Design and Implementation of a Dynamic Optimization Framework for Windows

Derek Bruening
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
iye@mit.edu

Evelyn Duesterwald
Hewlett-Packard Laboratories
Cambridge, MA 02142
duester@hpl.hp.com

Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
saman@lcs.mit.edu

Abstract

In this paper we explicitly address the challenges of building a dynamic optimization infrastructure. Our goal is to provide an engineering manual for researchers to motivate and facilitate engagement in dynamic optimization technology. The dynamic optimization framework we discuss has been implemented for the IA-32 family of architectures in a Windows environment. We describe implementation challenges specific to Windows and issues with multiple threads and cache management. We also give performance results for our implementation and indicate the overheads involved. This framework opens up opportunities for program introspection and performance enhancement as all program information is available at runtime.

1 Introduction

Recent dynamic optimization systems have demonstrated the feasibility and effectiveness of using a software system to optimize a program while it is executing. This ability of adapting a running program to its changing environment provides a promising new approach to overcome many of the obstacles of traditional static compilation.

A major challenge for effective static compiler optimization is the inability to accurately predict dynamic program behavior. Enhancing static optimization with profile information provides only a partial solution since profiles do not indicate phase changes in the program and thus do not provide much help if the program behavior changes frequently within the same region of code. Dynamic compilation can easily handle such a situation by allowing optimizations to be transient and adaptive.

Recent trends in software technology only add to the challenges of static compilation. The widespread use of object-oriented programming languages and the trend toward shipping software binaries as collections of Dynamically Linked Libraries (DLLs) rather than monolithic binaries has resulted in a greater degree of run-time binding. While run-time binding offers numerous engineering advantages, it makes traditional static compiler optimization more difficult, if not impossible. Also, it is difficult to debug highly optimized code. For this reason, many independent software vendors are reluctant to ship software binaries that are compiled with high levels of optimization, in spite of their improved performance potential.

Dynamic optimization has emerged as a response to many of these obstacles. Although a few dynamic optimization systems have been developed and described in the literature [3, 4, 9], further and more widespread progress of dynamic optimization technology has been slow. This is not too surprising considering the significant engineering overhead of merely building the basic code infrastructure to undergo research in dynamic optimization. A major challenge in engineering a dynamic optimizer is the ability to efficiently maintain complete control over the running application. This requires intercepting all abnormal control flow, such as exceptions. This level of control over the application is difficult to achieve without severely penalizing performance. In this paper we explicitly address the challenges of building a dynamic optimization infrastructure. Our goal is to provide an engineering manual for researchers to motivate and facilitate engagement in dynamic optimization technology.

The dynamic optimization framework we discuss has been implemented as a second generation system based on Dynamo [3] for the IA-32 family of architectures in a Windows environment. (Our implementation targets Windows NT and its derivatives (2000 and XP), not Windows 95, 98, or ME.) In this paper we are not considering specific optimizations that may be performed. Instead, we discuss the general issues for building the basic infrastructure.

We first present an overview of our dynamic optimization framework in Section 2. A major source of difficulty in engineering a dynamic optimization system is the degree of interaction with operating system features. Section 3 is devoted to discussing relevant Windows-specific engineering challenges. This includes the construction of the injector to gain control over the running application, and the management of Windows callbacks, exceptions, and asynchronous procedure calls. The handling of multi-threading is discussed in Section 4. We move on to an experimental evaluation of design issues. Section 5 presents general performance results and the performance impact of major design decisions regarding the code cache, one of the core components in any dynamic optimization system. We discuss related work in Section 6 and conclude the paper in Section 7.

2 System Overview

Our system interprets the input program's instructions by copying them one basic block at a time into a *code cache* and executing the copied blocks natively. A unit of code in the code cache is called a *fragment*. When a basic block is copied, its terminating branch is modified to return control to our system. We chose this copy-paste approach over emulation because of the complexity of the IA-32 instruction set.

Our basic blocks are not the same as traditional basic blocks. We do not end blocks at direct unconditional jumps, and we walk into calls, eliminating the call instruction. We do end a basic block at any other control transfer instruction.

The system identifies frequently executed *traces* of straight-line code by first identifying *trace heads*. A trace head is a basic block fragment that is either a target of a backward branch or a target of an exit from an existing trace. Each trace head has a counter which is incremented each time the trace head is executed. When the counter passes a threshold, the system enters *trace creation mode*. In this mode, the trace head and each subsequent basic block that is executed are copied into a new trace fragment in the code cache. Trace creation mode terminates when a backward branch is taken or when another trace is reached. The newly created trace consists of straight-line code with potential exits at the join points of the basic blocks that make up the trace. Indirect branches are inlined into the trace along with a comparison that ensures execution leaves the trace if the target of the indirect branch does not match the target that was recorded when the trace was created.

Fragments in the code cache are *linked* to each other. An exit from a fragment whose target is not yet in the code cache targets an *exit stub*, a small piece of code that records the desired target prior to returning to our system via a context switch. When a fragment exit is linked, the exit jump is modified to jump directly to the destination fragment. Our system performs all possible links when a fragment is first created. We found this up-front linking to have lower overhead than a lazy linking strategy: exiting the code cache the first time any fragment exit is taken in order to make a link.

An indirect branch exiting a fragment cannot be executed directly. In the code cache it becomes a jump to an *indirect branch lookup* routine that translates the target address from an application address to a fragment. If there is no such fragment, a new basic block fragment is created.

A flow chart showing the operation of our system is shown in Figure 1. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions. The highest overhead comes from the transformation of indirect branches, which end up executing far more instructions than the original single instruction. Thus, the two routines that handle them are very time-critical. In our implementation, an indirect branch lookup takes 15 instructions, while the comparison of the target of an inlined indirect branch takes 6 instructions.

