Optimization of Naïve Dynamic Binary Instrumentation Tools

by

Reid Kleckner

S.B., Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 2011

©2011 Massachusetts Institute of Technology
All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
August 29, 2010

Certified by .....
Saman Amarasinghe
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Dennis M. Freeman
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Optimization of Naïve Dynamic Binary Instrumentation Tools

by

Reid Kleckner

Submitted to the
Department of Electrical Engineering and Computer Science

August 29, 2010

In partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The proliferation of dynamic program analysis tools has done much to ease the burden of developing complex software. However, creating such tools remains a challenge. Dynamic binary instrumentation frameworks such as DyanamoRIO and Pin provide support for such tools by taking responsibility for application transparency and machine code manipulation. However, tool writers must still make a tough choice when writing instrumentation: should they inject custom inline assembly into the application code, or should they use the framework facilities for inserting callbacks into regular C code? Custom assembly can be more performant and more flexible, but it forces the tool to take some responsibility for maintaining application transparency. Callbacks into C, or "clean calls," allow the tool writer to ignore the details of maintaining transparency. Generally speaking, a clean call entails switching to a safe stack, saving all registers, materializing the arguments, and jumping to the callback.

This thesis presents a suite of optimizations for DynamoRIO that improves the performance of "naïve tools," or tools which rely primarily on clean calls for instrumentation. Most importantly, we present a novel *partial inlining* optimization for instrumentation routines with conditional analysis. For simpler instrumentation routines, we present a novel *call coalescing* optimization that batches calls into fewer context switches. In addition to these two novel techniques, we provide a suite of machine code optimizations designed to leverage the opportunities created by the aforementioned techniques.

With this additional functionality built on DynamoRIO, we have shown improvements of up to 54.8x for a naïve instruction counting tool as well as a 3.7x performance improvement for a memory alignment checking tool on average for many of the benchmarks from the SPEC 2006 CPU benchmark suite.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Computer Science and Engineering

Contents

List of Figures

Acknowledgements

This thesis was made possible with the help of many individuals. First and foremost, Saman Amarasinghe, my advisor, has provided me with the guidance needed to complete the project, for which I am grateful. I also want to thank him for making 6.035, the MIT undergraduate compilers course, as rigorous and practical as it was. Derek Bruening also deserves much credit for being the driving force behind DynamoRIO. Working closely on DynamoRIO, I have learned more and more about the transparency and performance challenges that it faces and how they have been overcome. Qin Zhao, another contributor to DynamoRIO, has helped me a great deal through conversation about the implementation details of the system. Qin is also responsible for the initial inlining prototype, which I have been working steadily to improve. I have learned something new after every one of our conversations.

On a more personal note, I thank my parents for everything they have given me. To my father, I have always enjoyed our conversations, technical and otherwise. To my mother, I am ultimately thankful for the time you have spent encouraging me to achieve.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Research Objectives

Dynamic program analysis tools built with dynamic binary instrumentation (DBI) have proven indispensable for developing large native applications. Memory debuggers such as DrMemory[1] and Valgrind[2] in particular have been instrumental in tracking down uninitialized memory and memory leaks. Race detectors such as Helgrind[3] and Thread Sanitizer[4] have made programming with shared memory feasible. Cache simulators and branch prediction simulators such as Cachegrind[5] provide a way to optimize cache usage in a given program.

These are just the most commonly used types of instrumentation tools. Researchers in academia and industry are always building new tools, both general-purpose as well as application-specific. For example, a research group at MIT recently created Jolt[6], a tool for detecting and escaping from infinite loops with Pin[7]. In industry, Determina extended DynamoRIO to support program shepherding[8], which detects application security violations.

For the last several years, Pin has been the framework of choice for researchers implementing custom program analysis tools. Pin's success in this area is due to Pin's facilities for abstracting away architectural details from the instrumentation tool. A Pin tool works by inserting calls to instrumentation routines, which record information about the program execution. For example, if a Pin tool wishes to analyze memory usage, it will instrument every memory access to call to a function with information about that memory access. Each call inserted with Pin generally expands to many instructions to spill registers, switch stacks, and materialize arguments, so tools that use pervasive instrumentation can be quite slow. To ameliorate this, Pin is cabable of inlining short routines that have no control flow.

On the other hand, DynamoRIO[9] has a larger and more general interface. Instead of only inserting

calls to plain C or C++ routines, a tool has the power to insert custom machine code into the application code. For example, DrMemory uses the flexibility of DynamoRIO to generate memory accesses that will fault if the application uses uninitialized data. While faulting is expensive, it happens rarely. In the common case that the check succeeds, a memory access is much faster than a conditional branch. As a result, DrMemory is on average twice as fast as Valgrind's Memcheck.[1] Pin does not provide the flexibility needed to generate and handle this kind of instrumentation.

While the ability to insert custom machine code is powerful and can support efficient tools, generating that code can be a daunting task for a researcher or a beginner learning the framework. To help ameliorate the burden, DynamoRIO also has facilities for inserting "clean calls," which are similar to Pin's instrumentation routines. However, DynamoRIO cannot inline clean calls, and is generally not suited to pervasive instrumentation using clean calls.

The goal of this thesis is to make it possible to build performant program anlysis tools with DynamoRIO without burdening the tool writer. We want to make tools that are developed using ordinary clean calls fast enough to be broadly useful. We believe that the slowdowns incurred by clean calls prevent people from using them, and we would like reduce those slowdowns so that these novel tools see wider adoption.

Throughout this thesis, we follow the efforts of a tool author attempting to implement three tools: an instruction counting tool, a memory alignment tool, and a memory access trace tool. To support the author of these tools, we make the following contributions:

- An *inlining* optimization for instrumentation routines.
- The first *partial inlining* optimization for instrumentation routines with conditional analysis.
- The first *call coalescing* optimization for instrumentation routines.
- A suite of x86 machine code optimizations leveraging the opportunities created by inlining.

Partial inlining is a well-understood compiler optimization that attempts to inline the commonly executed code without inlining an entire function, which would cause code bloat. In our case, not all tool code can be inlined into the application code stream. Partial inlining allows us to inline the parts we can and leave out the rest. As far as we know, this is the first application of this technique to dynamic binary instrumentation.

For tools built with pervasive clean calls, the main cost is usually the context switching between the application and DynamoRIO. The idea behind call coalescing is that we should make multiple calls after one context switch so that we need to make fewer context switches over all.

Finally, we found that applying the above techniques was not enough, and that we needed to build a

larger suite of general machine code optimizations to finish cleaning up the inlined code. For example, in our memory alignment tool, the size used for the alignment check is passed as a parameter, and it is always the same at each call site. If we inline without folding this constant into the alignment check, we are clobbering extra registers and issuing extra instructions.

Using the above techniques, we have been able to dramatically improve performance for an instruction counting tool by 54.8 times and a memory alignment tool by almost 4 times.

1.2 Thesis Overview

In Chapter 2, we discuss the motivation for using a dynamic binary instrumentation framework such as DynamoRIO, Pin, or Valgrind in the first place. In particular, we outline all the benefits they provide and the challenges they help overcome. We also take a closer look at the execution model of DynamoRIO because it has a great impact on the design of analysis tools.

In Chapter 3, we start with a naïve implementation of instruction count and walk through the stages of optimizations that we apply. As we go through the stages, the instrumentation code is progressively simplified until it starts to look like the version we would write using custom machine code.

In Chapter 4, we take a look at two more complex tools: a memory alignment checker and a memory trace tool. These tools have the common property that the instrumentation has an early conditional check that results either in a fast or a slow path being taken. In the memory alignment tool, no action needs to be taken if the memory access is aligned. In the memory trace tool, the trace buffer does not need to be flushed if it is not full. We describe how we go about inlining the fast paths while maintaining correctness when the slow path is taken.

In Chapter 5, we depart from our example-driven description of the system to step back and look at the system hierarchy.

In Chapter 6, we present the experimental performance results we achieved on the SPEC2006 CPU integer benchmarks[10] for all three of the tools examined in this thesis.

In Chapter 7.2, we look back on the contributions of this thesis and suggest possible directions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background

2.1 Dynamic Binary Instrumentation Frameworks

In order to understand this thesis, it is important to understand what a dynamic binary instrumentation (DBI) framework provides to a tool writer.

The primary job of a DBI framework is to interpret a native application as it executes and provide an abstract representation of the program code that the tool can analyze and instrument. At first, it would seem easier to simply disassmble the application in question and insert instrumentation code into it statically. However, for modern applications, this approach simply does not work. First, there are dynamically loaded libraries that the application may be linked against. Some of these are possible to identify, such as `libc` or others. Some, however, may be dynamically loaded via other interfaces such as `dlopen` on Linux and `LoadLibrary` on Windows. These are not possible to predict statically, and a static instrumentation tool will not be able to observe and instrument these instructions. Hence, a *dynamic* binary instrumentation framework is needed to run alongside the application and intercept every instruction that the application would have executed were it to run natively. Using dynamic instrumentation instead of static instrumentation, an analysis tool can observe and manipulate *all* instructions executed by the application instead of just some.

Furthermore, a dynamic framework maintains control even in the face of such convoluted techniques as self-modifying code. As techniques such as embedded Just In Time (JIT) compilation become more prevalent, it becomes more important to be able to observe such dynamically generated code. A DBI framework is also responsible for providing all of the native operating system interfaces to the application just as if the application were running natively. This can be a daunting challenge, as the operating system interface is large, and an application can register many points of entry with the operating system such as

signal handlers. A good DBI framework, such as DynamoRIO, Pin, or Valgrind, will intercept all of these requests and ensure that control is maintained and the tool author is able to observe all instructions.

Finally, a DBI framework provides transparency of the framework and the tool to the application. When trying to analyze a program, it can be frustrating when bugs disappear when run under the analysis tool. If the tool and the framework are completely transparent, then the application will not be able to tell that it is running under the tool. Furthermore, the tool will not affect the flow of the application in any way. For example, DynamoRIO provides a heap that is fully isolated from the application, so memory allocations by the tool do not disturb allocations by the application. A separate heap also prevents application bugs from overwriting DynamoRIO's data structures. After heap data structures, we also need to worry about the application's stack, for all of the same reasons. The application may have buffer overruns on the stack, or it might not even have a stack. Therefore, DynamoRIO maintains a separate stack. Applications also occasionally use introspection to enumerate all the threads in a process, query virtual memory protections, or walk the stack, for example. DynamoRIO makes sure that all of these application introspection techniques work as if they were running natively, so the tool author does not have to worry about it.

For all of these reasons, it is highly desirable to build dynamic program analysis tools with DBI frameworks. The goal of this thesis is to make it easier to use DBI frameworks to write analysis tools that perform well. For this thesis, we chose to start by modifying DynamoRIO, which we describe in the following section.

2.2 DynamoRIO's Execution Model

In order to dynamically instrument an application, DynamoRIO performs "process virtualization." All application code is translated into the software code cache, just as is done in VMWare's flagship virtualization product.[11] Essentially, DynamoRIO injects itself between the application and the operating system, as shown in Figure 2-1, instead of between the operating system and the hardware, as VMWare does. All control transfers between the application and the operating system are caught and mediated by DynamoRIO so that it can maintain control. To run an application under DynamoRIO, the application is launched in such a way that DynamoRIO is loaded and given control during process initialization.

