A Cross-Architectural Interface for Code Cache Manipulation

Kim Hazelwood
University of Virginia

Robert Cohn
Intel Corporation

Abstract

Software code caches help amortize the overhead of dynamic binary transformation by enabling reuse of transformed code. Since code caches contain a potentially-altered copy of every instruction that executes, run-time access to a code cache can be a very powerful opportunity. Unfortunately, current research infrastructures lack the ability to model and direct code caching, and as a result, past code cache investigations have required access to the source code of the binary transformation system.

This paper presents a code cache-aware interface to the Pin dynamic instrumentation system. While a program executes, our interface allows a user to inspect the code cache, receive callbacks when key events occur, and manipulate the code cache contents at will. We demonstrate the utility of this interface on four architectures (IA32, EM64T, IPF, XScale) and present several tools written using our API. These tools include a self-modifying code handler, a two-phase instrumentation analyzer, a code cache visualizer, and custom code cache replacement policies. We also show that tools written using our interface have comparable performance to direct, source-level implementations. Both our interface and sample open-source tools that utilize the interface have been incorporated into the standard distribution of the Pin dynamic instrumentation engine, which has been downloaded over 5,000 times in 18 months.

1. Introduction

Dynamic binary transformation systems are becoming commonplace in both the research and product communities. The ability to manipulate the instruction stream of an executing program enabled by these systems has had numerous implications in program performance, security, and portability. While developing a dynamic binary optimizer and covering the numerous corner cases is a challenge, several research groups have successfully built robust systems capable of executing today's applications with little or no overhead [7, 12, 21].

In working to reduce overall system overhead, a significant observation is that the single largest performance improvement results from the use of code caches. Software-managed code caches serve the role of storing transformed application code to enable reuse. They improve the overall system performance by amortizing the cost of expensive transformations over the entire program execution time.

Providing researchers access to the contents of the code cache enables many powerful opportunities. A user can manipulate the code cache contents to investigate run-time optimizations or security policies; they can instrument and compare applications across several architectures; they can even investigate the code cache implementation itself and develop and compare custom code cache replacement policies. Our users have demonstrated that the possibilities are nearly endless.

While general frameworks have been developed for studying binary instrumentation and optimization, they have not been designed to provide user-level access to the code cache. In fact, most frameworks have gone to great lengths to mask the presence of a code cache from the user by converting cached instruction addresses to their corresponding addresses in the original application before presenting the addresses to the user. Therefore, code cache related investigations have required access to the source code of a dynamic binary optimizer. Unfortunately, most of the popular and robust systems are not open source. Even with access to source code, the lack of an API that allows fine-grained control but still abstracts away the details prevents a casual user from rapidly constructing sophisticated analysis tools. These traits complicate the task of investigating and manipulating code cache contents and implementations. The goal of our work is to remedy this situation, and to provide a general framework for studying and altering code caches using a clean, well-designed interface.

Our framework is built upon Pin [21], a dynamic instrumentation tool developed at Intel Corporation that uses a code cache to amortize the cost of program instrumentation. We provide access to the statistics and details of the code cache contents, as well as support for insertions to and deletions from the code cache. This functionality enables implementation and wall-clock comparisons of client tools. Furthermore, both Pin and our cache interface are portable

Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)
0-7695-2499-0/06 $20.00 © 2006 IEEE

across four Intel architectures: IA32 (32-bit x86), EM64T (64-bit x86), IPF (64-bit Itanium), and XScale (ARM). Therefore, a user can evaluate their technique on four architectures using the same platform-independent API. Finally, an added benefit of building our system upon Pin is that users also have access to Pin's instrumentation client interface, which provides support for lightweight profiling and source-level information about the executing application.

To our knowledge, this is the first body of work that focuses on a client interface to code caches. Furthermore, this is the first work that compares code cache behavior across multiple architectures. The specific contributions of this paper are as follows:

- An introduction and evaluation of a general framework for code cache investigation and manipulation in the Pin dynamic instrumentation system.
- An investigation and comparison of code cache behavior on four different instruction-set architectures: IA32, EM64T, IPF, and XScale.
- An overview of several plug-in tools written using our code cache interface, including a self-modifying code handler, a two-phase instrumentation analyzer, a cache visualization GUI, and custom code cache replacement algorithms.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the Pin dynamic instrumentation tool as well as a historical review of code caches. Section 3 introduces the C/C++ API of the code cache interface that can be used to build ISA-independent tools and quantifies the overhead of using our API as compared to native Pin performance. Section 4 describes several example tools that were written using the interface. Finally, Section 5 summarizes our contributions and concludes.

2. Background

In this section, we describe several of the existing dynamic binary transformation systems, and provide an overview of the role of code caches in their implementation. Then we discuss the Pin dynamic instrumentation system, upon which our interface was built, and describe the internal structure and algorithms implemented in its code cache.

2.1. Dynamic Binary Transformation

Several systems fall under the category of dynamic binary transformation systems. Dynamic optimizers, such as HP's Dynamo and DELI [4, 13], HP/MIT's DynamoRIO [7], University of Virginia's Strata [25], and University of Minnesota's ADORE [10] perform run-time optimizations in a transparent manner in order to improve performance and/or enhance security. Other systems, such as IBM's DAISY and BOA [1, 15], and Transmeta's CMS [12] focus on dynamic translation to allow pre-compiled applications to run on new or incompatible hardware. Still other systems use binary transformation to introduce instrumentation to existing applications, such as Intel's Pin [21], Julian Seward's Valgrind [22], Microsoft's Detours and Vulcan [20, 26], University of Maryland's Dyninst [8], and Sun's DTrace [9]. Finally, systems such as IBM's Jikes RVM [2] perform adaptive re-optimization in addition to just-in-time compilation of Java programs.