Self-modifying code is not currently handled by our system. However, we have yet to encounter it in any of the large Windows applications we have been running. Since the instruction cache on the IA-32 platform is kept consistent with memory in hardware, the mechanism for detecting self-modifying code used by Dynamo [3] cannot be used. One solution is to write-protect every page from which we copy a basic block. Then if the application ever modifies a page whose contents are in our code cache we trap the page fault and flush appropriate fragments from the code cache. Note that new code generated at runtime, such as by just-in-time compilers [2, 17, 10, 14], is fully supported by our system.

3 Windows-Specific Engineering Challenges

In this section, we discuss a number of implementation issues that are specific to Windows. First we mention issues with thread-local state. Then we describe how we inject our system into arbitrary binaries in order to run legacy code. Finally, we explain how our system handles the many types of abnormal control flow in Windows.

3.1 Thread-Local State

Our system needs to save the complete machine context when switching from the code cache to system code, and must restore it when switching back. On

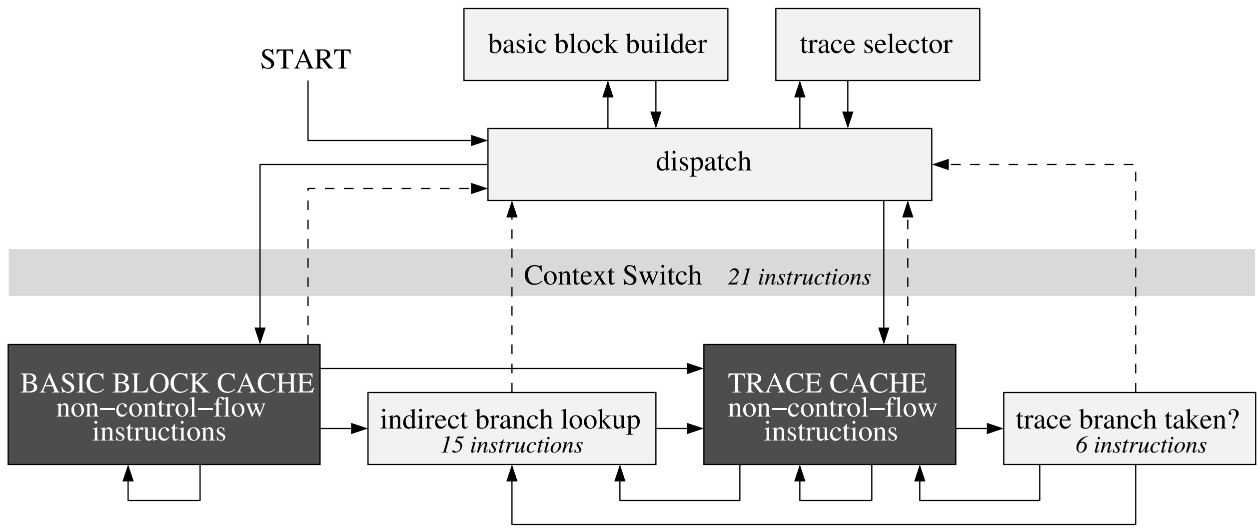

Figure 1: Flow chart of our system. Dark shading indicates application code; the rest is our system. The costs of the most time-critical parts of the system are indicated. For the "trace branch taken?" test, note that we need to save and restore the IA-32 `eflags` register and our own scratch register, accounting for the perhaps surprisingly high overhead.

Windows the context that needs to be saved includes more than just the registers. The operating system supports native threads and keeps some state for each thread. We cannot have our system corrupting that state when it runs in between code cache executions. In particular, we must save and restore the application's current error code. Nearly all Windows API calls set the error code. Our system makes a few Windows API calls in its own routines, and so must save and restore the application's current error code to avoid corrupting it.

3.2 Injector

Our system exports a number of routines allowing an application to start and stop its own execution under our system. When source code is not available, we use our *injector*, which injects our system into an arbitrary binary in order to run legacy code.

There are a number of strategies for injecting a DLL into a process on Windows [22]. Our approach is to create a process for a target executable such that the process begins in a suspended state. We then insert code onto the new process' stack that will load our system's DLL and call its initialization and startup routines. We change the suspended initial thread's program counter to point to this code on the stack, resume the process, and off we go. The target executable is now running under the control of our system.

3.3 Maintaining Control

Our system maintains control of all application code. By maintaining control we mean that none of the original application code is ever executed — all application code is executed via a copy of itself in our code cache. This requires intercepting all transfers of control.

Although for optimization purposes only hot code must be executed by a dynamic optimizer, we envision uses of our runtime system for purposes other than optimization where complete control of an application would be required. For example, various introspective tasks, such as instrumentation or code scanning to implement security protocols, require that all code be examined prior to execution. Furthermore, the bulk of the code executed in a typical Windows application is in callback routines, which would be missed if only normal control transfers were followed.

There are a number of abnormal transfers of control in Windows that require special support to intercept. The Windows operating system implements many features through message passing. The system delivers events to threads by adding messages to the threads' *message queues*. A thread processes its messages asynchronously via callback routines. Our system must intercept these callbacks.

Another source of asynchronous control flow is through the Asynchronous Procedure Call API that is provided by Windows. This allows threads to communicate with each other by posting messages to their respective queues. The details of how these messages are handled are very similar to event callbacks. Our runtime system must follow these calls.

The final source of asynchronous control flow is exceptions. The Windows operating system supports Structured Exception Handling. C++ exceptions in Windows are built on top of this exception mechanism. Our system must intercept the control flow of exceptions in order to maintain control of an application.