When DynamoRIO takes control, it sets up its own execution context, separate from that of the application. DynamoRIO's context consists of a separate stack, and a thread-local data structure describing its state. The application context consists of the original application stack along with all of the registers that DynamoRIO may clobber while executing its own code. Once DynamoRIO has switched to its own stack, it determines what the next application program counter would have been and begins the process of interpretation, entering the basic block builder from Figure 2-1.

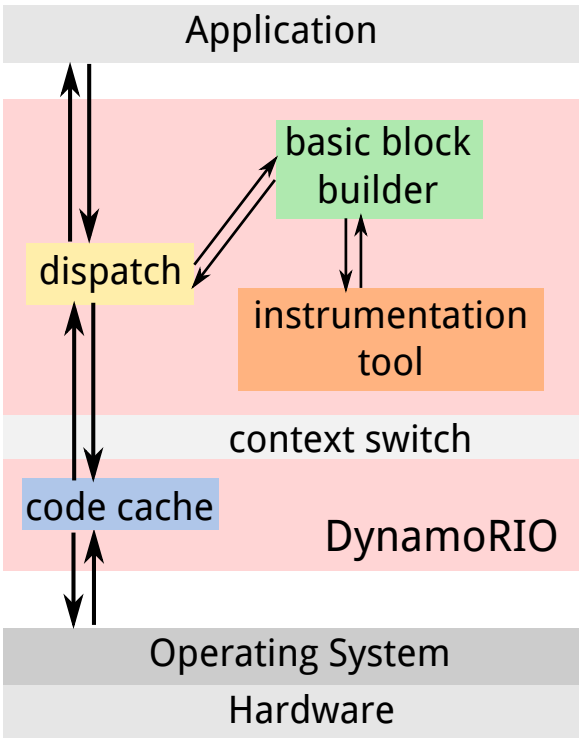

Figure 2-1: High-level diagram of DynamoRIO executing an application.

For a previously unencountered target application PC, the builder decodes instructions from the first instruction until the next control transfer. Before modifying the instructions to run in the code cache, DynamoRIO presents the instructions to the tool for instrumentation, if a tool is loaded. The tool then has the opportunity to analyze the basic block and insert its own instructions or clean calls. Such instructions are marked as "meta," meaning they are not application instructions and should not be modified to maintain control. Inserting custom code into the application is the most efficient way to perform analysis, but most naïve clients will stick to inserting clean calls. Clean calls require a full context switch across the gray border, and significantly bloat code size, requiring extra memory and polluting the cache.

After instrumentation, DynamoRIO mangles the terminating control transfer instruction to maintain control when the basic block exits. Specifically, control flow instructions are mangled so that they jump into DynamoRIO's central dispatch mechanism which will figure out what to do next, which is represented by the arrow leaving the code cache and entering the dispatch block in Figure 2-1.

Once DynamoRIO is done modifying the application instruction stream, the instructions are encoded into a "fragment" in the code cache. The code cache is the memory space allocated by DynamoRIO for translated application code. Finally, DynamoRIO switches back to the application context and starts executing the new fragment.

When the basic block finishes execution, instead of transferring to the original application target, it will re-enter the DynamoRIO VM with information about the original target application program counter. If the target application PC is not in the code cache yet, DynamoRIO will then repeat the process of translation for the next basic block. After translation, it will return to the application context and continue execution from the freshly translated fragment.

Bouncing back and forth between DynamoRIO's central dispatch and the code cache is expensive, so DynamoRIO attempts to "link" basic blocks together. If the control transfer target is direct, the two basic blocks can be linked by modifying the terminating control transfer instruction of the previous fragment to directly target to the beginning of the next fragment. As a result, when a code path executes more than once, it will not have to leave the code cache to look up the next fragment to execute.

Now that we have presented the motivations and challenges involved with dynamic binary instrumentation, we demonstrate how simple inlining and our suite of optimizations work together to optimize a naïve instruction counting tool.

Chapter 3

Inlining

3.1 Instruction Count Tool

The first tool we examine in this thesis is instruction count. The goal of this tool is to compute the exact number of instructions that would be executed by the application if we were to run it natively. The simplest way of achieving this goal is to iterate across each instruction and insert a call to a routine which increments a 64-bit counter before executing the routine. A fragment of the source code for this tool is shown in Figure 3-1.

Instruction count is a simple example, and it would be fair to suggest that in this case a reasonable tool author would be able to optimize this to a single instrumentation routine call in each basic block. However, we believe that for more complicated tools, this is not easy, and inserting many calls to instrumentation routines around the instructions they pertain to is the simple and expected way to write the tool. For example, a tool like `opcodemix`, a sample Pin tool that reports the distribution of x86 instruction opcodes executed, is implemented the same way. The simple way to implement `opcodemix` is to instrument each instruction with a call to a function that takes the instruction opcode as a parameter. The routine then increments the appropriate counter based on the opcode. We expect that all optimizations applying to our sample instruction count tool would be just as effective for `opcodemix` as well as other similar tools.

3.2 Clean Calls

Implemented with the standard clean call infrastructure that DynamoRIO provides, this simple tool creates large fragments of code that perform poorly. Figure 3-2 shows an x86_64 assembly listing of a clean call.

```
#include "dr_api.h"
#include "dr_calls.h"

/* ... */

static void inc_count(void) {
    global_count++;
}

dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    instr_t *instr;
    for (instr = instrlist_first(bb); instr != NULL;
         instr = instr_get_next(instr))
        drcalls_insert_call(drcontext, bb, instr, (void *)inc_count,
                            false /* save fpstate */, 0);
    /* ... */
}
```

Figure 3-1: Abbreviated source code sample for our instruction count tool.

```
mov    %rsp -> %gs:0x00                                  # Switch to dstack
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp                          # Allocate frame
mov    %rax -> 0x48(%rsp)                                # Save GPRs
... save rbx-r14 ...
mov    %r15 -> 0x00000088(%rsp)
lahf   -> %ah
seto   -> %al
mov    %rax -> 0x00000090(%rsp)                          # Save "aflags"
movdqa %xmm0 -> 0x000000b0(%rsp)                         # Save XMM regs
... save xmm1-xmm14 ...
movdqa %xmm15 -> 0x00000290(%rsp)
call   $0x004041c0 %rsp -> %rsp 0xfffffff8(%rsp)         # Call inc_count
movdqa 0x000000b0(%rsp) -> %xmm0                         # Restore XMM regs
... restore xmm1-xmm14 ...
movdqa 0x00000290(%rsp) -> %xmm15
mov    0x00000090(%rsp) -> %rax                          # Restore "aflags"
add    $0x7f %al -> %al
sahf   %ah
mov    0x48(%rsp) -> %rax                                # Restore GPRs
... restore rbx-r14 ...
mov    0x00000088(%rsp) -> %r15
mov    %gs:0x00 -> %rsp                                  # Restore appstack
```

Figure 3-2: Assembly listing for a single clean call.

Although all we want to do is increment a counter, we end up doing all of the following work in order to support as much arbitrary C code as possible:

1. Switch to a clean stack.
2. Save all registers.
3. Save the flags register.
4. Save "volatile" XMM registers.
5. Materialize arguments into parameter registers. Not shown above, since inc_count takes no arguments.
6. Call tool routine.
7. Restore "volatile" XMM registers.
8. Restore the flags register.
9. Restore all registers.
10. Switch back to the application stack.

Saving the XMM registers seems excessive for a short snippet of C code, but even if the tool is avoiding the use of floating point instructions, it is still possible for the compiler to choose to use XMM registers. More likely, however, is that the tool may call `memcpy`. In recent versions of glibc, `memcpy` will make use of the XMM registers to widen the load and store width, which will clobber the caller-saved XMM registers. Therefore DynamoRIO must conservatively save this set of registers, even though they are most likely unused.

For the SPEC bzip2 benchmark, we get terrible performance. Natively, this application runs in 4.5 seconds. With clean calls, the application takes 9 minutes and 20 seconds, giving a 124x slowdown. Through the rest of the chapter we show how to improve on that.

Our primary target is to avoid doing all this extra work of context switching. Our first technique is to *inline* the instrumentation routine into the application code stream.

3.3 Simple Inlining

For inlining routines that have no control flow, the process is simple. The tool provides us with a function pointer which is the PC of the first instruction in the routine. We start decoding instructions from this PC until we find the first `ret` instruction. Assuming the routine has no control flow, this is adequate for reliably decoding the routine. When we extended our inliner to perform partial inlining we had to augment our decoding algorithm as described in Section 4.3.

Inlining is only possible if we are able to analyze the callee and only if the analysis suggests that doing so would be profitable. The analysis is designed so that if there are any corner cases that the analysis cannot handle reliably, it can fail, and inlining will fail. For example, if the stack pointer is dynamically updated in the middle of the function, this inhibits stack usage analysis. Therefore we bail out in such cases. Assuming our analysis is successful, we only inline if the following criteria apply:

- The callee is a leaf function. Our definition of leaf function is conservative, meaning it must not have any calls, trapping instructions, indirect branches, or direct branches outside of the function.
- The simplified callee instruction stream is no longer than 20 instructions. This limit is in place to avoid code bloat. The limit of 20 instructions has not been experimentally tested and was chosen to match Pin's behavior.
- The callee does not use XMM registers.
- The callee does not use more than a fixed number of general purpose registers. We have not picked an appropriate limit yet.
- The callee must have a simple stack frame that uses at most one stack location.
- The callee may only have as many arguments as can be passed in registers on the current platform, or only one if the native calling convention does not support register parameters.

If the routine meets the above criteria, we can then simplify the clean call process described in Section 3.2 down to the following steps:

1. Switch to a clean stack.
2. Save clobbered registers.
3. Save the flags register if used.
4. Materialize arguments into parameter registers.
5. Emit the simplified inline code stream.
6. Restore the flags register if used.
7. Restore clobbered registers.
8. Switch back to the application stack.

In particular, we skip all XMM saves and most of the register saves, especially on x86_64 which has more registers than most callees use.

For our instruction count tool, the disassmbled `inc_count` routine is shown in its original form in Figure 3-3 and after inlining in Figure 3-4. Note how all the stack frame setup and teardown related instructions from the original routine have been removed in the inlined version, and been replaced with our own stack management.

```
push   %rbp %rsp -> %rsp 0xffffff8(%rsp)       # Frame setup
mov    %rsp -> %rbp
mov    <rel> 0x00000000006189e0 -> %rax        # Load global_count
add    $0x0000000000000001 %rax -> %rax        # Increment
mov    %rax -> <rel> 0x00000000006189e0        # Store global_count
leave  %rbp %rsp (%rbp) -> %rsp %rbp           # Frame teardown
ret    %rsp (%rsp) -> %rsp                     # ret
```

Figure 3-3: Assembly for the `inc_count` instrumentation routine.

```
mov    %rsp -> %gs:0x00                        # Switch to dstack
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp                # Create frame
mov    %rax -> 0x48(%rsp)                      # Save %rax, only used reg
lahf   -> %ah                                  # Save aflags
seto   -> %al
mov    %rax -> 0x00000090(%rsp)
mov    <rel> 0x00000000006189e0 -> %rax        # Inlined increment code
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x00000000006189e0
mov    0x00000090(%rsp) -> %rax                # Restore aflags
add    $0x7f %al -> %al
sahf   %ah
mov    0x48(%rsp) -> %rax                      # Restore %rax
mov    %gs:0x00 -> %rsp                        # Back to appstack
mov    (%rbx) -> %rax                          # APPLICATION INSTRUCTION
... Process repeats for each application instruction
```

Figure 3-4: `inc_count` inlined into an application basic block without further optimization.