Nearly all dynamic binary transformation systems employ one or more *code caches* to store altered copies of the original instructions. This approach has been very successful in amortizing the costs of transformation, often resulting in a net performance improvement of the overall system.

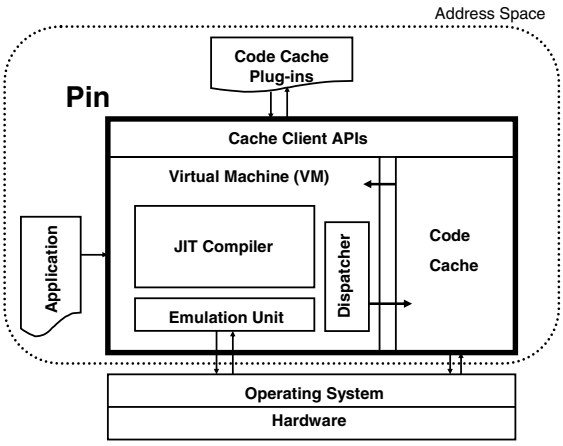

Figure 1. Software architecture of the code cache plug-in interface.

2.2. Pin Overview

Pin [21] is a dynamic binary rewriting system developed at Intel Corporation. It supports IA32, EM64T, ARM and IPF programs for Linux, Windows, and FreeBSD. Pin was designed with instrumentation in mind. Hence, instrumenting a program is both easy and efficient. A user can write instrumentation tools using an API that is rich enough to allow many plug-ins to be source compatible for all the supported instruction sets. Pin allows a tool to insert function calls at any point in the program. It automatically saves and restores registers so the insert call does not overwrite application registers. Pin reallocates registers and inlines instrumentation to improve performance.

Figure 1 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code

cache, and an instrumentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly, such as system calls that require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code.

A thorough description of the internal functionality of Pin is provided by Luk et al. [21].

2.3. Pin's Code Cache Structure

Like many other binary modifiers, Pin uses a code cache to amortize its overhead. Pin's code cache is partitioned into multiple equal-sized *cache blocks* (see Figure 2) that are generated on demand. This configuration allows the code cache to adapt to increasing application requirements. In the default case, each cache block is sized at `(PageSize * 16)`, which evaluates to 64 KB on IA32, EM64T and XScale, and 256 KB on IPF. Pin's entire code cache is unbounded by default on IA32, EM64T and IPF, while a 16 MB limit is placed on the XScale code cache due to a hard limit on XScale resources. Users may override the code cache or block size limits via command-line switches. Alternatively, users may dynamically adjust these values at run time using our client API as we will describe in Section 3.

Code traces – or more specifically *superblocks* – are used as the basis for instrumentation and code caching in Pin. Unlike other systems, which provide a separate cache for basic blocks and superblocks, Pin combines all regions into a single code cache. Just before the first execution of a basic block, Pin speculatively creates a straight-line trace of instructions that is terminated by either (1) an unconditional branch, or (2) an instruction count limit. The first termination condition – unconditional branches – is very different from other dynamic transformation systems which follow the execution path through unconditional branches while generating traces [14]. Pin's approach stems from the fact that it is designed to be an instrumentation system, so it made more sense to ensure that traces reside in contiguous memory before allowing users to add instrumentation.

After generating a trace, Pin immediately places it in the code cache and updates the cache directory. The directory is a hash table of code cache contents that is indexed by both the original application address of the first instruction in the trace and a register binding at that trace entrance: $< originalPC, registerBinding >$. Recording the register bindings allows Pin to reallocate registers across trace

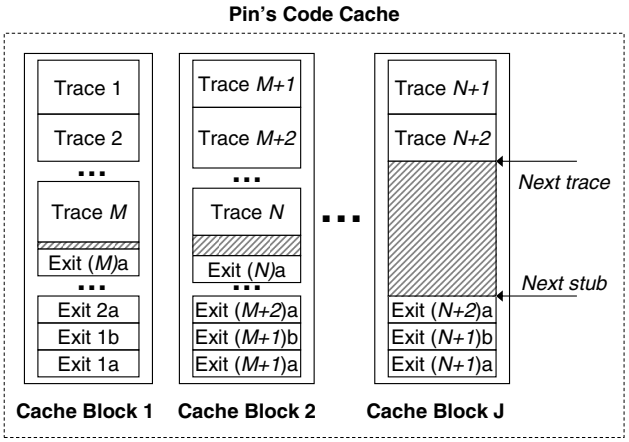

Figure 2. Pin's code cache consists of multiple cache blocks. Traces are inserted at the *top* of a cache block. Exit stubs are inserted at the *bottom*.

boundaries. A side-effect is that multiple traces may exist in the code cache with the same starting address but different register bindings.

For every potential off-trace path, Pin generates an exit stub, which redirects control back to the VM and passes information about the next trace to execute. Over time, Pin will patch any branches targeting exit stubs directly to the target trace in the code cache (this process is also called *linking*). For this reason, the code cache is configured such that the exit stubs are geographically separated from the traces, as shown in Figure 2. This configuration is designed to improve the hardware instruction-cache performance because in the common case, traces will branch to other nearby traces and not to the distant exit stubs.