Other sources of abnormal control flow include `setjmp/longjmp` and the ability to set a suspended thread's program counter through the Windows API function `SetThreadContext`.

Note that much of the information needed to handle these issues is not officially documented. The Windows source code has not been examined by any of the authors. As such, other methods than those we present here for handling these challenges may exist. All of our information was obtained from observation and from a few books and articles [22, 19, 21, 20].

Also note that we are targeting Windows NT and its derivatives, not Windows 95, 98, or ME. Different techniques may be required to maintain control in these other versions of Windows.

3.3.1 Callbacks

In the Windows model of event delivery, each thread has several message queues. Different types of messages are placed in each queue, but the mechanism for processing all messages is essentially the same. At certain points during program execution, the operating system checks for pending messages. If there are any, the thread's state is saved. Then, for each message, the thread is "commandeered" for use in running the routine that is registered to handle the message.

The left side of Figure 2 illustrates the control flow of a callback for a sample thread. Each column indicates a separate execution context. When the thread enters the kernel via the IA-32 interrupt instruction `int 0x2E`, the kernel checks the thread's message queues. In this case there is a message pending. The kernel then saves the thread's user context and sets the thread up to execute the callback registered for the message's type. The thread returns to user mode through a dispatch routine in the system library `ntdll.dll`. When callback routines finish, they do not normally return. Instead, they indicate that they are finished by either calling the routine `NtCallbackReturn` or by executing `int 0x2B`. This causes the thread to re-enter the kernel. If there are no more messages pending, the kernel restores the saved user context and upon returning to user mode the thread continues with its original execution.

Callbacks can be nested, that is, another callback can be triggered if during execution of a callback the kernel is entered and there are pending messages.

We need to intercept the callback mechanism in order to run callback routines under our control. Fortunately, after the kernel sets up a thread to run a callback it re-enters user mode through an exported routine in `ntdll.dll` called `KiUserCallbackDispatcher`. This is the perfect spot to intercept a callback. We insert a jump instruction at the top of `KiUserCallbackDispatcher` that targets our own routine. Our routine finishes by jumping back to `KiUserCallbackDispatcher`.

We also need to intercept the return of a callback. This is because our execution context contains a few extra pieces of information that we keep in our own data structures. The operating system saves and restores the user context of a thread when it commandeers it to run a callback. However, it only saves and restores the context that it knows about. Our extra state is naturally not saved and restored by the operating system. This extra state includes the application's error code mentioned in Section 3.1 and possibly the real value of a stolen register (see Section 4). Thus we need to be notified both when a callback is about to happen and when a callback is finished so that we can manually save and restore our state. We can catch the return of a callback by watching for both `int 0x2B` instructions and calls to `NtCallbackReturn`.

Note that it is possible (but not officially documented) to access the operating system's thread-local data structure using the segment register `fs` [20]. If we kept our important state in this data structure (in the thread-local storage slots kept there) the operating system *would* save and restore our state for us. This would avoid the need to intercept the end of a callback. However, we have not tested the viability of this scheme.

There is another major effect of callbacks on our system. If a thread is suspended inside the code cache while the operating system commandeers it to run callbacks, we must make sure not to delete the fragment the thread is suspended inside. Unfortunately, there is no way to determine from user mode exactly which instruction a thread was executing when it was commandeered to run a callback. However, this only happens when the thread makes a system call and enters kernel mode through interrupt `0x2E`. This means we cannot delete any fragments containing `int 0x2E`. To make cache management simpler, we redirect all such interrupt instructions through a single point (first storing a return address in our context structure). This allows us to manage the cache without having a lot of undeletable fragments.

3.3.2 Asynchronous Procedure Calls

Asynchronous procedure calls look exactly like callbacks to our system except for the fact that they enter user mode in the routine `KiUserApcDispatcher` in `ntdll.dll`. In all other respects we handle them just like we handle callbacks.

3.3.3 Exceptions

There is a fair amount of information on the details of how Windows handles exceptions [21]. From the point of view of our system, exceptions operate similarly to callbacks. The right side of Figure 2 shows an example of an exception. A faulting instruction causes a trap to the kernel. The kernel saves the user context and then commandeers the thread to run exception-handling code. User mode is re-entered through `KiUserExceptionDispatcher`