The inlined version of the code is clearly an improvement over the clean call version of the code, and is 5.8x faster than the clean call version, running in 96.3 seconds. Still the cost of switching contexts to the DynamoRIO stack to save registers dwarfs the cost of the actual code we wish to execute. While there are opportunities for reducing the work done here, we introduce the novel technique of "call coalescing" to reduce the number of switches needed.

3.4 Call Coalescing

In order to do coalescing, we need visibility of the entire instrumented basic block. Therefore we choose to represent calls to instrumentation routines as artifical pseudo-instructions that are expanded after instrumentation. Before expanding each call, we see if it is possible to group them together within the basic block. If the call has register arguments, we avoid moving it outside the live range of its register arguments or past any memory writes. This approach works well for instruction count and other tools that pass immediate values to instrumentation routines, because we have total freedom as to how they are scheduled.

Once the calls are scheduled, we expand them to another level of pseudo-ops of stack switches and register saves and register restores. Before lowering these pseudo-ops, we pass over the instruction stream to delete consecutive reciprocal operations. Currently we identify three pairs of operations as being reciprocal: switching to dstack and back, restoring and saving a register, and restoring and saving flags. Coalescing all calls in a three instruction basic block results in the code in Figure 3-5.

By coalescing calls together we achieve a 2.1x improvement over standard inlining, bringing our execution time down to 45.6 seconds. However, we still spend an entire four instructions setting up a stack frame just so that we can save `%rax` and the flags register. In the next section, we describe how we can avoid switching stacks altogether.

3.5 Avoiding Stack Switching with TLS

The original reason for switching stacks is that we cannot rely on being able to use the stack used by the application. There are many reasons why this might be the case. The application might be using the red zone on Linux x86_64, which is an area past the end of the stack usable by leaf functions. The application might be close to the guard page which will grow the stack. The application might have a dangling pointer to the stack and we may want to find that bug. Finally, the application might be running without a stack and just using `%rsp` as a general purpose register. In any case, it is important to assume nothing about the application stack.

Our main use for the stack when inlining code is as temporary scratch space where we can save

```
mov    %rsp -> %gs:0x00                          # Switch to dstack
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp                  # Setup frame
mov    %rax -> 0x48(%rsp)                        # Save %rax
lahf   -> %ah                                    # Save flags
seto   -> %al
mov    %rax -> 0x00000090(%rsp)
mov    <rel> 0x0000000000618c80 -> %rax          # Call 1
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000618c80
mov    <rel> 0x0000000000618c80 -> %rax          # Call 2
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000618c80
mov    <rel> 0x0000000000618c80 -> %rax          # Call 3
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000618c80
mov    0x00000090(%rsp) -> %rax                  # Restore flags
add    $0x7f %al -> %al
sahf   %ah
mov    0x48(%rsp) -> %rax                        # Restore %rax
mov    %gs:0x00 -> %rsp                          # Restore appstack
... three application instructions
```

Figure 3-5: Three inlined calls to `inc_count` coalesced together.

registers that are clobbered. For single-threaded applications we can simply use global variables, but this breaks down quickly if we want to share the code cache between threads. In order to have per-thread global variables, we need to use "thread local storage," or TLS. On Windows and Linux the x86 `fs` and `gs` segment registers are used to point to a memory region unique for each thread. On Linux, only one segment is used, so we claim the other and use it for our own TLS memory region. On Windows, however, the operating system will not preserve our segment register value across interrupts, so we cannot set our own. Instead, we steal TLS space from the application by introspecting on the application's thread execution block (TEB) and marking our storage as allocated.[12] DynamoRIO uses a small amount of this space internally, and we cannot afford to allocate too much or we will interfere with the execution of the application. Therefore we do not enable this optimization by default. Finally, if we wish to perform partial inlining as described in Chapter 4, we cannot yet transfer from the inline code to the slowpath without an existing stack frame.

Figure 3-6 shows the instruction count example code from the previous section with the TLS scratch space optimization turned on. Note that it does not need any stack frame setup, and simply saves registers and flags into TLS slots.

Using TLS space on this example only gains us about a 1% performance improvement. When we initially implemented this optimization, we obtained much better results, because we implemented TLS

```
mov    %rax -> %gs:0x00000088                    # Save %rax
lahf   -> %ah                                    # Save flags
seto   -> %al
mov    %rax -> %gs:0x00000090
mov    <rel> 0x0000000000619780 -> %rax          # Call 1
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000619780
mov    <rel> 0x0000000000619780 -> %rax          # Call 2
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000619780
mov    <rel> 0x0000000000619780 -> %rax          # Call 3
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000619780
mov    %gs:0x00000090 -> %rax                    # Restore flags
add    $0x7f %al -> %al
sahf   %ah
mov    %gs:0x00000088 -> %rax                    # Restore %rax
... three application instructions
```

Figure 3-6: Three inlined and coalesced calls to `inc_count` that use TLS instead of dstack.

usage before call coalescing. After coalescing, there are fewer stack switches to worry about, so we are only able to delete 3 frame setup instructions.

To address the next source of inefficiency in our instruction counting tool, we use some classic compiler optimizations on the inlined code, now that the register spilling has been sufficiently optimized.

3.6 Redundant Load Elimination and Dead Store Elimination

Redundant load elimination (RLE) and dead store elimination (DSE) are classic compiler optimizations that we found useful for optimizing tools. In our last example, the inlined code is loading `global_count`, incrementing it, and storing it three separate times. Rather than having three separate loads and stores, we would like to have one load at the beginning and one store at the end.

The logic for our version of RLE is that if there is a store to and a load from the same memory operand without any potentially aliasing memory writes between them, we can use the source register from the store instead of loading the value again. We have to be careful that the register that was written to the memory location is still live by the time the load occurs, and that it is live for all uses of the load's destination register. If that is the case, then we rewrite all uses of the load's destination register over its live range to use the store's source register. Figure 3-7 shows the example code fragment after applying RLE.

Dead store elimination (DSE) is similar, in that if there is a store to memory, followed by no potentially aliasing reads from memory, followed by a store to the same memory location, we can delete the first

```
mov    %rax -> %gs:0x00000088                  # Save %rax
lahf   -> %ah                                  # Save flags
seto   -> %al
mov    %rax -> %gs:0x00000090
mov    <rel> 0x0000000000619780 -> %rax        # Load 1
add    $0x0000000000000001 %rax -> %rax
mov    %rax -> <rel> 0x0000000000619780        # Store 1
add    $0x0000000000000001 %rax -> %rax        # Load 2 is gone
mov    %rax -> <rel> 0x0000000000619780        # Store 2
add    $0x0000000000000001 %rax -> %rax        # Load 3 is gone
mov    %rax -> <rel> 0x0000000000619780        # Store 3
mov    %gs:0x00000090 -> %rax                  # Restore flags
add    $0x7f %al -> %al
sahf   %ah
mov    %gs:0x00000088 -> %rax                  # Restore %rax
... three application instructions
```

Figure 3-7: Three inlined calls to `inc_count` after applying just RLE.

```
mov    %rax -> %gs:0x00000088                  # Save %rax
lahf   -> %ah                                  # Save flags
seto   -> %al
mov    %rax -> %gs:0x00000090
mov    <rel> 0x0000000000619780 -> %rax        # Load 1
add    $0x0000000000000001 %rax -> %rax        # Increment 1
add    $0x0000000000000001 %rax -> %rax        # Increment 2
add    $0x0000000000000001 %rax -> %rax        # Increment 3
mov    %rax -> <rel> 0x0000000000619780        # Store 3
mov    %gs:0x00000090 -> %rax                  # Restore flags
add    $0x7f %al -> %al
sahf   %ah
mov    %gs:0x00000088 -> %rax                  # Restore %rax
... three application instructions
```

Figure 3-8: Three inlined calls to `inc_count` after applying RLE and DSE.

store. Because we do not need to rewrite any uses of any registers, this is simpler than RLE. Figure 3-8 shows the application of both RLE and DSE.

After applying both optimizations, we get a satisfactory 2.1x speedup, bringing the bzip2 execution time much closer to the native execution time at 21.1 seconds. Still, our code is not satisfactory. We are using the `add` instruction, which updates the carry, overflow, and other bits in the x86 flags register. We cannot assume that the application is not using these flag bits, so we must preserve them, requiring an extra stack slot and some extra flag manipulation instructions. The next section describes how to avoid this.

```
mov    %rax -> %gs:0x00000088                    # Save %rax
mov    <rel> 0x0000000000619780 -> %rax          # Load global_count
lea    0x01(%rax) -> %rax                        # No flags usage
lea    0x01(%rax) -> %rax                        # No flags usage
lea    0x01(%rax) -> %rax                        # No flags usage
mov    %rax -> <rel> 0x0000000000619780          # Store global_count
mov    %gs:0x00000088 -> %rax
... three application instructions
```

Figure 3-9: Three inlined calls to `inc_count` after avoiding aflags usage.

3.7 Flags Avoidance

Many x86 arithmetic instructions modify the x86 flags register. This is problematic, because it means we have to save and restore the bits of the flags register that we touch. This can be done either using the `pushf` and `popf` instructions, or a sequence of `lahf`, `seto`, `lahf`, and `add`. The `popd` instruction turns out to be fairly expensive, because it sets many complex execution flag like the single-stepping trap flag for debuggers. Also, the `lahf` sequence avoids stack usage, so it is preferred.

However, we have the opportunity to transform inlined code to avoid flags usage if possible. Instead of using the `add` instruction to perform additions and subtractions, we can use the `lea` instruction. The `lea` instruction was intended to be a "load effective address" instruction for dealing with things like the complex x86 segmented address space model. For its source, we provide an arbitrary x86 memory operand, for which it computes the effective address and stores the result in a destination register. However, we can provide a memory operand which is performing simple additions instead of address arithmetic. Because `lea` is not designed for arithmetic, it does not modify flags.

We have a pass which can automatically perform this transformation. However, because `lea` does not support memory operands as destinations, we have to introduce a temporary register along with a load and store if the arithmetic operation uses a memory operand as a destination. Our pass attemps to pick a register which has already been used by the inlined code so that we do not need to introduce any extra saves and restores.

We show the results of applying this pass to our repeated additions in Figure 3-9. As shown in the listing, this is a big win, because it eliminates three instructions on both sides of the call site as well as two memory accesses.