Pin links cached traces proactively. As each trace is inserted into the code cache, all off-trace branches are immediately patched to any targets that already reside in the code cache. For those targets not yet present, a special marker is placed in the code cache directory. This marker allows future traces to link any previously-generated branches in other traces to the new trace.

To keep memory usage under control, Pin employs a shared code cache across all threads in a multithreaded application, and employs a staged flush algorithm to handle code invalidations and other consistency events. Each cache block (see Figure 2) has an associated stage that indicates the number of flushes that have been triggered since the program began. As each thread enters the VM, it is redirected to the cache blocks marked with the latest stage, and the thread count for the previous stage is decremented. When the thread count for a particular stage reaches zero, the space allocated for all cache blocks marked with that stage is freed.

Callbacks	Actions	Lookups	Statistics
PostCacheInit	FlushCache	TraceLookupID	MemoryUsed
TraceInserted	FlushBlock	TraceLookupSrcAddr	MemoryReserved
TraceRemoved	InvalidateTrace	TraceLookupCacheAddr	CacheSizeLimit
TraceLinked	UnlinkBranchesIn	BlockLookup	CacheBlockSize
TraceUnlinked	UnlinkBranchesOut		TracesInCache
CodeCacheEntered	ChangeCacheLimit		ExitStubsInCache
CodeCacheExited	ChangeBlockSize		
CacheIsFull	NewCacheBlock		
OverHighWaterMark			
CacheBlockIsFull			

Table 1. A subset of the actions and opportunities available in the code cache API.

3. The Code Cache API

Now that we understand the structure of Pin and its code cache, we can describe our interface for inspecting and controlling the code cache. We implemented the API using a methodology similar to Pin's existing approach to user-defined instrumentation. The software architecture is depicted in Figure 1. As the figure indicates, there are three programs executing: the input application (such as a SPEC benchmark), the Pin instrumentation system, and the user's code cache plug-in that alters the cache contents, specifies the replacement policy, and/or collects code cache information from Pin. The cache plug-in communicates with Pin using our code cache plug-in API described in this section.

3.1. The Client Interface

The API calls we provide to our users (shown in Table 1) can be grouped into four categories: callbacks, actions, lookups, and statistics. *Callbacks* allow our users to be notified when key code cache events occur. *Actions* can be invoked any time the plug-in has control (such as during a callback). *Lookups* provide access to Pin's internal data structures that keep track of the code cache's contents. Finally, the plug-in can inspect *statistics*. We discuss each of these categories in the following paragraphs.

Callbacks Our API allows a user to register routines to be executed after specific events (such as initialization, insertions, etc.) occur in the code cache. These routines give the client tools an opportunity to "gain control" of the executing application in order to perform whatever actions they wish. Implementing a code cache replacement policy, for example, simply requires a user to register a routine that will be called any time the code cache exceeds a high water mark and/or completely fills. Ten events for which we provide callbacks are listed in the leftmost column of Table 1.

Actions While the callback opportunities in the previous section allow a client to gain control, the API routines in this section allow a client to then invoke actions in the code cache. One particularly useful action is trace invalidation. Users have found that this action allows them to direct the regeneration of code (after it is determined to be *hot*, for example). After an invalidation, the code will likely be regenerated, and during that phase, the user can add new instructions or change some other trait of the newly-generated code before it is re-inserted into the code cache.

While trace invalidation is a single API call to our users, it actually performs quite a bit of work behind the scenes. This includes converting the program address to a code cache address (if necessary), unlinking all incoming and outgoing branches that originate from or target other cached traces, updating all of the internal data structures, ensuring correctness in light of multithreading concerns (i.e., ensuring that no other thread is executing the invalidated trace), etc. All of this complexity is masked from our users and is embedded into a single API call.

Cache Lookup The API routines in the *lookup* category allow a plug-in to directly access the code cache directory that contains information about all of the traces and exit stubs in the code cache. As described in Section 2.3, the directory is simply a hash table that contains the entries and details of traces currently residing in the code cache. This information can be used either to collect information or to guide actions. Invalidating a trace, for example, requires knowledge of either the original program address or the code cache address. The mapping from original to code cache addresses is available via code cache lookup routines.

Statistics Our final API category exports various summary statistics concerning the contents, history, and footprint of the code cache. These statistics can be useful not only for an analysis of the code cache's contents, but also to trigger code cache events (such as trace invalidation or full code cache flushes) if called for by the design.

Summary In this section, we provided a high-level view of our API for writing code cache manipulation tools, and

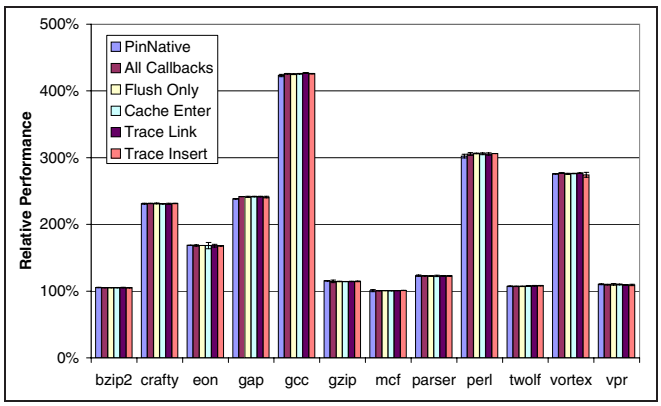

Figure 3. Wall-clock performance comparison of Pin without callbacks to Pin with various code cache callback combinations. All values are relative to native performance (without Pin). Values below 100% represent speedups with respect to native performance.

more detailed information is available in the full Pin manual [11]. It's important to emphasize that the code cache API is provided *in addition to* Pin's extensive instrumentation API defined in the user manual. Therefore tools can be designed that perform both instrumentation and code cache manipulation, as we will demonstrate in Section 4.