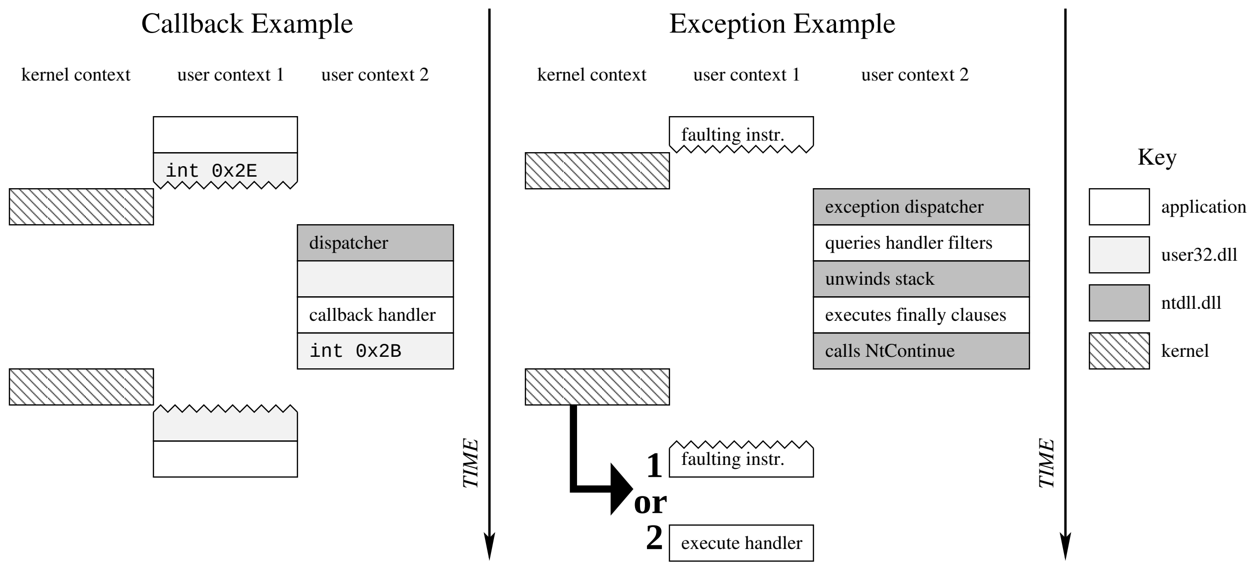

Figure 2: Example control flow for a callback (left) and an exception (right). In each case, execution moves downward, with each of the three columns indicating a separate execution context (all for the same thread).

in `ntdll.dll`. This user-mode code examines potential exception handlers. When an appropriate handler is found and cleanup code has been executed, there are two choices on how to continue. Either the faulting instruction is re-executed (in which case the kernel restores the saved context, just like it does with a callback) or execution continues with the handler and the code after the handler. Both cases go through the routine `NtContinue`. Our system intercepts this routine and changes the target program counter to point to one of our routines. We save the real target and use it as our starting point when the kernel resumes execution with our routine.

As with callbacks, the saved user context that may be returned to points into a fragment in the code cache, making that fragment not deletable. As opposed to callbacks, for an exception we can identify this undeletable fragment because the operating system makes available the context of the faulting instruction.

In addition to the control flow, state saving and restoring, and cache management problems that they share with callbacks, exceptions have another problem. An exception handler has access to the user context of the faulting instruction. The handler can examine the program counter and machine registers. The problem is that these values are incorrect — the program counter points into the code cache, and the registers may not be what the handler expects due to optimizations. Our system needs to transform the context passed to the exception handler to make it appear as though the exception happened in the original application code. Note that this problem of translating a machine context from the code cache to the original application code is not unique to Windows exceptions. The same problem occurs with Unix signal handlers.

Unfortunately, it is very difficult to perform a context translation in the presence of optimizations that reorder instructions. Although we do not yet perform such optimizations, our system does not currently attempt to fix up exception contexts. To solve the context translation problem, we have to keep a mapping from application addresses to code cache addresses for basic blocks and traces, and we must be able to reverse any optimizations we perform. We plan to address this issue in the future. We expected to have problems both with exception contexts and with self-modifying code, but neither have occurred in any of the large Windows programs we have been running.

3.3.4 Other Abnormal Control Flow

Fortunately, `setjmp/longjmp` on Windows do not require any special actions on the part of our system. The unwinding of the application stack and the final setting of the program counter are all performed in user mode. Our system simply sees a normal indirect branch at the end of it all.

Conversely, the Windows API function `SetThreadContext` must be intercepted by our system. We simply change the program counter value that is being set for the target thread to point to our own routine (we save the original value). When the target thread is resumed (`SetThreadContext` may only be called on a suspended thread) the operating system sets its program counter to our routine and we take over from there.

3.3.5 Alternative Solutions

We came up with a few alternative methods for ensuring that all application code runs under our control. We were hoping to find a solution that was not too hardwired to the particular version of Windows we were running on, but we were not successful.

One method is to mark all pages except our code cache and our system's own code as non-executable. Then whenever control switches to any application code, we trap the page fault and redirect control to our system. Unfortunately, IA-32 makes no distinction between read privileges and execute privileges. Thus we do not want to mark all pages non-executable because that makes them non-readable, which is not something we want to do to pages containing data. And it is not possible in general to know all possible pages that might contain code. The inability to identify and separate all pages with potential application code made us reject this solution.

Another solution is to watch for registration of all callback routines and replace the registration with a routine in our own system. However, this requires detailed knowledge of all possible callback routines within the entire Windows API, which is very large. Also, this is again specific to the particular version of Windows being run. We rejected this solution for these reasons.

4 Handling Multiple Threads

There were three major issues we confronted when supporting multiple threads. The first was data structure synchronization, which is relatively simple to deal with by making use of synchronization primitives provided by the operating system. Below we discuss the other two issues, thread-local scratch space and cache management.

4.1 Thread-Local Scratch Space

Thread-local scratch space is needed to perform tasks such as comparing the target of an indirect branch to a constant to see if execution should continue along a trace. Allocating thread-local memory is not difficult — the Windows API supports such memory. The challenge is to provide efficient access to scratch space while in the middle of arbitrary application code. The performance hit of making an API call whenever space is needed is unacceptable. Furthermore, the process of making that call and handling its results requires its own scratch space. We need more immediate access to thread-local memory than an API call.

We came up with four different methods for providing efficient thread-local scratch space. The first is to use the stack. We rejected this solution because it assumes that the stack pointer is always valid. We want our system to be robust and handle hand-coded applications that do not obey software conventions.

The second method is to steal a register and have that register always point to the thread-local storage. This solution was employed in Dynamo [3]. It incurs a performance penalty that on an architecture with few registers like IA-32 might be steep.

The third method is to use the thread-local storage slots that are kept in the operating system's thread information block (TIB), accessible directly by the segment register `fs` [20]. The operating system saves and restores the TIB along with the rest of the machine context when executing callbacks and other control transfers. This solution is only available on Windows. It is not officially documented and could change in future versions of Windows. We have not implemented it in our system.