After applying our flags avoidance optimization, we obtain a 1.4x speedup over our previous time. Still, it is obvious to a human that this is not as efficient as it could be. We do not need three separate increment instructions. They can be folded together to a single `lea` that adds 3 to `%rax`.

```
mov    %rax -> %gs:0x00000088                    # Save %rax
mov    <rel> 0x0000000000619780 -> %rax          # Load global_count
lea    0x03(%rax) -> %rax                        # Add 3
mov    %rax -> <rel> 0x0000000000619780          # Store global_count
mov    %gs:0x00000088 -> %rax                    # Restore %rax
... three application instructions
```

Figure 3-10: Three inlined calls to `inc_count` after avoiding aflags usage.

3.8 Folding `lea` Instructions

Using some of the same register liveness analysis tools we developed to support our RLE and DSE optimizations, we can do the same register rewriting for `lea` operations. If one `lea` defines a register used in another `lea`, we can attempt to combine the two memory operands. For simple base and displacement memory operands, as in the listing shown in the previous section, this is as simple as adding the two displacements together and substituting in the original base register. We are also able to handle more complex cases involving two memory operands both of which have a base, displacement, index, and scale. After we apply our instruction folding optimizations to the example at hand, we get the assembly listing in Figure 3-10.

Applying our `lea` folding optimization gives us a 1.5x speedup over our previous best, bringing our execution time down to 9.8 seconds. The application's native execution speed is 4.5 seconds on our hardware, and the unoptimized instrumented execution time was 9 minutes and 20 seconds. After applying all of our optimizations we are able to improve the instrumented execution time 57 fold, obtaining only a 2.1x slowdown from native execution.

3.9 Chapter Summary

Using the example of instruction counting, we have shown how our inliner and suite of supporting optimizations can drastically improve the performance of naïve instrumentation tools. First, we started by looking at how a basic clean call is implemented and at how it does more work than is necessary. We moved on to do basic inlining, where we switch stacks, save only the registers and flags necessary to execute the routine inline, and execute it. Using the suite of optimizations developed during this thesis, we were able to incrementally improve on the code generated in the first example. For our first optimization, using pseudo-instructions, we were able to schedule multiple calls together and avoid duplicated work. Next we avoided the stack switch altogether by using thread local storage, which may not always be available. Regardless, we then moved on to look at how redundant load elimination and dead store elimination can make a big difference in cleaning up our inlined code. Finally, we finished by looking at the low-level machine specific optimizations for avoiding aflags usage and combining multiple arithmetic

instructions.

In the next chapter, we look at two new examples of a memory alignment checker and a memory trace tool too demonstrate the major contribution of this thesis: partial inlining.

Chapter 4

Partial Inlining

In this chapter we explain our partial inlining technique for speeding up analysis routines that have a common fast path and an uncommon slower path. Many analysis tools are structured in this exact way. To demonstrate how partial inlining works, we consider two example tools in this chapter: a memory alignment checker, and a memory trace tool.

4.1 Tools with Fastpaths

The memory alignment checker is a tool that checks if a memory access is aligned to the size of the access and diagnoses accesses that are unaligned by reporting information about the access. For the alignment checker, it is fair to assume that for most programs most memory accesses are unaligned, unless there is an alignment issue in the program. In the case where the access is aligned, the analysis routine does no work. This represents the fast path through the routine. If an access is unaligned, the tool needs to make other function calls to either record the access data in memory or write the information out to a file. This represents the slow path. If many accesses are unaligned, the tool will be slow anyway because it must do work to diagnose every unaligned access. The source code for the alignment instrumentation routine is shown in Figure 4-1.

The memory trace tool has a similar property. The goal of the `memtrace` tool is to make a trace of all the memory accesses in the program and output it to a file for offline analysis. This includes the application PC, the effective address, whether the access was a read or a write, and how large it was. Rather than formatting a string to write to a file after every access, it is faster to fill a buffer with this information and flush it every time it fills up. If the buffer is not full, the analysis routine does little work. If the buffer is full, it needs to perform IO, which will require making system calls. The source code for the memory trace instrumentation routine is shown in Figure 4-2.

```
void
check_access(ptr_uint_t ea, app_pc pc, uint size, bool write)
{
    /* Check alignment.  Assumes size is a power of two. */
    if ((ea & (size - 1)) == 0) {
        return;
    }
    dr_fprintf(STDOUT,
               "Unaligned %s access to ea "PFX" at pc "PFX" of size %d\n",
               (write ? "write" : "read"), ea, pc, size);
}
```

Figure 4-1: Source for the instrumentation routine from the `alignment` tool.

```
typedef struct _memop_t {
    ptr_uint_t ea;
    ptr_uint_t pc;
    uint size;
    uint write;
} memop_t;

#define BUFFER_SIZE 1024
static uint pos;
static memop_t buffer[BUFFER_SIZE];
void
buffer_memop(ptr_uint_t ea, ptr_uint_t pc, uint size, bool write)
{
    buffer[pos].ea = ea;
    buffer[pos].pc = pc;
    buffer[pos].size = size;
    buffer[pos].write = write;
    pos++;
    if (pos >= BUFFER_SIZE)
        flush_buffer();
}
```

Figure 4-2: Source fragment for the `memtrace` tool.

We believe that this pattern of having fast, early checks in analysis routines is common. For example, an undergraduate researcher in our group has been working on a race detector for native applications using the FastTrack algorithm[13]. In the implementation, there is a notion of epochs defined by synchronization events. Using shadow memory provided by Umbra[14], the initial version of this tool checked if the memory in question had been accessed during the last epoch. If so, it has already been checked for races. It turned out that for most applications memory was reused in the same epoch at least 50% of the time, creating the fast path/slow path structure that we believe is common. In later versions of the race detector, the researcher generated custom instrumentation to implement the check inline before jumping to an out of line clean call. Essentially, the researcher was performing manual partial inlining, which is what we are attempting to do automatically.

Before we dive into partial inlining, we need to cover how ordinary clean call instrumentation works with complex argument values, as those are used throughout our examples.

4.2 Clean Calls with Arguments

We have already discussed the major points about how clean calls work in Section 3.2. However, our previous instruction counting example did not need to pass arguments to the instrumentation routine. Our alignment checker and memory trace tool do.

Arguments to instrumentation calls in DynamoRIO are specified as x86 instruction operands evaluated in the application context. We support calling functions with as many arguments as can be passed in registers using the native platform calling convention. However, we can only materialize arguments after we have saved the register used to hold the argument, or we will clobber it. This creates a chicken and egg problem, because the argument may, for example, reference the stack pointer, which is different once we switch to DynamoRIO's execution context. Because we cannot materialize arguments before the stack switch or register spills, there is no safe point in the inlined code to simply evaluate the argument operand as it is. Instead we have interpret it and materialize it from the saved application context.

For immediate values, there is no complication. We simply use a `mov` instruction to materialize the value in the argument register.

For register operands, if they are not clobbered by the routine, they will not have been saved to the stack. Therefore, the application value is still live, so we perform a simple register-to-register copy. If the value is used in the inline code, we make sure to generate a register spill before argument materialization, so we load it back from our scratch space.

Another issue is that as we materialize arguments into registers, we may clobber registers that we wish to use in later arguments. For example, on Linux x86_64, the first argument is materialized into

`%rdi`, but the second argument may read `%rdi`. Our solution is to only materialize arguments that end up being used by the body inline. By following this rule, we can only write registers that are saved in the context switch, and we can rematerialize the application value of `%rdi` from our scratch space.

Finally, if the register is `%rsp`, we have a special case. The application stack pointer is always saved in the TLS scratch space instead of the stack, so we must restore it from there, as shown later in Figure 4-3.

For memory operands, we support two operations: we can either load the application value from the memory addressed by the operand using a `mov` instruction, or we can load the effective address of the memory operand with a `lea` instruction, giving us a pointer to the application value.

X86 memory operands complicate matters because they may use two registers for the base and index. Similar to the register operand case, we need to need to materialize the registers used by the operand before we can materialize the argument. We cannot use the original registers of the operand, because they may now contain materialized arguments. Therefore, we load the base into the register we are using to hold the argument. We know the argument register will be dead after the load or address computation, so this is safe. We then modify the base register of the memory operand to be the argument register. If the index has been clobbered, then we need a second dead register. We hard code the choice of `%rax` for holding the index register because there are no calling conventions that DynamoRIO supports that use `%rax` as a register parameter, and it is often already saved because it must be used to save flags. If it is not already saved, it will be marked for saving later when we re-analyze the code sequence with argument materialization.

For a complete example, if we instrument the load instruction `movq 0x44(%rsp, %rdi, 4) -> %rax` with an unoptimized clean call, we emit the code in Figure 4-3.

As in our instruction count example, using clean calls is clearly not the most efficient way to implement this. However, our new routine code has control flow, so before we can do any inlining, we need to find a reliable way to decode it.

4.3 Decoding with Control Flow

When a tool inserts a clean call, all it provides is a list of machine operand arguments and a function pointer to call. Given a function pointer, it is challenging to reliably decode the function itself and determine where it ends.

The naïve approach described in Section 3.3 of scanning forward from the function entry to the first `ret` instruction breaks down quickly if we want to handle control flow. Partial inlining requires this, so our decoding algorithm has to handle control flow past the first `ret` instruction. Even our simple

```
mov    %rsp -> %gs:0x00                 # Switch to dstack
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp
mov    %rax -> 0x48(%rsp)               # Save GPRs
... save other GPRs
mov    %rdi -> 0x10(%rsp)               # Save rdi at offset 0x10
... save other GPRs and flags
... save all XMM regs
                                        # Argument materialization:
mov    %gs:0x00 -> %rdi                 # Load app rsp into rdi
mov    0x10(%rsp) -> %rax               # Load app rdi into rax
lea    0x44(%rdi,%rax,4) -> %rdi        # Load effective address into rdi
mov    $0x00000000006187c0 -> %rsi      # App PC of instruction
mov    $0x00000008 -> %edx              # Size of access
mov    $0x00000000 -> %ecx              # Not a write
call   $0x00404670 %rsp -> %rsp 0xfffffff8(%rsp)  # Call routine
... clean call restore code
```

Figure 4-3: Materializing `0x44(%rsp, %rdi, 4)` into `rdi`.

`check_access` instrumentation routine jumps past the first ret, as shown in Figure 4-4.

Our decoding algorithm simply remembers the furthest forward branch target, and decodes up until at least that address. Once we have passed that address, we continue decoding until we reach a `ret`, a backwards branch, or a probable tail call. Our heuristic for identifying tail calls is to check if the target address is not between the entry address and the entry plus 4096. Our choice of cutoff is arbitrary and has not been tuned, but it is unlikely that a single instrumentation routine will be larger than 4096 bytes.

After decoding, we rewrite all the branches to use labels inserted in the instruction list instead of raw PC values. This allows us to follow a branch to a later point in the stream during optimization.

With the instruction list in hand, we proceed to partial inlining.

4.4 Inlining the Fast Path

Once we have the full routine instruction stream, we scan from the entry point to the first control flow instruction. In the assembly listing of Figure 4-4, it is the first `jz` instruction at point +13 bytes. If it is a conditional branch, as it is in the example, we scan forward from both branch targets: the fall-through target and the branch taken target. If the first control flow instruction we hit on either path is a `ret` instruction, we say that path is a "fast path." If both paths are fast, we abort our partial inlining analysis and inline both as normal. If neither path is fast, we also abort partial inlining, and most likely will fail to inline the routine at all.