3.2. System Performance

From a usability standpoint, our goal was is to provide as much information about and access to the code cache as possible without sacrificing the overall system performance. For many of our users' applications, it is very important that the performance of a plug-in style implementation approaches the performance of a direct source-code implementation in order to enable fair, wall-clock comparisons of multiple design choices. For instance, the performance of a code cache management policy implemented using our API should provide a realistic representation of the performance of a direct implementation of that policy.

One way we accomplished this goal is that all of the callbacks out to a client's routines occur at a point when Pin's VM has control (i.e., Pin's own code is executing, not the application's code). Therefore, there is no need to perform an expensive state switch to/from the application's register state in order to execute a user's callback routine. (Note: This register state switch is a major cause of slowdown in standard binary instrumentation.) This fact allows us to provide our code cache interface with a much lower performance penalty than standard instrumentation tasks.

To confirm that our API doesn't needlessly introduce overhead to the design, we measured the wall-clock performance of executing the SPEC benchmarks[1] while exercising our API with empty callback routines. This allows us to isolate the overhead of our API from the overhead of the additional functionality the user incorporates into the design.

Figure 3 shows the result of our measurements, relative to the native run time of the benchmarks (without Pin). For each benchmark, the leftmost bar reports the run time of executing the benchmark under Pin's control, without exercising our API. The second bar exercises several of our callback opportunities[2], including calling routines when (a) the cache is full, (b) control enters the cache, (c) a trace is linked to another, and (d) a trace is inserted into the code cache. The next four bars isolate the overhead of each of the four callback opportunities described above. As we can see from the figure, the overhead of each callback mechanism almost always falls within the noise of wall-clock timing results, simply because our callback API does not trigger a register state switch. This remains true when we exercise several callback opportunities at once (`All Callbacks`) or when we exercise a callback opportunity that occurs fairly frequently at run time (`Trace Link`).

In the event that a user adds complex logic to their user-level callback routines, we would expect to see the same slowdown that they would experience were they to directly implement that complex logic in the source code. We have validated this internally by comparing direct and API-based implementations of the code cache replacement policies described in Section 4.4.

In summary, users should feel confident that the cache API provides both the ability for rapid prototyping, and comparable performance to a direct implementation of their desired functionality in the source code. The results (even wall-clock timing results) that users will acquire are both relevant and realistic, yet the system is still robust, portable, and easy-to-use.

4. System Utility

The API described in Section 3 can be used to write a variety of architecture-independent code cache tools. We use some sample tools to illustrate the utility of the API. In this section, we present the details of tools designed for code cache introspection, manipulation, visualization, and replacement. All of the tools were very easy to develop due to the richness of our API, and generally required less than 100 lines of commented C++ code. The tools we present are freely available (including the source code) as part of the standard distribution of our code cache client.

[1] We report the median run time of five executions of the reference input set. The error bars indicate the run-time variance.

[2] We are simply trying to isolate the overhead of the API, therefore we do not perform any complex logic in the callback routines.

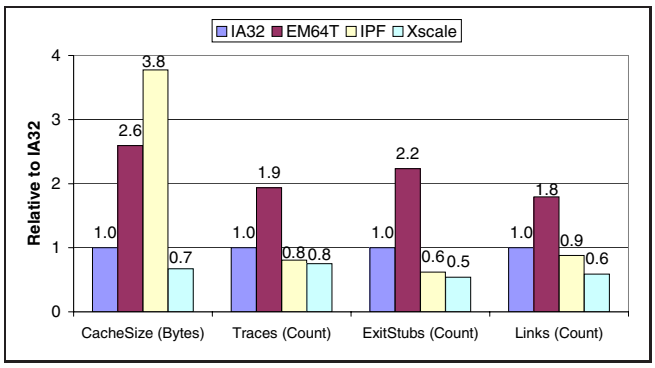

Figure 4. Code cache statistics of SPECint2000 on four architectures, with IA32 as a baseline.

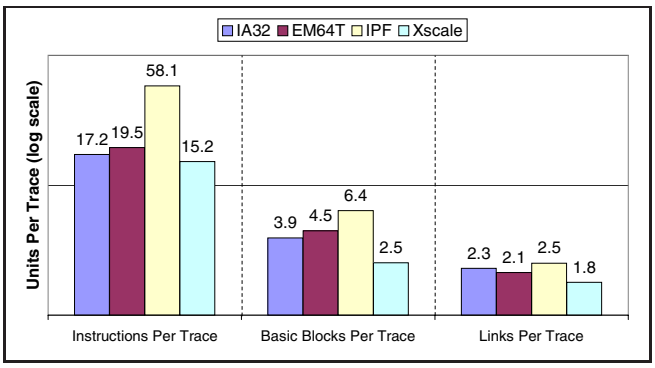

Figure 5. Trace statistics on four architectures averaged across SPECint2000.

4.1. Cross-Architectural Code Cache Comparison

When researching software code caches, it is necessary to understand the actual contents of the code cache. Our interface takes this one step further, and allows a user to compare code cache contents across four architectures: IA32, EM64T, IPF, and XScale. This permits researchers to understand the ISA tradeoffs of every design choice. Our first sample tool compares code caches across the four architectures in terms of the final unbounded code cache size, the number of traces and exit stubs generated, the average instruction length of a trace, and the number of times the system patches trace branches to link to other traces. The tests were performed using the SPECint2000 benchmarks with the training input set to allow a fair comparison with the XScale system, which does not have sufficient memory to execute the reference input set. (The reference inputs result in similar conclusions for the remaining architectures.)

