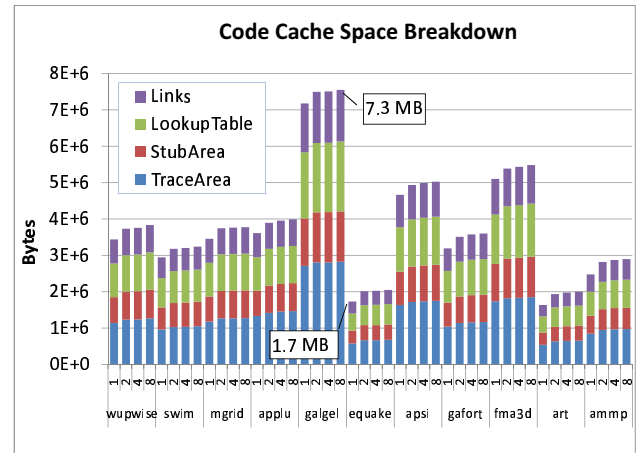
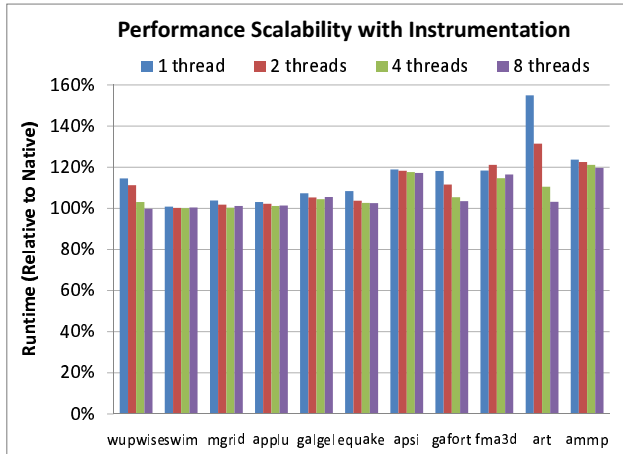


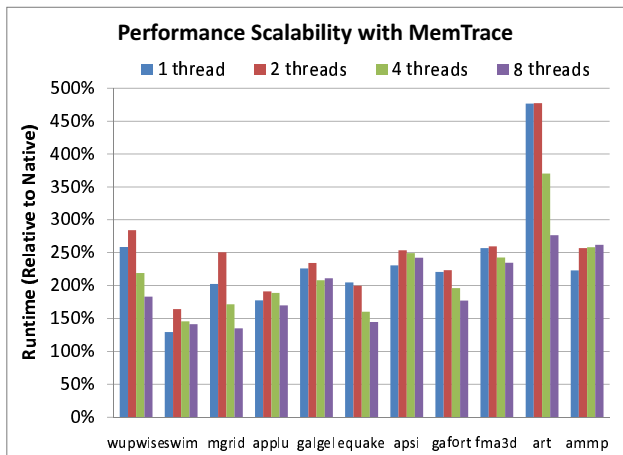
Figure 16. The memory scalability of Pin without instrumentation on SPEC OMP 2001. The subgroups on the horizontal axis are thread counts.

The `MemTrace` Pintool records each memory address accessed by a thread in a thread-private buffer. The source for this tool can be found in the Pin distribution. Cache simulators, memory corruption detectors, and data race detectors need to observe all memory addresses referenced by an application and this tool serves as a lower bound on their overhead. It has higher overhead than `inscount2_mt` because it must instrument every memory instruction instead of just once per basic block.

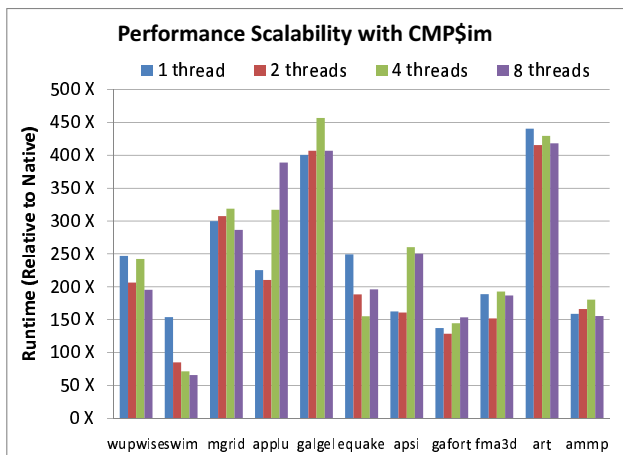
Finally, the `CMP$im` (Jaleel et al. 2008) Pintool uses Pin to collect the memory addresses of multithreaded and multiprocess programs and then uses a software model of the memory system to analyze program behavior. The memory system model uses multiple threads to simulate effect of memory references. `CMP$im` reports miss rates, cache line reuse, the sharing of cache lines, and coherence traffic. `CMP$im`’s memory system model configuration is very versatile and can be used to predict the performance of applications on future systems. While `CMP$im` is not yet publicly available, the Pin distribution includes the source for a simple cache model, `dcache.cpp`.



(a) The Light-Weight InsCount Pintool



(b) The Medium-Weight MemTrace Pintool



(c) The Heavy-Weight CMP\$im Pintool

Figure 17. The runtime scalability of various Pintools on SPEC OMP 2001. The Pintools range from a simple instruction counting tool to a complex cache simulator. Results are relative to native.