If one path is fast and the other is not, we designate the other path the "slow path." We then delete all

```
TAG  0x0000000000404670
 +0    mov    %edx -> %r8d                      # Copy size to r8
 +3    mov    %rdi -> %r10
 +6    lea    0xffffffff(%r8) -> %eax           # (size - 1)
 +10   test   %rax %rdi                         # ea & (size - 1)
 +13   jz     $0x00000000004046b0               # First branch for if
 +15   mov    <rel> 0x0000000000618764 -> %edi
 +21   test   %cl %cl
 +23   mov    $0x004133e6 -> %edx
 +28   mov    $0x004133eb -> %eax
 +33   cmovnz %rax %rdx -> %rdx
 +37   cmp    %edi $0xffffffff
 +40   jz     $0x00000000004046b8               # Branch past ret
 +42   mov    %r8d -> %r9d                      # Merge point
 +45   mov    %r10 -> %rcx
 +48   mov    %rsi -> %r8
 +51   xor    %eax %eax -> %eax
 +53   mov    $0x00413470 -> %esi
 +58   jmp    $0x00000000004039e8               # Tail call to dr_fprintf.
 +63   nop
 +64   ret    %rsp (%rsp) -> %rsp               # First branch target
 +66   data16 nop    0x00(%rax,%rax,1)
 +72   mov    <rel> 0x0000000000618770 -> %rax  # Target of second branch
 +79   mov    0x70(%rax) -> %edi
 +82   jmp    $0x000000000040469a               # Backwards jump to +42
END 0x0000000000404670
```

Figure 4-4: Assembly for the `check_access` instrumentation routine from our alignment checker.

```
mov    %edx -> %r8d                    # Entry block
mov    %rdi -> %r10
lea    0xffffffff(%r8) -> %eax
test   %rax %rdi
jz     $0x00000000004046b0             # Inlined check
call   <label>                         # Slowpath transition
jmp    <label>                         # Jump to ret past fastpath code
                                       # Fastpath code,
                                       # none in this example
ret    %rsp (%rsp) -> %rsp             # Return
```

Figure 4-5: Assembly for the `check_access` instrumentation routine from our alignment checker.

instructions that do not flow directly from the entry to the conditional branch and then to the fast path. In place of the slow path, we insert a synthetic call instruction which we will expand later as described in Section 4.5, and a jump to the `ret` instruction on the fast path. The result of this transformation on the `check_access` code is shown in Figure 4-5.

This code fragment is much more amenable to inlining, but it is not complete. We still need to generate code to handle the slow path case out of line.

4.5 Slowpath Transition

To enter the slow path, we abandon the inline execution and restart the routine from its entry point with fresh arguments. The alternative we considered was to attempt a side entry into the routine at the original slow path target PC, but this approach would require carefully recreating the stack frame state to match what it would have been if the routine had been executed out of line. Rather than create such an analysis, we chose to simply execute the entry block again. Unfortunately, if there are side effects in the entry block, executing them twice may break the tool. For this section, we restrict ourselves to working on routines without side effects in the entry block and address this problem later in Section 4.6.

Before we can call the tool routine entry point, we need to set up the stack as though we had just finished the clean call entry sequence. By the time we reach the slow path, the entry block leading up to the check will have already clobbered many application registers, so we cannot blindly save all registers. Instead, we save all application registers that were not saved before the inlined code. This means emitting extra saves and restores for general purpose registers, the flags register, and all of the XMM registers.

We do not emit all of the save and restore code inline, because it would pollute the instruction cache with many uncommonly executed instructions. On Linux x86_64, a single clean call expands to at least 75 instructions and 542 bytes. Therefore, we emit the slow path transition code in a code cache maintained on the side. When we reach the slow path, we jump into this code fragment, and it handles the rest of the transition before returning. Because we have already switched stacks, this is easily implemented with

the `call` and `ret` instructions.

Even though we materialize the arguments to the function before the inlined code, it is likely that they will be clobbered in the inlined code. Instead of trying to preserve the original values, we rematerialize them in the slow path. While it is possible to share the register spilling code, the arguments to the routine are usually different at every call site. Therefore we rematerialize the arguments before we jump into the shared slow path transition code. It is possible to detect if the original materialized arguments are still live and do not need to be rematerialized, but we have not yet implemented this optimization because the slow path is unlikely to be executed in the first place.

One major consideration in our design is that DynamoRIO has a function called `dr_get_mcontext` that allows the tool to inspect all application registers saved at the base of the stack. Even if this function is called from the slow path of a partially inlined routine, we want it to behave correctly. This means the stack has to be set up in exactly the same way it would have if we had executed a clean call, which means spilling all application registers into the right stack slots.

4.6 Deferring Side Effects

As discussed previously, there is a problem with re-executing the entry block of the instrumentation routine. If there are any side effects in the entry block, they will occur twice. For example, if we wish to count every memory access in the entry block, as we might want to do with our alignment checker, we would end up counting it twice when we have an unaligned access. We solve this problem by deferring such side effects until after the fast path check.

To identify instructions with side effects, we test each instruction in the entry block to see if it writes to non-stack memory. A non-stack write is any instruction writing to absolute addresses, to PC relative addresses, or to addresses relative to any register other than the stack pointer.

Next, we attempt to defer the side effects until after the conditional branch. The conditional branch may depend on memory reads which depend on memory writes that we want to defer, so we may not be able to defer all side effects, in which case we fail to perform partial inlining. Our algorithm starts at the conditional branch and goes backwards towards the entry point while deferring all instructions that can be moved safely past the branch. An instruction can be moved past the branch if:

1. Its result is unused by the conditional branch, meaning our analysis starting from the branch considers it dead.
2. It does not use any registers clobbered by the branch or its dependent instructions, which are the instructions that could not be deferred.

```
BEFORE DEFER:
mov    <rel> 0x000000007221b7a0 -> %r8d              # Cannot defer
lea    <rel> 0x000000007221b7c0 -> %r9
mov    %r8d -> %eax                                  # Cannot defer
add    $0x00000001 %r8d -> %r8d                      # Cannot defer
lea    (%rax,%rax,2) -> %rax
cmp    %r8d $0x000003ff                              # Cannot defer
lea    0x00(,%rax,8) -> %r10
mov    %rdi -> (%r9,%rax,8)                          # Write
mov    %rsi -> 0x08(%r10,%r9,1)                      # Write
lea    <rel> 0x000000007221b7d0 -> %rsi
mov    %edx -> (%rsi,%rax,8)                         # Write
mov    %ecx -> 0x04(%r10,%rsi,1)                     # Write
mov    %r8d -> <rel> 0x000000007221b7a0              # Write
jbe    $0x0000000041341560                           # Cond branch
ret    %rsp (%rsp) -> %rsp

AFTER DEFER:
mov    <rel> 0x000000007221b7a0 -> %r8d
mov    %r8d -> %eax
add    $0x00000001 %r8d -> %r8d
cmp    %r8d $0x000003ff
jbe    $0x0000000041341560                           # Cond branch
lea    <rel> 0x000000007221b7c0 -> %r9
lea    (%rax,%rax,2) -> %rax
lea    0x00(,%rax,8) -> %r10
mov    %rdi -> (%r9,%rax,8)                          # Write
mov    %rsi -> 0x08(%r10,%r9,1)                      # Write
lea    <rel> 0x000000007221b7d0 -> %rsi
mov    %edx -> (%rsi,%rax,8)                         # Write
mov    %ecx -> 0x04(%r10,%rsi,1)                     # Write
mov    %r8d -> <rel> 0x000000007221b7a0              # Write
ret    %rsp (%rsp) -> %rsp
```

Figure 4-6: Memory trace buffer filling routine before and after deferring side effects.

Figure 4-6 shows the instruction stream of our memory trace tool routine before and after deferring side effects.

If we are unable to defer side effects until after the branch, we abandon partial inlining and leave the instruction stream unmodified.

4.7 Optimization Opportunities

Putting together all of the techniques discussed so far, we are able to successfully perform partial inlining, but there is still work to be done. Figure 4-7 shows what the assembly would look like if we stopped here with our partial inlining techniques. As shown in the listing, there are many opportunities for improvement. For example, we materialize the size argument into `%edx`, which is then copied to `%r8d`.

`%r8d` is decremented and used in the `test` instruction. Ideally, we can propagate the copy, propagate the constant through the decrement, and propagate the resulting value into the `test` instruction. We also have dead code clobbering registers, such as the left over copy from `%rdi` to `%r10` in the fast path. Furthermore, the benefits of cleaning up this code is that it will use fewer registers, which will save a load and store for ever register avoided. In the next sections, we demonstrate how our optimizations iteratively improve the code.

4.8 Dead Code Elimination

The basis for most of our transformations is register liveness analysis. This is a standard dataflow analysis, starting at the end of the instruction stream, and stepping backwards, maintaining a set of live and dead registers based on which registers instructions read from. We also model the liveness of various bits of the x86 flags register, so we can know if a `cmp` or `add` instruction is needed by a branch or `addc` instruction.

With our register liveness analysis, we can use that information to delete dead instructions. Dead instructions arise often when performing partial inlining, because code that may have used a register previously may now be deleted, rendering the instruction dead. We have to apply the usual precautions of not deleting instructions which have purposes besides using the result, such as control flow instructions, memory writes, and labels.

An example of an instruction deleted by our dead code elimination pass is in Figure 4-7 after the fastpath label with the "Dead" comment next to it.

4.9 Copy Propagation

Again, from liveness information, we can detect when it is profitable to eliminate unnecessary copies. If the source register is dead and the destination of a copy is live, we can simply replace all uses of the destination register over its live range with the source register, so long as the source is not clobbered before all uses of the destination register.

These copies are often introduced by the compiler when the slow path needs access to a value before it is modified. Since we delete the slow path, the original value is unused, and the copy is an extra step we do not need.