The fourth and final method is to use thread-private code caches. Since all code is run by only one thread, an absolute address can be used for scratch space (IA-32 allows addressing an absolute address directly). This option is discussed further in Section 4.2.2.

We first implemented the second solution, register stealing. We decided to steal `edi` as it is not used for any special purposes (many IA-32 registers have special meanings to certain instructions) except by the string instructions. To measure the performance penalty of register stealing, we also implemented using an absolute address as scratch space (which assumes a single thread). Figure 3 shows the results on the SPEC2000 benchmarks [24] (which are all single-threaded). The average slowdown is about 1% for unoptimized code and 7% for optimized code. The discrepancy is easily explained: optimized code uses the registers more effectively than unoptimized code and thus "notices" if you steal one of the registers. Note that our register stealing was not extremely sophisticated and that slightly better performance could have been achieved.

After encountering cache management issues that thread-private caches would solve, we decided to use thread-private caches and eliminate our register stealing. The next section discusses cache management in more detail.

4.2 Cache Management

There are two major choices when it comes to the code cache and multiple threads: either have a single thread-shared code cache, or give each thread its own thread-private code cache. We discuss each of these below. For each of these strategies on Windows, care must be taken to not delete undeletable fragments, which we discussed in Section 3.3.

4.2.1 Thread-Shared Code Cache

Multiple threads sharing the same cache complicate the issue of cache management. We would like to be able to delete fragments from the cache to make room for new fragments. However, there is no efficient way to determine if any threads are executing inside a given fragment. The system must force all threads out of the cache whenever it wants to perform cache management. First, all fragments must be unlinked so that there are no loops in the cache.

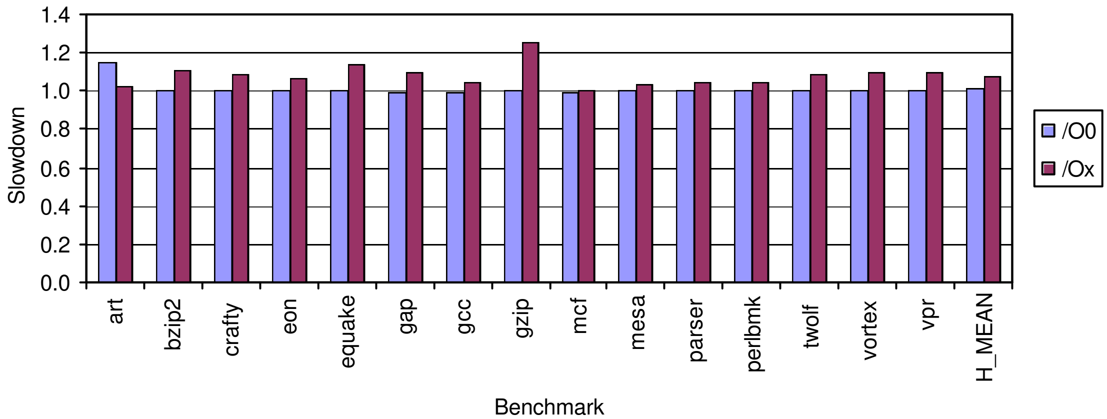

Figure 3: Performance impact of register stealing on the non-FORTRAN SPEC2000 benchmarks. H_MEAN is the harmonic mean. /O0 indicates that the benchmark was compiled unoptimized and /Ox indicates that the benchmark was compiled with full optimizations.

Then the system sets a flag so that no thread can enter the cache, and waits for all threads to exit the cache. Due to this high overhead, many or all fragments must be deleted to reduce the frequency of cache management. Previous systems typically flush the entire cache [3].

Dividing the cache into segments can help. A data structure keeps track of how many threads are in each segment. Threads must check for permission before entering a segment. This adds overhead on all inter-segment links. When room is needed in the cache, the system picks a thread-free segment and prevents other threads from entering while it deletes some or all of the fragments in that segment. However, there is no guarantee that a segment can be found that has no threads in it. When no thread-free segment can be found, the system must force threads out of a segment using the strategy for single-segment cache management.

4.2.2 Thread-Private Code Cache

Thread-private caches have a number of attractive advantages over a single thread-shared cache, such as less synchronization, no register stealing needed for thread-local scratch space (see Section 4.1), and much simpler and more efficient cache management. Their only disadvantage is the space duplication and time to duplicate fragments that are used by multiple threads.

How much code is shared among threads? It depends on the application; in a web server, many threads run the same code. However, in a desktop application, threads typically perform distinct tasks. Studies have shown that nearly all instructions in desktop applications are executed by one thread [16]. We did a study using our system in thread-shared cache mode. We measured the number of fragments that are executed by more than one thread. Note that it does not matter how frequently executed these fragments are. The only performance penalty is the time and space to duplicate the fragments. In fact, once they are duplicated there are more opportunities for optimization because the thread-private fragments can be specialized for their particular thread.

Figure 4 shows the percentage of both basic block fragments and trace fragments that are used by more than one thread. The figure gives results for the four programs described in Section 5. The first set of numbers are for the batch-style scenarios. These scenarios have tiny percentages of shared fragments. More threads are created and potentially more code shared when these desktop applications are used in a more interactive fashion. Figure 4 gives a second set of numbers for our applications when used in more interactive scenarios. For these scenarios the number of threads created by the application increased for `winword` from 3 to 9 and for `powerpnt` from 4 to 5 (the others remained as listed in Table 2). The percentages certainly rose, but only the traces in `powerpnt` rose to a significant percentage. This matches previous results [16], where only `powerpnt` has a significant fraction of instructions executed in any thread other than the primary thread.