Figures 17 and 18 show how the performance and memory scales when Pin instruments the SPEC OMP 2001 benchmarks us-

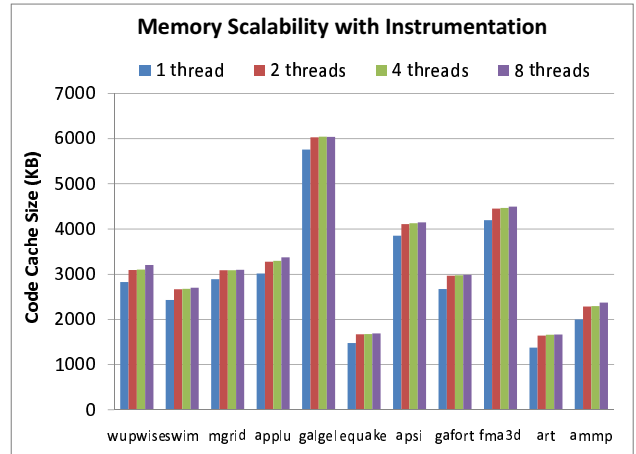


Figure 18. The memory scalability of the inscount Pintool on SPEC OMP 2001. Other Pintools exhibited similar trends.

ing the three plug-in Pintools. Again, thread counts are varied from 1 to 8 threads. As the figures indicate, the performance continues to follow similar trends to the native execution (stable or decreasing bars means that overhead does not increase as threads increase). Meanwhile, memory remains fairly stable, with a very small increase in memory footprint as additional threads are introduced. These trends hold regardless of the complexity of the Pintool.

7. Related Work

Several process-level dynamic binary instrumentation systems have been developed over the past decade. Dynamo (Bala et al. 2000) was developed at Hewlett-Packard Labs to provide dynamic optimization capabilities to PA-RISC applications. DynamoRIO (Bruening et al. 2003) was then ported to IA-32 with support for Windows and Linux applications, again with the motivation of providing dynamic optimization capabilities. Valgrind (Nethercote and Seward 2007) is a heavyweight open-source instrumentation system that allows users to track applications in very fine detail. Strata (Scott et al. 2003) is a dynamic modification engine that targets IA-32 and SPARC. HDTrans (Sridhar et al. 2006) is a lightweight open-source translation framework that targets IA-32 and focuses on performance at the expense of user-friendliness. Finally, Pin (Luk et al. 2005) is a dynamic binary instrumentation system that supports IA-32, Intel64, Itanium, and ARM.

Of the instrumentation systems mentioned above, DynamoRIO, Valgrind, and Pin all provide support for multithreaded programs. DynamoRIO switched from a thread-private to a thread-shared code cache implementation when scalability issues arose (Bruening et al. 2006). Valgrind serializes the execution of multiple threads with a thread-locking mechanism. It simplifies Valgrind and instrumentation, but comes with a large performance penalties for parallel programs.

Any instrumentation tool that support Windows applications must provide support for multithreaded programs in order to be widely adopted. Of the aforementioned systems, only DynamoRIO and Pin support Windows applications. While the Windows implementation is outside the scope of this paper, many challenges are common across the Linux and Windows platforms, and several of our algorithms for handling multithreaded programs are platform independent.

8. Conclusions

Dynamic binary instrumentation systems have emerged as invaluable tools for program analysis, simulation, optimization, and security. Most of the prior work discussing DBI infrastructures has focused on executing single-threaded benchmarks, such as SPEC 2000, despite the ubiquity and importance of multithreaded applications today. Handling multithreaded applications in a robust and scalable way is challenging when implementing dynamic instrumentation systems. Therefore, this paper fills a void in the current literature by describing a set of techniques employed in the widely-used Pin instrumentation system that enable robust execution of multithreaded programs. There are two basic challenges. The first is that we need to generate code on the fly without requiring that all threads stop. This has ramifications in the way newly generated code is linked to the rest, the method for resolving indirect branches, and eviction algorithms in the software code cache. The second challenge is that the runtime needs to be thread-aware and thread-safe. This impacts the way we handle system calls and Unix signals. We evaluate Pin with multithreaded programs and show that Pin adds overhead to execution, but preserves the scalability of applications.

Acknowledgments

We would like to thank the members of the Intel Pin development team for their contributions to Pin itself. Without their support, the system would be much less robust and scalable than it is today. We also would like to thank the anonymous reviewers for their careful reviews and insightful feedback.

References

- Ole Agesen. Binary translation of returns. In *Workshop on Binary Instrumentation and Applications*, pages 7–14, San Jose, CA, USA, October 2006.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, Canada, June 2000.
- Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163, Ottawa, Ontario, Canada, 2006. ACM.
- Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, CA, USA, March 2003.
- Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *International Symposium on Code Generation and Optimization*, pages 28–38, San Francisco, CA, USA, March 2006.
- Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *International Journal on High Performance Computing Applications*, 14(4):317–329, 2000.
- Apala Guha, Kim Hazelwood, and Mary Lou Soffa. Reducing exit stub memory consumption in code caches. In *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, Ghent, Belgium, January 2007.
- Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Computer Architecture News*, 33(5):63–68, 2005.
- Kim Hazelwood and Michael D. Smith. Managing bounded code caches in dynamic binary optimization systems. *Transactions on Code Generation and Optimization (TACO)*, 3(3):263–294, September 2006.
- Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.
- Galen Hunt and Doug Brubacher. Detours: binary interception of win32 functions. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.
- Intel. Intel64 architecture memory ordering white paper, August 2007.
- Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp\$im: A pin-based on-the-fly multi-core cache simulator. In *Fourth Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, USA, June 2005.
- Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, USA, June 2007.
- Cristiano Pereira, Harish Patil, and Brad Calder. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 173–182, September 2008.
- Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization*, pages 36–47, March 2003.
- James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.
- Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungle. HDTrans: An open source, low-level dynamic instrumentation system. In *2nd International Conference on Virtual Execution Environments*, pages 175–185, Ottawa, Ontario, Canada, 2006.
- Vijay Sundareshan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *International Symposium on Code Generation and Optimization*, pages 87–97, San Francisco, CA, USA, March 2006.