There is an example of such a copy in Figure 4-7 in the inline code sequence next to the "Extra copy" comment. The result of propagating the copy is shown in Figure 4-8.

```
mov    %rsp -> %gs:0x00
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp
mov    %rax -> 0x48(%rsp)
mov    %rdx -> 0x38(%rsp)
mov    %rdi -> 0x10(%rsp)
mov    %r8 -> 0x50(%rsp)
mov    %r10 -> 0x60(%rsp)
lahf   -> %ah
seto   -> %al
mov    %rax -> 0x00000090(%rsp)
mov    %gs:0x00 -> %rdi                # Argument materialization
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
mov    $0x00000008 -> %edx             # Constant
mov    %edx -> %r8d                    # Extra copy
lea    0xffffffff(%r8) -> %eax         # Decrement
test   %rax %rdi
jz     fastpath
# Slowpath start
mov    %rcx -> 0x40(%rsp)              # Extra register saves
mov    %rsi -> 0x18(%rsp)
mov    %gs:0x00 -> %rdi                # Argument rematerialization
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
mov    $0x00000000006187c0 -> %rsi
mov    $0x00000008 -> %edx
mov    $0x00000000 -> %ecx
call   $0x0000000075d18040 %rsp -> %rsp 0xfffffff8(%rsp)
mov    0x40(%rsp) -> %rcx              # Extra register restores
mov    0x18(%rsp) -> %rsi
jmp    mergepoint
fastpath:
mov    %rdi -> %r10                    # Dead
mergepoint:
mov    0x00000090(%rsp) -> %rax
add    $0x7f %al -> %al
sahf   %ah
mov    0x48(%rsp) -> %rax
mov    0x38(%rsp) -> %rdx
mov    0x10(%rsp) -> %rdi
mov    0x50(%rsp) -> %r8
mov    0x60(%rsp) -> %r10
mov    %gs:0x00 -> %rsp
mov    0x44(%rsp,%rdi,4) -> %rax       # Application instruction
```

Figure 4-7: Partially inlined `check_access` routine without optimizations.

```
mov    %gs:0x00 -> %rdi                # Argument materialization
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
mov    $0x00000008 -> %edx             # Constant
lea    0xffffffff(%rdx) -> %eax        # Decrement
test   %rax %rdi                       # Test
jz     fastpath
```

Figure 4-8: Example with copy removed.

```
mov    %gs:0x00 -> %rdi                # Argument materialization
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
test   %rdi $0x00000007                # Test with an immediate
jz     fastpath
```

Figure 4-9: Check from `check_access` after constants and `lea` instructions have been folded.

4.10 Constant Folding

Perhaps the most classic optimization that compilers perform is constant folding. We fold constants at the machine code level by searching for instructions that materialize immediate values into registers. When we find such a materialization, we check if we can rewrite the uses of the register to use an immediate instead. We also have logic for folding an immediate into an x86 memory operand displacement.

In Figure 4-8, we can combine the `mov $0x08, %edx` instruction with the `lea 0xffffffff(%r8), %edx` instruction into `lea 0x7, %edx`, which is effectively another immediate value materialization. Our `lea` folding optimization described in Section 3.8 is able to fold the displacement-only `lea` into the `test` instruction. The result is shown in Figure 4-9.

Although we have only deleted a few register-to-register arithmetic operations in the inline code, we have completely eliminated the use of the `%edx` register and avoided an extra register spill.

4.11 Chapter Summary

As shown in this chapter, partial inlining can be quite profitable for conditional analysis routines such as those present in our alignment checker and memory trace examples. Partial inlining is fairly tricky, and requires special attention to decoding in the face of control flow, identifying the fast path, transitioning to the slow path, and avoiding repeated side effects.

However, just these techniques are not enough to make partial inlining truly profitable. As shown in Figure 4-7, the inserted code can be quite large. In order to further optimize the inlined code, we needed to develop a suite of more traditional compiler optimizations to take advantage of the opportunities that

inlining creates. In particular, we found that dead code elimination, copy propagation, constant folding, and `lea` folding were particularly effective.

The final result of these combined optimizations on the alignment checker is shown in Figure 4-10. The techniques presented in this chapter are promising, but unlike the previous chapter, we have not been able to approach native performance in our example. Even after applying all of our optimizations, we still have 20 instructions executed on the fast path for every memory access in the program. Clever tools that instrument all memory accesses such as DrMemory have many techniques that we should automate as future work.

In the Chapter 6, we look at the actual performance achieved with these optimizations on the SPEC 2006 CPU benchmarks.

```
mov    %rsp -> %gs:0x00                 # Stack switch
mov    %gs:0x20 -> %rsp
mov    0x000002c0(%rsp) -> %rsp
lea    0xfffffd50(%rsp) -> %rsp
mov    %rax -> 0x48(%rsp)               # Only two register saves needed
mov    %rdi -> 0x10(%rsp)
lahf   -> %ah                           # Flags save
seto   -> %al
mov    %rax -> 0x00000090(%rsp)
mov    %gs:0x00 -> %rdi                 # Arguments and entry check
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
test   %rdi $0x0000000000000007
jz     done
# Slowpath start, this code is not executed commonly
mov    %rcx -> 0x40(%rsp)               # Extra register saves
mov    %rdx -> 0x38(%rsp)
mov    %rsi -> 0x18(%rsp)
mov    %r8 -> 0x50(%rsp)
mov    %gs:0x00 -> %rdi
mov    0x10(%rsp) -> %rax
lea    0x44(%rdi,%rax,4) -> %rdi
mov    $0x00000000006187c0 -> %rsi
mov    $0x00000008 -> %edx
mov    $0x00000000 -> %ecx
call   $0x0000000077b0c040 %rsp -> %rsp 0xfffffff8(%rsp)
mov    0x40(%rsp) -> %rcx
mov    0x38(%rsp) -> %rdx
mov    0x18(%rsp) -> %rsi
mov    0x50(%rsp) -> %r8
jmp    done
# Slowpath end.
done:
mov    0x00000090(%rsp) -> %rax         # Flags restore
add    $0x7f %al -> %al
sahf   %ah
mov    0x48(%rsp) -> %rax               # Register restore
mov    0x10(%rsp) -> %rdi
mov    %gs:0x00 -> %rsp                 # Swich to appstack
mov    0x44(%rsp,%rdi,4) -> %rax        # Application instruction
```

Figure 4-10: Alignment tool routine after partial inlining and optimization.

.

Chapter 5

System Overview

After walking through the examples from the previous chapters, we now describe the system in its final form, in order to look at how the components fit together.

5.1 Call Site Insertion

To use our system, the tool author makes calls to insert calls as they would a normal clean call. At this point, we have the following information: a function pointer to call, the number of arguments, and the arguments for this particular call site. We expect that in a given tool there will a small number of routines which are called many times. Therefore, we take the function pointer and number of arguments, which are the only things we can reasonably assume will be constant, and analyze the routine. Analysis includes decoding the routine, analyzing stack usage, and optimizing it, and is covered later in Section 5.2. After we get the analysis results, we save them, along with the rest of the information for inserting this call, into a pseudo-instruction representing the entire call. We use the pseudo-instruction approach to make call coalescing easier, which is described later. At this point, we return back to the tool, where it performs further analysis and instrumentation.

5.2 Callee Analysis

For every routine that we wish to call, we need perform analysis to decide if it can be inlined or partially inlined. First, we need to decode the routine. In the absence of debug info or any symbols at all, we need to use our own heuristics to try to find the extent of the function. Our algorithm is to decode one instruction at a time and remember the furthest forward branch, conditional or unconditional, within the next 4096 bytes of instructions. If it falls outside that range, we consider it a tail call. After passing the

furthest forward branch, we continue decoding until the next return, backwards branch, or tail call.

Once we have decoded the routine, we analyze its usage of the stack. In particular, we want to find and remove frame setup code that will not be inlined. In general, we try to match frame setup and tear-down instructions together in order to remove them. For functions with multiple exit points, we need to find a matching tear-down instruction on each exit path for each setup instruction. Furthermore, we need to consider high-level instructions, like `leave` and `enter`, that implement multiple steps of frame setup or tear-down.

5.3 Partial Inlining

Next we consider the routine for partial inlining. As described in Section 4.4, we check if the first control transfer instruction after the entry point is a conditional branch. If so, we scan forward from both the fallthrough instruction and the branch taken target, looking for a `ret` instruction. If one path has a `ret` and the other does not, it becomes the fast path and we apply our partial inlining transformation. First, we delete all instructions except those in the entry block and the fast path block. Next, we insert a synthetic call in the slow path block that we expand later. We cannot expand it at the moment, because we are doing call site independent analysis and do not have access to the function arguments, which would need to be rematerialized.

Because the slow path will eventually re-enter the routine from the beginning, we need to defer all side-effects from the entry block into the fast path block, as described in Section 4.6. An instruction has side-effects if it has any non-stack memory write. We are careful not to move any instruction in such a way that its input registers are clobbered, and defer non-side effect instructions to preserve this property.

5.4 Optimization

Next we run our suite of machine code optimizations to try to clean up the code. Optimization at this stage is particularly important if we have applied partial inlining, because we may have deleted uses of values in the entry block in the slow path. It also reduces the number of registers used and deletes instructions, meaning we are more likely to meet our criteria for inlining. We apply the following optimizations in order:

1. Dead code elimination
2. Copy propagation
3. Dead register reuse
4. Constant folding

5. Flags avoidance
6. `lea` folding
7. Redundant load elimination
8. Dead store elimination
9. Dead code elimination
10. Remove jumps to next instruction

This sequence was tested to work well on the example tools we optimized.

All of these optimizations have been discussed in previous chapters, except for removing jumps to following instructions. This situation occurs in partial inlining cases where the fast path has no instructions, as in the alignment example. The slow path ends with a jump to the restore code after the fast path, but if the fast path is empty, the jump is not needed.

5.5 Inlining Criteria

At this point, we have simplified the routine instruction list as much as possible, and it is time to make our decision about whether we can inline at all. To decide, we use the following criteria:

- The callee is a leaf function. A non-leaf function requires saving all registers.
- The simplified callee instruction stream is no more than 20 instructions. This avoids code bloat from overly aggressive inlining. This limit was chosen to match roughly the point at which the overhead of a clean call no longer dominates the cost of a call, so using a full clean call has little penalty.
- The callee does not use XMM registers. This avoids XMM saves.
- The callee does not use more than a fixed number of general purpose registers. We have not picked an appropriate limit yet.
- The callee must have a simple stack frame that uses at most one stack location.
- The callee may only have as many arguments as can be passed in registers on the current platform, or only one if the native calling convention does not support register parameters.

If any of these criteria are not satisfied, we throw away our simplified instruction list and mark the routine as not suitable for inlining. The summary of all of this analysis is stored in a cache, so on future calls to the same routine we will know immediately if the routine can be inlined or not.

5.6 Basic Block Optimization

After all of the instrumentation calls have been inserted by the tool, the tool is required to call our system one last time before returning the list of instructions back to DynamoRIO. At this point, we apply optimizations to the entire basic block, which is how we were able to vastly improve performance on our instruction count example.

At this point, all calls in the instruction stream are a single pseudo-instruction, so they are easy to move around. Our current simple scheduling heuristic moves calls together, so long as they have only immediate integer arguments. If they read application register or memory values, we do not want to move the instrumentation outside of the live range of that register.

Once calls have been scheduled together, we expand them one layer. First, the application state saving operations are inserted, which are represented with pseudo-instructions. Next, the simplified routine code is inserted, which is described further in Section 5.7. Last, the restore code is inserted, again represented as pseudo-instructions.

Because the save and restore operations are high-level pseudo-instructions, they are easy to identify and match with other instructions with reciprocal operations. For example, if we switch to the application stack and then back to DynamoRIO's stack, those two operations cancel each other out and we can delete them both. If we restore and then save flags, those are reciprocal and can be deleted. Similarly, we can avoid restoring and then saving a register. When this is done, if the calls were scheduled together, there should be one save sequence followed by multiple inlined calls followed by one restore sequence.

Last, we run one more optimization sequence over the inlined code. As shown in the instruction count example, RLE and DSE were very effective for identifying extra loads and stores to the global count value. Folding `lea` instructions is also beneficial in that example.