As we can see from Figures 4–5 the code cache behavior on one architecture is not necessarily indicative of the behavior on another architecture, despite the fact that the same dynamic instrumentation system was used to perform the study. As we move to 64-bit architectures, such as EM64T and IPF, we see that code cache pressure increases. This is indicated in Figure 4 by the 2.6X and 3.8X code cache expansion on IPF and EM64T, respectively, as compared to IA32. Furthermore, we see that more code is generated on EM64T than on IA32. There are a number of contributing factors. First, the instruction encoding is less dense for 64-bit ISAs. Second, the larger number of registers gives Pin more freedom to do code expanding optimizations.

In Figure 5, we see that traces on IPF are much longer. This is expected because of the padding nops required by instruction bundling and the aggressive use of speculation. We can validate this using the code cache API by inspecting the instructions after they are inserted into the code cache to measure the number of nops and use of predication.

4.2. Self-Modifying Code Handler

Self-modifying code is a challenge to handle efficiently in a dynamic translator [6, 12]. The problem occurs when an application executes some code, modifies it, and then executes the new code at the same address. After the first execution, a dynamic translator will save a copy of the code in its code cache. When the modified code is executed, the dynamic translator must detect that the code it has in its cache is no longer valid. Without any detection, it will continue to execute the old code and the program will eventually fail.

Several mechanisms have been proposed to detect self-modifying code such as write-protecting code pages, checking store addresses, and inserting extra code to check that instruction memory has not changed. Access to source code and an intimate understanding of the code generation system is typically required to study solutions for self-modifying code. By combining an instrumentation and cache control API, it is possible to implement a simple solution in 15 lines of code. In Figure 6, we show the code written by one of our users. The function `InsertSmcCheck` is the instrumentation function which is passed a list of instructions in the trace. While an instrumentation function typically inserts calls to count basic block executions or record the effective address for a memory reference, this particular function makes a copy of the original instructions in the trace and inserts a call to `DoSmcCheck`, passing it the address in memory of the instructions and the saved copy. When the trace is executed, it calls `DoSmcCheck`. This function compares the current contents of the instruction memory against the saved copy. If it has changed, it invalidates the cached copy of the trace and uses `PIN_ExecuteAt` [24] to re-invoke the trace. Note that this example is just a demonstration – it does not handle a trace that overwrites its own code (after the check) or multithreading.

It is also possible to study mechanisms that use page

Self-Modifying Code Handler

```
void main (int argc, char **argv)
{
  PIN_Init(argc, argv);
  TRACE_AddInstrumentFunction(InsertSmcCheck,0);
  PIN_StartProgram(); // Never returns
}

// Pin calls this function every time a new trace is
//  encountered
void InsertSmcCheck ()
{
  traceAddr = (VOID *)TRACE_Address(trace);
  traceSize = TRACE_Size(trace);
  traceCopyAddr = malloc(traceSize);
  if (traceCopyAddr != 0)
  {
    memcpy(traceCopyAddr, traceAddr, traceSize);

    // Insert DoSmcCheck call before every trace
    TRACE_InsertCall(trace,
          IPOINT_BEFORE, (AFUNPTR)DoSmcCheck,
          IARG_PTR, traceAddr,
          IARG_PTR, traceCopyAddr,
          IARG_UINT32, traceSize,
          IARG_CONTEXT, IARG_END);
  }
}

// This function is called before every
//   trace is executed
VOID DoSmcCheck(VOID * traceAddr, VOID * traceCopyAddr,
                USIZE traceSize, CONTEXT * ctxP)
{
  if (memcmp(traceAddr, traceCopyAddr, traceSize) != 0)
  {
    smcCount++;
    free(traceCopyAddr);
    CODECACHE_InvalidateTrace((ADDRINT)traceAddr);
    PIN_ExecuteAt(ctxP);
  }
}
```

Figure 6. A code cache client tool (developed by one of our users) that detects and handles self-modifying code.

protections by instrumenting memory management system calls. Mechanisms that watch store addresses can be implemented by instrumenting memory store instructions. Note that the example is architecture independent and allows the writer to focus on the detection mechanism and not the details of managing the cache, inserting extra code, and the idiosyncrasies of the instruction set.

4.3. Two-Phase Instrumentation

Instrumentation is a powerful technique for observing dynamic program behavior, but it can be costly. Collecting a control-flow profile may not reduce the performance of a program significantly, but observing every memory instruction can slow down a program by 25x or more, depending on the amount of work done to process each memory reference. Hardware performance monitors incur much less overhead by sampling, but tools are limited to observing events that the original hardware designers anticipated. Instrumentation allows a tool to inspect all of the architectural state.

Arnold [3] and Hirzel [19] propose using a combination of instrumentation and sampling to get the flexibility of instrumentation and the performance of sampling. They duplicate whole procedures and insert checks to decide whether to execute the instrumented or un-instrumented code. If instrumented code is executed infrequently, the cost is dominated by the overhead of the checks, not the instrumentation.