We believe that these results validate a thread-private cache strategy. The disadvantages of such a strategy are minimal due to the very small amount of code shared among threads. The numbers in the next section are for our system using thread-private caches.

5 Experimental Results

We used four desktop applications to build a benchmark suite, described in Table 1. These scenarios model long-running batch computations, not highly interactive use. However, it is exactly in these long-running computations that the user notices delays, and it is these scenarios that are most worthwhile targeting with dynamic optimization. More interactive scenarios are hard to measure, mask our slow-

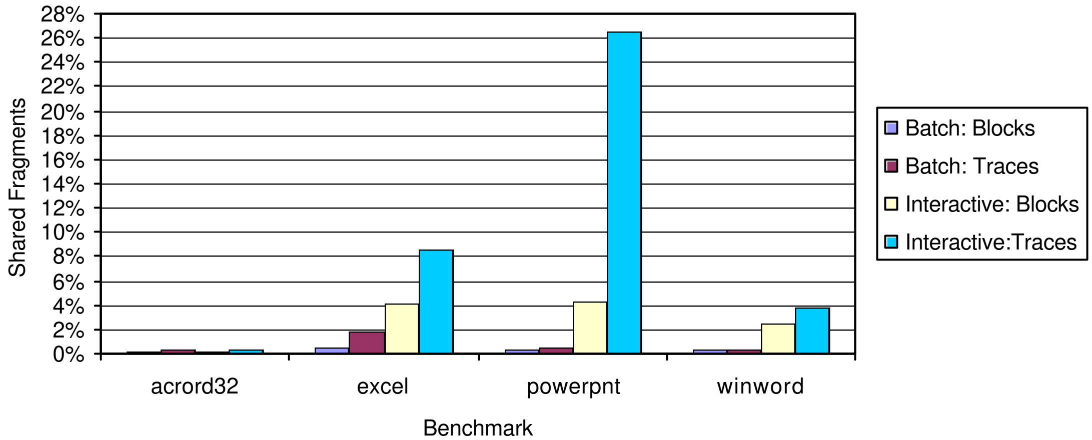

Figure 4: Percentage of both basic block fragments and trace fragments that are used by more than one thread, for two different scenarios: a batch scenario and an interactive scenario. For the interactive scenarios, `winword` increased its threads to 9 and `powerpnt` increased its threads to 5. Otherwise the threads are as listed in Table 2.

Program	Benchmark description
acrord32	Adobe Acrobat Reader 4.0: loads a 6.8MB (956 page) PDF document and searches for the word "bogus" (which is not present).
excel	Microsoft Excel 9.0: loads a 2.4MB spreadsheet full of interdependent formulae and modifies one cell, triggering extensive re-calculations.
powerpnt	Microsoft PowerPoint 9.0: loads an 84-slide (455KB) presentation and adds text to the title of every slide.
winword	Microsoft Word 9.0: loads a 1.6MB document, replaces 'a' with 'o', then "selects all" and changes the font type and size.

Table 1: Descriptions of our desktop benchmarks.

downs, and give fewer opportunities for meaningful performance gains.

Table 2 gives statistics for each of the programs, including application size, number of threads, and the number of basic block and trace fragments our system creates when executing each program. Table 3 lists statistics for the sizes of these basic blocks and traces. Note that these sizes include the *exit stubs* that are added to the end of each fragment and so are larger than the sizes of basic blocks in the original application code.

The final column of Table 2 gives the performance of each benchmark with an unlimited thread-private cache size. For comparison, Figure 5 shows the performance of our system on the SPEC2000 benchmarks [24], also with unlimited cache space.

We used program counter sampling to analyze the time spent in the various portions of our system. For the SPEC2000 benchmarks, from 0% to 2% of execution time is spent in our system code itself (the part of the system on the top half of Figure 1). Essentially all time was spent in the code cache. Of this time, up to 17% is spent in the indirect branch lookup, and up to 7% in the "trace branch taken" instructions. According to these numbers, we should be getting at worst a 26% slowdown, yet we have close to a 160% slowdown for `crafty`. We are in the process of examining our cache and branch prediction behavior to try and solve this problem. We believe that we have run into a difficult challenge of achieving performance on modern processors: the hardware is aggressively tuned toward a certain type of code. Our code cache looks different from the code that the hardware is optimized for. For one thing, we have eliminated calls and turned returns into indirect jumps. Pentium II and later processors have a return stack for predicting targets of returns. Our experiments show that this return stack is critical in providing performance for code with a lot of procedure calls, yet it is useless inside our code cache. We are attempting to address this problem by modifying our handling of calls and returns to make use of the hardware return stack.

The Windows benchmarks have similar results, except that the overhead of our system code is higher, up to 10%. Much of this extra time is spent decoding and encoding IA-32 instructions. We plan to do some significant performance tuning on that part of our system.

Program	.EXE Size	DLLs	Total Size	Threads	Blocks	Traces	Slowdown
acrord32	2.3	22	12.3	2	85366	6541	1.90
excel	7.2	9	16.4	3	79741	3108	1.59
powerpnt	4.3	10	14.1	4	146579	12057	1.66
winword	8.8	13	21.3	3	97072	6505	1.61

Table 2: Statistics for our benchmarks: the static size of the .EXE file (in MB), the number of DLLs loaded, the sum of the sizes of the .EXE and all DLLs loaded by the program (in MB), the number of threads, the number of unique basic blocks executed, the number of traces created, and the slowdown vs. native execution using a thread-private cache of unlimited size.