5.7 Call Site Expansion

As discussed in Section 5.6, each call is expanded after being scheduled. At this point, we have the simplified routine code from the shared routine analysis cache. However, we need to materialize arguments, which are different at each call site.

Materializing immediate values is trivial, but anything that reads an application register value is fairly complicated. Our rule is that if we saved an application register, we should reload it, because we might have clobbered it during argument materialization or flags saving. We have to special case the stack pointer, because we save it in a TLS slot.

For memory operand arguments, we have to be extremely careful. We cannot use a single load instruction to re-materialize the argument, and we have limited available registers. We solve this by realizing that there are at least two available registers on all platforms: the destination register itself, and `%rax`, which is never used as a parameter register on any platform. We restore the base register into the argument register and the index register into `%rax`, and rewrite the memory operand to use these two registers. If either or both of the original application registers are unclobbered, we leave them untouched.

Oftentimes there are a few immediate values as arguments, which can be folded further. The alignment checker is a good example of this, because it passes the access size parameter which is combined with the address to perform the check. By folding constants and `lea` instructions one more time, we are able to fold that immediate value into the `test` instruction in Figure 4-9.

Finally, after our optimization step, we perform our register usage analysis to emit save and restore code around the code we want to inline. We save this analysis until after argument materialization and optimization in order to spill as few registers as possible.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Performance

To measure the performance of our instrumentation optimizations, we ran the SPEC2006 CPU integer benchmark suite under our example instrumentation tools. In particular, we focused on an instruction counting tool, a memory alignment checker, and a memory trace tool. Each tool exercises different aspects of our optimizations. Instruction count, for example, is a very simple tool which is amenable to inlining and coalescing. The alignment tool has a simple check before diagnosing an unaligned access, and is amenable to partial inlining. The memory trace tool checks if the buffer is full before inserting, and is also amenable to partial inlining.

Due to the large slowdowns we wish to measure in unoptimized tools, we only ran the benchmarks on the test input size instead of the reference input size. However, this makes some benchmarks run in under one second, so we removed any benchmark that completed in less than two seconds natively. This removed the `xalancbmk`, `libquantum`, `omnetpp`, and `gcc` benchmarks. We were unable to run the `perl` benchmark natively at all, so we removed it as well.

The measurements were taken on a single machine from a set of machines donated by Intel to the MIT Computer Science and AI Lab (CSAIL). The machine uses a 12-core Intel Xeon X5650 CPU running at 2.67 GHz. The machine has 48 GB of RAM, and 12 MB of cache per core. We disabled hyperthreading to avoid the effects of sharing caches between threads on the same physical core. All benchmarks were performed on CSAIL's version of Debian, which uses 64-bit Linux 2.6.32. We used the system compiler, which is Debian GCC 4.3.4.

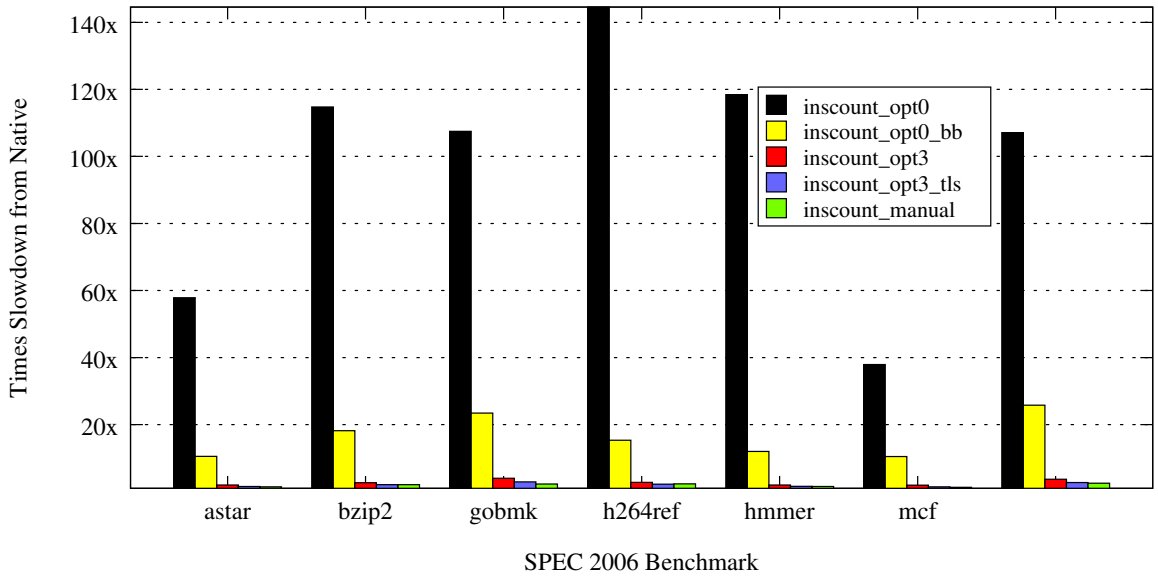

Figure 6-1: Instruction count performance at various optimization levels.

6.1 Instruction Count

Figure 6-1 shows our performance across benchmarks from the suite for instruction count at various levels of optimization. In order to see the performance at higher optimization levels, we have redrawn the graph in Figure 6-2 without the `inscount_opt0` and `inscount_opt0_bb` data. The optimization level refers to how aggressive our inlining optimizations are and does *not* indicate the optimization level with which the tool was compiled. An optimization level of zero means that all calls to instrumentation routines are treated as standard clean calls, requiring a full context switch.

The first configuration, `inscount_opt0`, is a version of instruction count that inserts a clean call at every instruction to increment a 64-bit counter. This configuration represents the most naïve tool writer possible, who is not sensitive to performance, and simply wants to write a tool.

The second configuration, `inscount_opt0_bb`, represents a more reasonable tool writer, who counts the number of instructions in a basic block, and passes them as an immediate integer parameter to a clean call which increments the counter by that amount. This configuration is also run at optimization level zero, so all calls are unoptimized and fully expanded. This configuration is representative of a tool writer who does not wish to generate custom assembly, but is taking steps to not leave easy performance gains on the table.

The third configuration, `inscount_opt3`, is the same as the first configuration with all optimizations enabled. Our chart shows a dramatic improvement, but we are still performing an extra stack switch to do a very small amount of work. The following configuration, `inscount_opt3_tls`, is again the same,

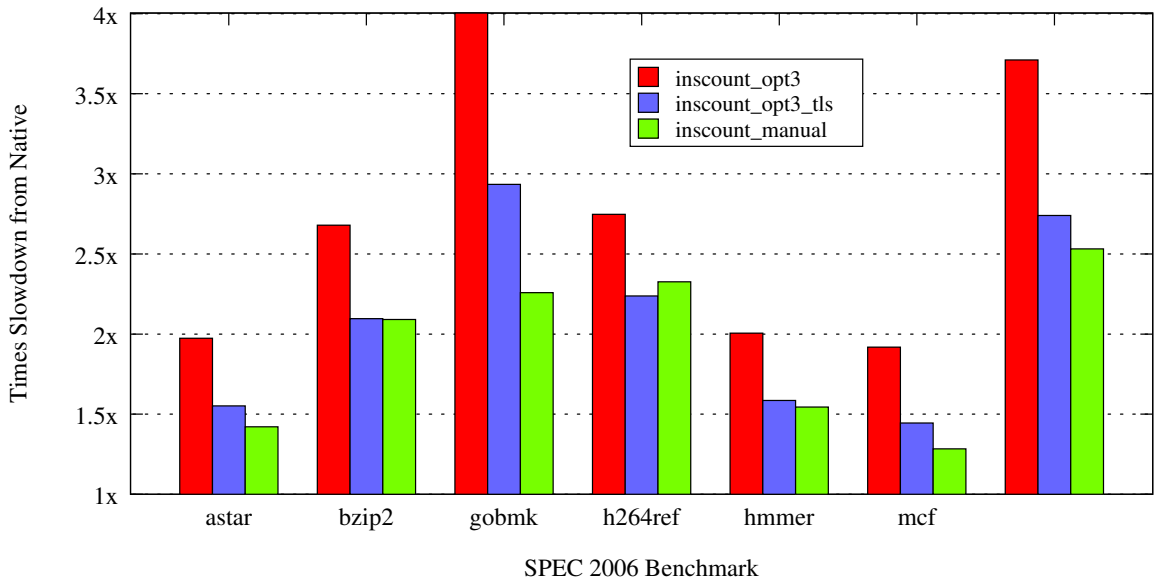

Figure 6-2: Instruction count performance of the optimized configurations for easier comparison.

except instead of switching stacks, thread-local scratch space is used, as discussed in Section 3.5.

The final configuration, `inscount_manual`, is the same tool written using custom machine code instrumentation. This is what we would expect a clever, performance-conscious tool author to write. As shown in Figure 6-2, `inscount_opt3_tls` is quite comparable to `inscount_manual`, meaning that for this tool, we almost reach the performance of custom instrumentation. On average, the automatically optimized instruction count tool is achieving 2.0 times slowdown from native, while the manual instrumentation achieves on average 1.9 times slowdown.

Finally, we show the speedup that optimization produces over the naïve tool in Figure 6-3. On average, `inscount_opt3_tls` is 47.6 times faster than `inscount_opt0` and 7.8 times faster than `inscount_opt0_bb`.

6.2 Alignment

Our alignment tool benchmarks are mainly showcasing the performance improvements from using partial inlining. The instrumentation routine starts by checking that the access is aligned, and if it is not, it issues a diagnostic. Issuing a diagnostic is a complicated operation, and for the SPEC benchmarks, happens infrequently. Therefore, we inline the common case, which does no more work than an alignment check and updating a counter of all memory accesses, and leave the diagnostic code out of line. As shown in Figure 6-4, on average we have a 43.8x slowdown before applying partial inlining, and a 11.9x slowdown after turning on our optimizations. This means we are achieving on average a 3.7x speedup with our partial inlining optimizations. Speedup is broken down by benchmark in Figure 6-5.

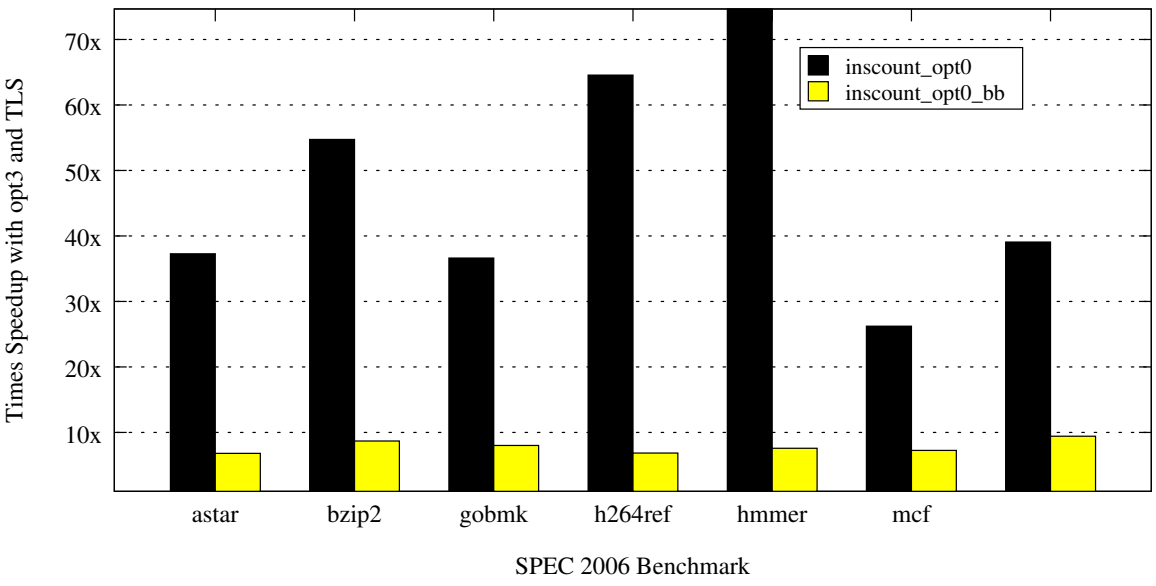

Figure 6-3: Speedup over `inscount_opt0` and `inscount_opt0_bb` after enabling all optimizations.