Dynamic compilation systems use a simpler, but more limited technique called two-phase compilation [5, 7, 23]. In the first phase, the compiler inserts instrumentation to count execution. When the instrumentation routines indicate that a trace or routine is hot, it is recompiled with optimization and no instrumentation. In this section we show how cache control in an instrumentation system can be used to implement two-phase instrumentation, greatly improving the instrumentation performance. In two-phase instrumentation, all traces initially have heavyweight instrumentation. When a trace is known to be hot, it is recompiled without instrumentation.

The goal of our two-phase instrumentation tool is to observe a memory address stream to find the addresses of instructions that are likely to reference global data. The information can be used for an optimization in a static compiler that speculatively keeps global variables in registers. In the baseline case, we check addresses for the entire run of the program. Memory instructions are instrumented to store their effective address in a buffer, and the buffer contents are processed when full. A conservative static analysis is used to eliminate the instrumentation of instructions that are known to touch only stack or global data.

As can be seen from the data series labeled `full` in Figure 7, the slowdown for programs with full profiling varies from no overhead to as much as 14.9x slower with an average of 6.2x (520%). Two phase instrumentation is implemented by instrumenting traces to observe memory references and count executions. When the count exceeds a threshold, the trace expires and we invalidate the trace from the code cache (similar to the self-modifying code example). On the next execution of the trace, it is retranslated with no instrumentation. Eventually, all frequently executed traces are uninstrumented, running at full speed. In Figure 7, the data series labeled `100` shows the slowdown for program with two-phase instrumentation and a threshold of 100 executions. The maximum slowdown has been reduced to 5.9x with an average of 2.0x.

An assumption in two-phase instrumentation is that the early behavior is a good predictor for the entire execution of the program. For varying thresholds, we measured how well the early executions of a trace predict the behavior of the full

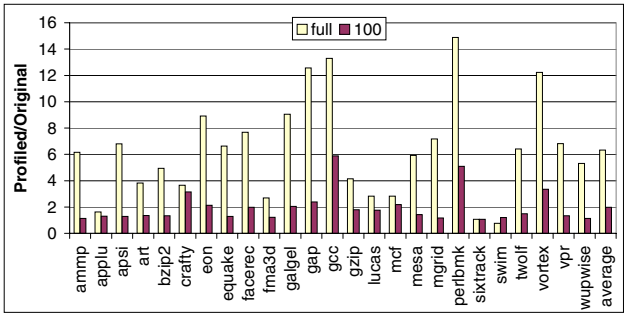

Figure 7. Memory profiling slowdown comparison of full-run profiling and two-phase profiling with a threshold of 100.

program. In Table 2, we see that on average 5% of dynamic references were incorrectly predicted to be unaliased with global data (false positive). The average error is misleading because one program (wupwise), has 100% error, but all other programs have a maximum error of 0.25%. Table 2 also shows that false negatives are rare as we find almost all of the unaliased references. Finally, we use the code cache API to measure the size of the expired traces. The results in Table 2 indicate that typically 1/3 of the code that is executed at least once exceeds the execution threshold and is discarded. This is a good indicator of the cost for two-phase instrumentation as most of the time overhead comes from the extra compilation of expired traces.

Converting a full program profiler into a two-phase profiler adds about 30 lines of C code to the tool. (The full source code for the two-phase profiler is distributed with the API. For brevity, we do not include it in this paper.) At the head of the trace, we insert a call to decrement the per-trace counter. When the count reaches zero, we call `CODECACHE_InvalidateTrace` to remove the instrumented trace from the cache and record that this address has expired. The next time the code is executed, it is not found in the code cache, causing a new trace to be fetched. The instrumentation tool notes that the trace has expired, and chooses not to instrument it.

Arnold-Ryder and bursty sampling have the potential to be more accurate with lower overhead. However, it also requires duplicating all the code and finding the proper places to switch between instrumented and uninstrumented copies, which makes it harder to implement and generalize. Our experience with a publicly available tool with a large user community [21] is that the accuracy and overhead of phased instrumentation is sufficient for many tasks, and the simplicity of implementation makes it a tractable technique. For users that need the higher accuracy, we are investigating simple extensions to the code cache API to support the presence of multiple versions of a trace in the code cache

	100	200	400	800	1600
speedup over full	3.34	3.31	3.23	3.29	3.24
false negative	2.59%	1.07%	1.06%	0.86%	0.82%
false positive	5%	5%	5%	5%	5%
expired traces	38%	37%	35%	33%	31%

Table 2. Performance and accuracy of two-phase profiling with varying thresholds.

at a given time, and techniques for dynamically selecting between the versions at run time.

4.4. Code Cache Replacement

Like most caching problems, code caches have several capacity and consistency issues that result in the need for a cache management policy. Earlier studies have shown that the code expansion exhibited in code caches results in a significant motivation for bounding the size of a code cache [17, 18], particularly when executing large interactive applications. An equally pressing motivation results from dynamically-loaded and -unloaded libraries, self-modifying code, and multi-threaded applications that result in consistency issues and require the removal of stale translations from the code cache.

Several code cache replacement algorithms have been proposed that focus on maintaining the application's current working set in the code cache while minimizing code cache maintenance overhead. All such investigations have required access to the source code of the dynamic binary transformation system, therefore the domain has been restricted to a few researchers. Unless the system has been designed with a user-interface to the code cache in mind, access to the source code isn't always the best solution for investigating code caches. The assumptions of the code cache are often tightly integrated into the design in unseen ways.

As mentioned earlier, a compelling motivation for our code cache plug-in interface is that it is the first to allow users to implement a complete, custom code cache replacement policy without the need to access the source code of the dynamic transformation system. This motivation is even more compelling by noting that of the dozen or so dynamic transformation systems listed in Section 2.1, very few are open source and readily available.