Program	Basic Block Sizes (Bytes)				Trace Sizes (Bytes)			
	Min	Max	Mean	StdDev	Min	Max	Mean	StdDev
acrord32	27	4034	55.3	30.5	27	23500	179.2	342.1
excel	27	1589	53.1	22.3	30	2971	175.0	181.2
powerpnt	27	2315	53.4	23.4	27	5881	172.1	178.6
winword	27	2821	54.2	31.2	27	2873	160.8	163.0

Table 3: Statistics for the sizes (in bytes) of basic blocks and traces in our benchmarks. These sizes include the exit stubs that are added to the end of each fragment. Each exit stub is 15 bytes (3 5-byte instructions).

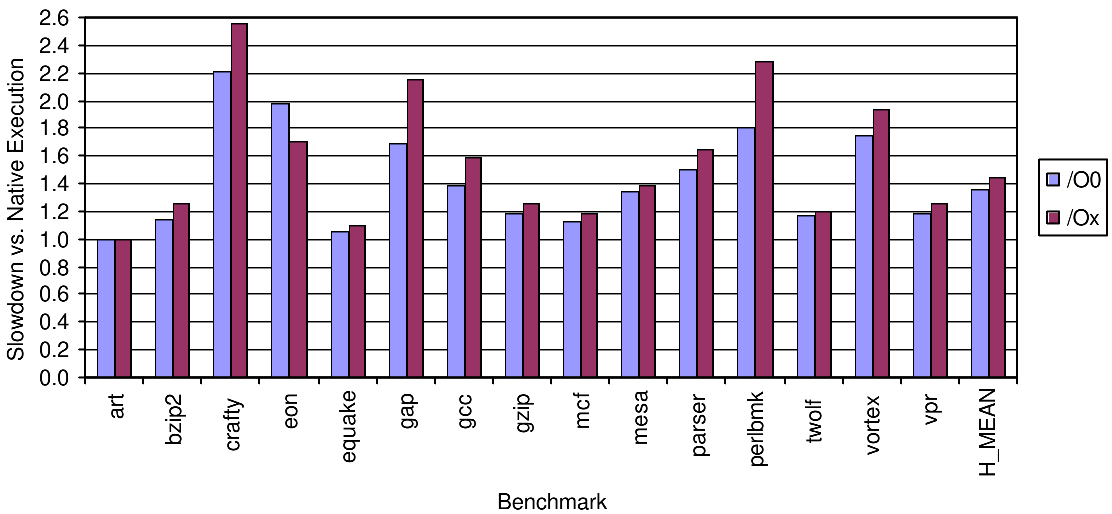

Figure 5: Slowdown of our system versus native execution on all non-FORTRAN SPEC2000 benchmarks [24]. H_MEAN is the harmonic mean. /O0 indicates that the benchmark was compiled unoptimized and /Ox indicates that the benchmark was compiled with full optimizations.

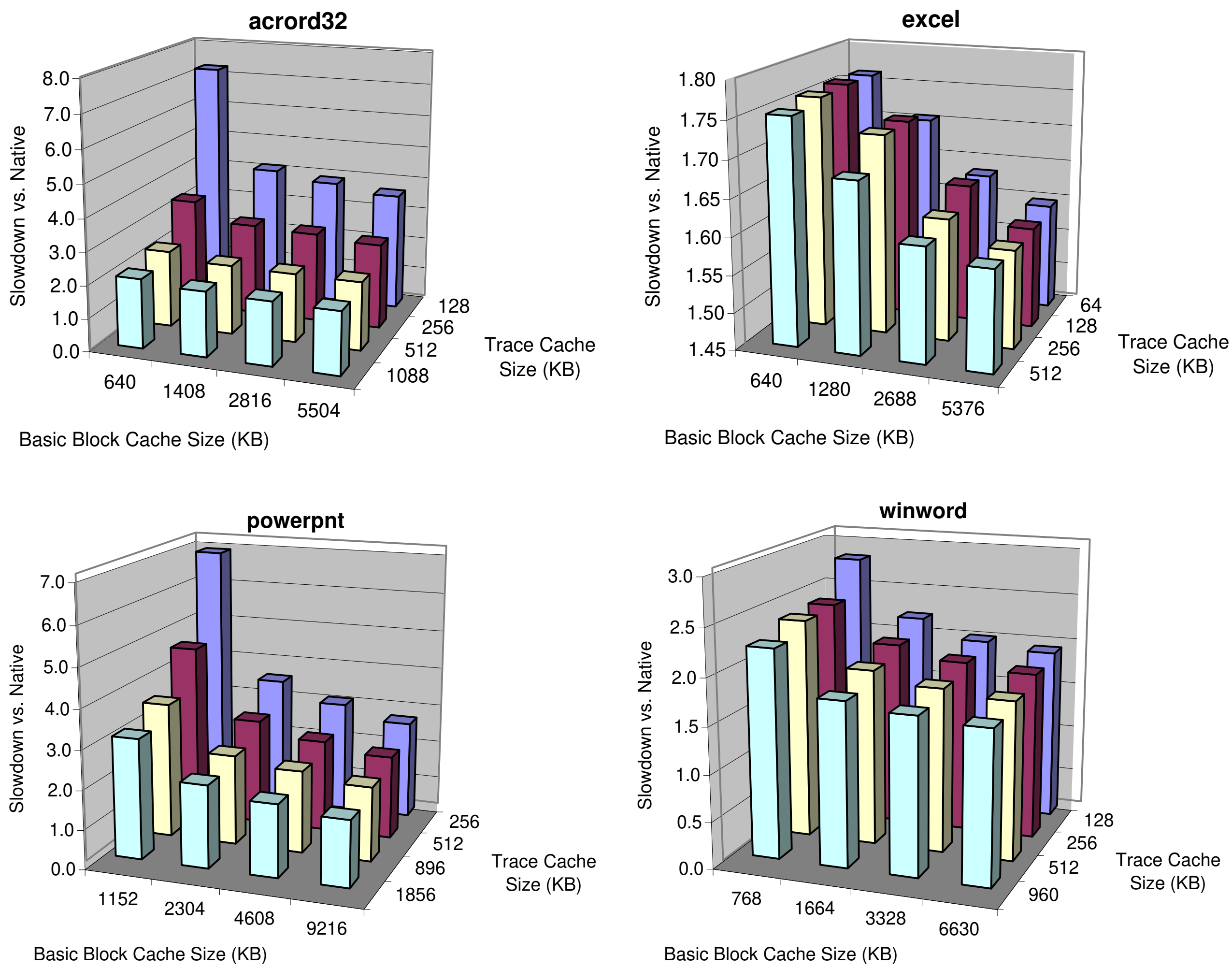

Figure 6: Performance of benchmarks when varying the cache parameters. The x axis shows four values for basic block cache size: the maximum the program would want, and one-half, one-quarter, and one-eighth of that maximum value. The y axis has the same values for the trace cache. The z axis shows the resulting performance as a slowdown versus native execution.

5.1 Cache Parameters

Figure 6 shows the performance of our benchmarks when varying the sizes of the basic block and trace portions of the code cache. These size parameters are upper limits on the sum of the cache sizes for all threads. For each benchmark we took the maximum trace and basic block cache sizes that the program would want. We then set the sizes of each cache to one-eighth, one-quarter, and one-half of the maximum cache size, and the maximum size itself, and plotted the resulting performance versus native execution. The results show that cutting the cache size in half does not affect performance much. Only when both caches are shrunk to fractions of their maximum sizes does performance suffer significantly.

The graphs show that some benchmarks have smaller working set sizes than others. `Excel`, for example, is not affected much by shrinking both of its caches to one-eighth their maximum sizes. `Acrord32`, on the other hand, suffers over a seven times slowdown when the same thing is done to its caches.

The graphs also show that for a copy-paste interpreter such as ours, the basic block cache is nearly as important as the trace cache. We have not yet tuned our trace selection for this system; we expect better and more important traces with further work on the specifics of building them.

We are fairly satisfied with the performance results in this paper. There is a lot of performance tuning to be done on the infrastructure that can bring the overheads down. And we have not even begun to perform optimizations on traces.

6 Related Work

There are a few software dynamic optimization systems in the literature. They include Dynamo [3], which our system is derived from, and Wiggins/Redstone [9], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on. The Mojo [4] system, like ours, targets Windows NT running on IA-32. However, details of Mojo, including thread-local state and interception of Windows control flow, are not sufficiently described to allow implementation by the reader. They were unclear on whether the work could be duplicated without access to the Windows source code. Our work shows that it is possible to build a Windows optimization system without Windows source code access.

Dynamic optimization of the processor's instruction stream is performed in superscalar processors. The Trace Cache [23] allows such optimizations to be performed off of the critical path.

Dynamic translation systems [6, 5, 15, 12] employ techniques similar to ours. These are interpreters that cache the native translations of frequently executed code. They have been built for various domains including emulation of one architecture on another.

Dynamic compilation employs numerous techniques relevant to software dynamic optimization. Dynamic compilation is used both for interpreted languages as *just-in-time compilation* [2, 17, 10, 14] and for compiled languages [13, 8].

Other related fields include link-time optimization [18, 7], which shares with dynamic optimization the fact that it optimizes binary code, and low-overhead profiling [1, 11], which is crucial in a dynamic optimization system for quickly identifying important regions of code.

7 Conclusions

We have described a dynamic optimization framework for the Windows operating system running on the IA-32 architecture. In addition to identifying the major engineering challenges in capturing all program behavior on Windows and handling multiple threads, we have made a case for thread-private caches and shown some initial performance numbers for various cache sizes.

This framework opens up opportunities for program introspection and performance enhancement, as all program information is available at runtime. Our future work includes implementing an instrumentation system using our framework. We also plan to study the applicability of known compiler optimization methods, as well as discover novel techniques for optimization within this framework.

8 Acknowledgements

We would like to thank Vasanth Bala and Mike Smith for their initial development work on our dynamic optimization infrastructure.

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating System Principles (SOSP '97)*, October 1997.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[4] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[5] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), March 1998.

[6] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS*, 1994.

[7] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/NT applications. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, December 1996.

[8] Charles Consel and F. Nöel. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages (POPL '96)*, January 1996.

[9] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, August 1999.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, January 1984.

[11] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), October 2000.

[12] Kemal Ebcioglu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Microarchitecture (ISCA '97)*, June 1997.

[13] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, May 1999.

[14] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994. `http://www.cs.ucsb.edu/oocsb/papers/urs-thesis.html`.

[15] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation, January 2000. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[16] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on Windows NT. In *25th Annual International Symposium on Microarchitecture (ISCA '98)*, June 1998.

[17] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[18] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, January 2001.

[19] Gary Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN, 2000.

[20] Matt Pietrek. Under the hood. *Microsoft Systems Journal*, 11(5), May 1996.

[21] Matt Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, 12(1), January 1997.

[22] Jeffrey Richter. *Programming Applications for Microsoft Windows, Fourth Edition*. Microsoft Press, Redmond, WA, 1999.

[23] Eric Rotenberg, Steve Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, December 1996.

[24] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. `http://www.spec.org/osg/cpu2000/`.