Some of the more common cache management policies include *flush-on-full*, first-in first-out *(FIFO)*, and least-recently used *(LRU)*. Figure 8 shows the code necessary to implement a standard flush-on-full code cache replacement policy, and illustrates the interface's ease-of-use. As shown in the figure, only two API calls were needed (`CODECACHE_CacheIsFull` and `CODECACHE_FlushCache`) to implement this policy aside from the boilerplate Pin instrumentation routines:

Full Code Cache Flush

```
void main (int argc, char **argv)
{
    PIN_Init(argc, argv);
    CODECACHE_CacheIsFull(FlushOnFull);
    PIN_StartProgram(); // Never returns
}
void FlushOnFull ()
{
    CODECACHE_FlushCache();
}
```

Figure 8. Sample code for a flush-on-full code cache replacement policy implemented as a plug-in utility using our API. When the code cache signals that it is full, this routine flushes the entire code cache.

Medium-Grained FIFO Implementation

```
void main (int argc, char **argv)
{
    PIN_Init(argc, argv);
    CODECACHE_CacheIsFull(FlushOldestBlock);
    PIN_StartProgram(); // Never returns
}
void FlushOldestBlock ()
{
    static int nextBlockId = 1;
    CODECACHE_FlushBlock(nextBlockId++);
}
```

Figure 9. Sample code for a medium-grained FIFO replacement policy. When the code cache signals that it is full, this routine flushes the oldest cache block (which contains multiple traces).

`PIN_Init` and `PIN_StartProgram`. When executed, this code will *override* the default mechanisms already implemented in Pin.

Moving to a slightly more sophisticated cache policy, Figure 9 shows the code necessary to implement the medium-grained FIFO replacement policy proposed by Hazelwood and J. Smith [16]. This policy deletes an entire cache block (containing numerous traces) at a time. This policy results in an improved cache miss rate compared to flush-on-full (because there are more traces residing in the code cache on average), without the high invocation count and link repair overhead of a fine-grained trace-at-a-time flush policy. Again, we see that the implementation is extremely straightforward (only one more API call than the previous policy) because the system takes care of all directory updates and link repairs behind the scenes.

The `TraceInvalidate` mechanism makes it straightforward to implement a pure `FIFO` replacement policy, while the instrumentation API makes it straightforward to compute hit rates and implement `LRU` by inserting counter code into the traces that execute. More sophisticated policies that take into account threading simply require the use of our high-water mark detection API, which allows the system to initiate the flushing process early enough to allow threads the opportunity to phase themselves out of the old code before freeing the associated code cache memory.

4.5. Code Cache Visualization

For many applications, insight can be gained by visualizing the contents and structure of the code cache. To this end, we developed a code cache visualization tool, shown in Figure 10. The *Code Cache GUI* is a graphical tool for browsing and manipulating the content of the code cache. Internally, this tool has been used for debugging, tuning, and verification of Pin's code cache and the algorithms and policies associated with it.

The tool intercepts code cache events such as the creation of a new trace and renders the information in a graphical form. A user can browse this information interactively and also trigger actions such as cache flushes.

As can be seen from the screenshot in Figure 10, the main window is subdivided into five areas. From top to bottom there is (1) a status line (2) a trace table (3) individual trace information (4) code cache actions, and (5) breakpoints.

The *Status Line* area shows various summary statistics for the traces currently residing in the code cache. The *Trace Table* lists the main characteristics of all (or a subset of) the traces in the code cache. For each trace it displays information such as the trace ID, the original address, the code cache address, the trace size, the originating function name, etc., and it is possible to sort the table by any of these criteria. The *Individual Trace* area allows you to inspect and/or flush a particular trace from the code cache. The *Cache Action* area is a collection of actions affecting the entire code cache, e.g. flushing the entire code cache or writing all the traces into a file which can later be reread. (It is possible to reload a code cache log file back into the GUI for offline investigation.) Finally, the user can specify various breakpoints either symbolically or by address, which, when hit, will cause the tool to stop processing further traces and effectively stall the instrumented application.

4.6. Dynamic Optimizations

The code cache API can also be used to supplement the functionality of existing dynamic optimizers. As a demonstration, we wrote a simple tool to dynamically optimize programs that do integer divide by powers of two. In the first phase of program execution, we do value profiling of

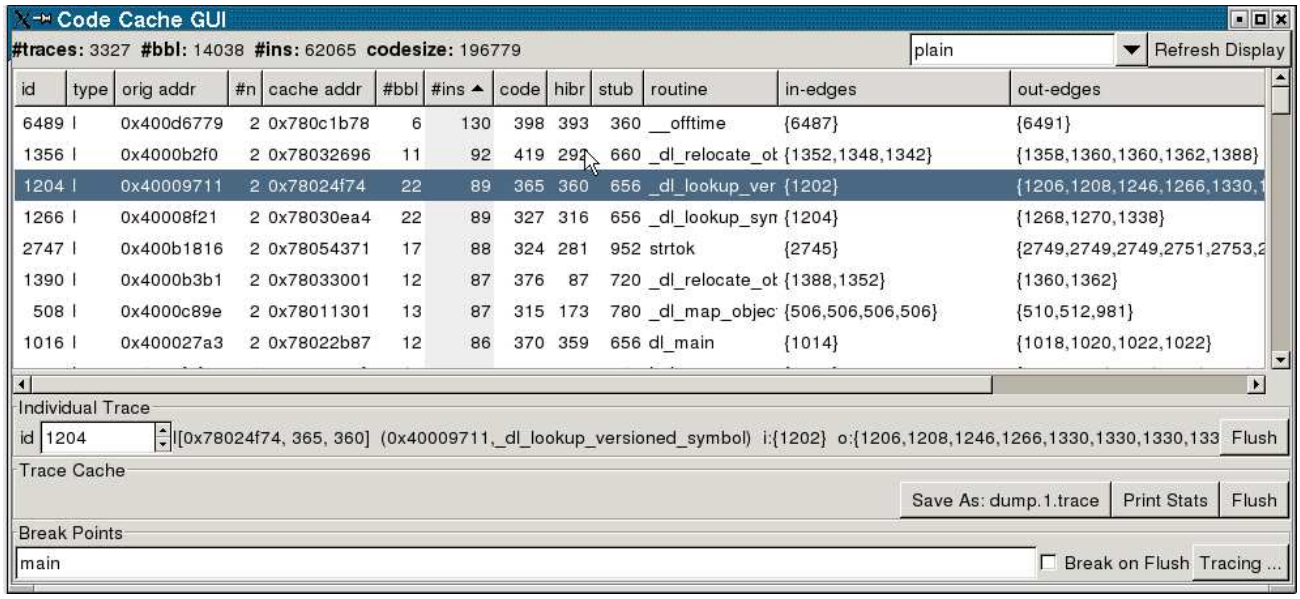

Figure 10. Screen shot of a code cache visualization tool that is driven by our plug-in interface.

the operands of integer divide instructions. In the next phase, we remove the instrumentation and strength reduce divides with frequently occurring divisors, e.g. (a/d) becomes $(d == 2)\ ?\ (a >> 1) : (a/d)$.

One of our users has also implemented a multi-phase optimization for prefetching. The tool begins by profiling for hot traces. When discovered, the traces are then invalidated and re-instrumented to profile for strided memory references. Finally, in the third phase, traces are regenerated to include prefetches with the appropriate stride.

4.7. Tool Summary

The sample uses of our code cache interface described in this section provide a feel for the utility and ease-of-use of our system. Each tool will execute on four architectures[3] and three operating systems. All of the tools were implemented using less than 100 lines of C++ code, with the exception of the Code Cache Visualization Tool, which also included about 500 lines of Python code for rendering and formatting the GUI. All of the sample applications are currently shipped with our code cache interface.

Transparent access to an executing application via a code cache is a very powerful opportunity. Just as ATOM opened the doors for many new opportunities in computer architecture and compiler research 10 years ago, exposing the code cache via a well-defined API could enable a similar level of creative applications. The examples presented in this section only scratch the surface of possible applications.

[3]Pin supports all four Intel architectures, yet those architectures are different enough to provide insight into other existing or new architectures.

5. Conclusions

Code caches are vital to the performance of any dynamic binary transformation system, but they also present a unique opportunity to access the instructions and state of an executing application. In the past, code cache investigations have been hindered by the need for a general framework that exposes the details of a code cache in a real system. This paper addresses that need and introduces a low-overhead but highly functional API for accessing and altering the details of the code cache in the Pin binary instrumentation system. We demonstrated the utility of this API by using it to perform an architectural comparison of code caches, to handle self-modifying code, to investigate two-phase instrumentation, to develop a code cache visualization GUI, and to implement several code cache replacement algorithms. Furthermore, we showed that implementing code cache replacement policies using our API approaches the performance of a direct implementation, without the need for access to or an understanding of the source code of Pin.

Acknowledgments

We would like to thank Geoff Lowney and the rest of the Intel Pin team for their support and input to our API design. We also thank Robert Muth for his code cache visualization tool and his contribution to an earlier draft of this paper. We'd also like to acknowledge Alex Skaletsky for his self-modifying code handler and John Pieper for his prefetch optimizer. Finally, we thank the anonymous reviewers for their helpful feedback.

References

[1] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, December 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[3] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *ACM Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[5] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *36th International Symposium on Microarchitecture*, pages 191–201, December 2003.

[6] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *3rd International Symposium on Code Generation and Optimization*, March 2005.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First International Symposium on Code Generation and Optimization*, pages 265–275, March 2003.

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

[9] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, pages 15–28, 2004.

[10] H. Chen, J. Lu, W. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference*, pages 241–255, 2004.

[11] R. Cohn and R. Muth. Pin 2.0 user guide, July 2004.

[12] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *First International Symposium on Code Generation and Optimization*, pages 15–24, March 2003.

[13] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *35th International Symposium on Microarchitecture*, pages 257–268, 2002.

[14] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, October 2000.

[15] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th International Symposium on Computer Architecture*, pages 26–37, June 1997.

[16] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *2nd International Symposium on Code Generation and Optimization*, pages 89–99, Palo Alto, CA, March 2004.

[17] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *6th Workshop on Interaction between Compilers and Computer Architectures*, pages 102–110, February 2002.

[18] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In *36th International Symposium on Microarchitecture*, pages 169–179, San Diego, CA, December 2003.

[19] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[20] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, July 1999.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[22] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[23] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 1–12, April 2001.

[24] H. Pan, K. Asanovic, R. Cohn, and C.-K. Luk. Controlling program execution through binary instrumentation. In *Workshop on Binary Instrumentation and Applications (WBIA-2005)*, September 2005.

[25] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *First International Symposium on Code Generation and Optimization*, pages 36–47, March 2003.

[26] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.