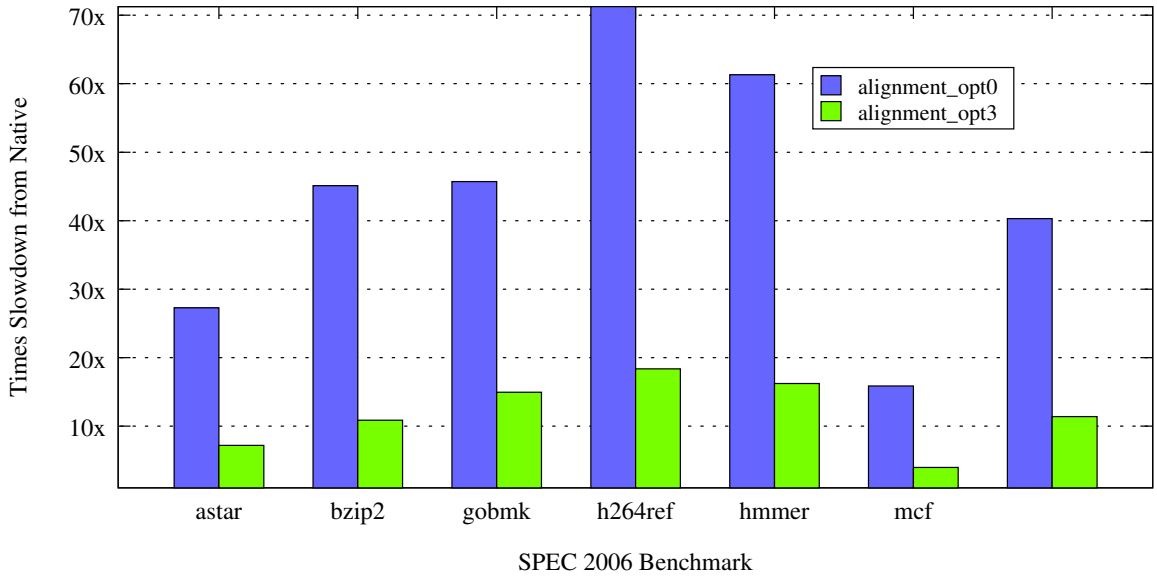

Figure 6-4: Memory alignment tool slowdown from native execution.

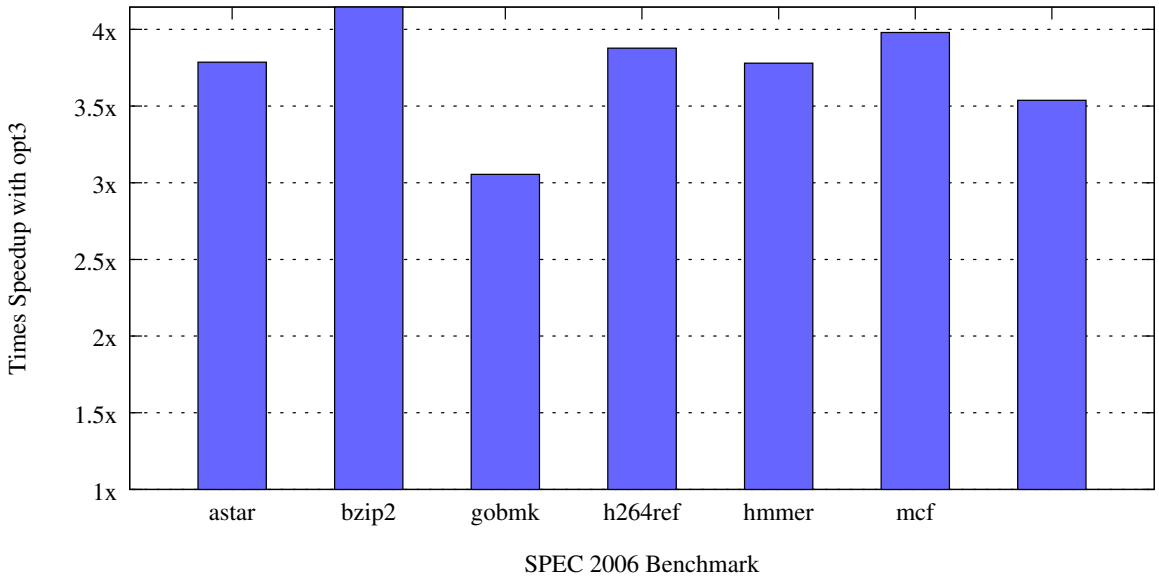

Figure 6-5: Memory alignment tool speedup when optimizations are enabled.

6.3 Memory Trace

The memory trace tool fills a buffer with information about all the memory accesses in a program. Specifically, it tracks the effective address of the access, the program counter of the instruction, the size of the access, and whether the access was a read or a write. All information is written to the last free element of the buffer, and a check is performed to determine if the buffer is full. The buffer is 1024 elements in size, meaning the buffer needs to be processed very infrequently, making this a suitable case for partial inlining. In Figure 6-6, we show the slowdowns from native execution speed with and without optimizations. In Figure 6-7, we show the speedup achieved with turning on optimizations. On average, the tool has a 53x slowdown without optimizations, and a 27.4x slowdown with optimizations. This represents a 1.9x speedup when turning on optimizations.

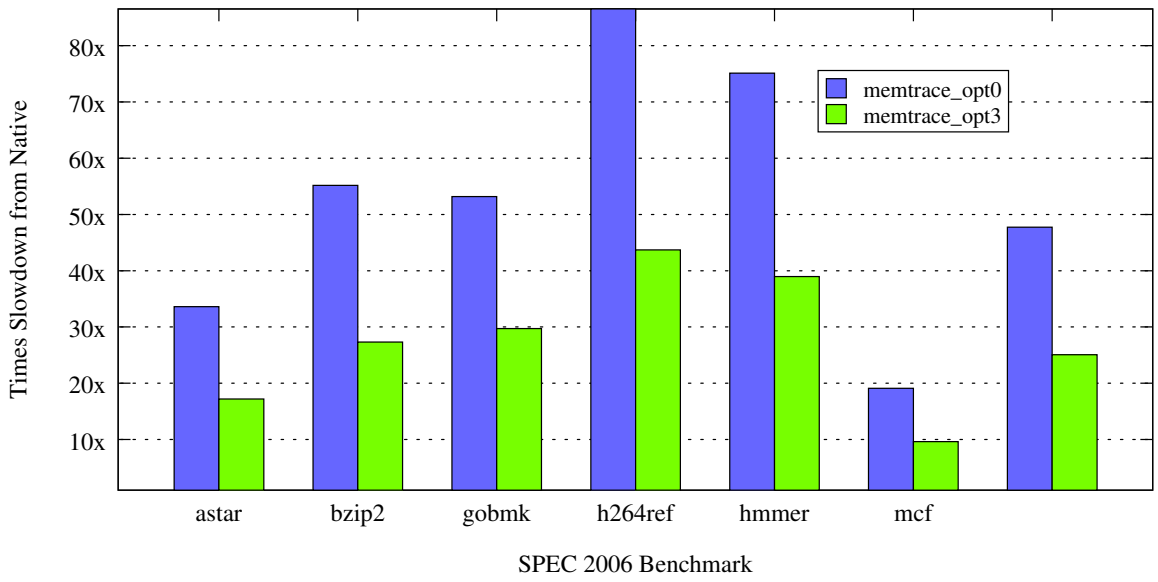

Figure 6-6: Memory trace tool slowdown from native execution.

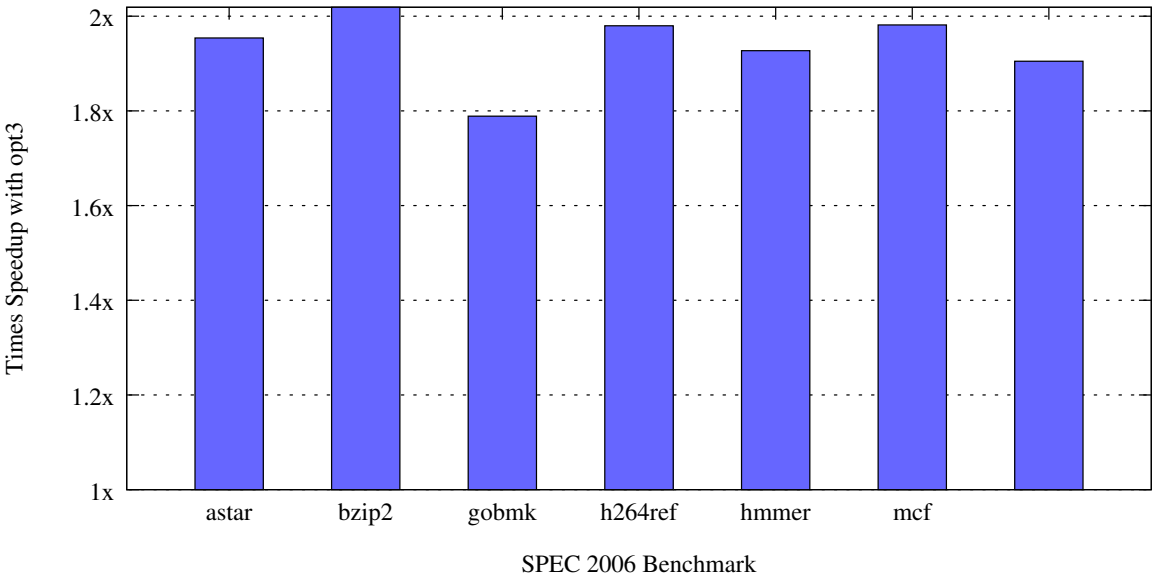

Figure 6-7: Memory trace tool speedup when optimizations are enabled.

Chapter 7

Conclusion

7.1 Future Work

While our techniques achieve great success with our instruction count example, our parital inlining example performance falls short of our ambitions. In order to improve this, we need to consider a few things.

First, we should look into register re-allocation. Pin uses a linear scan register allocator to steal scratch registers from the application which it can use in the inline code to avoid extra register spills. If we used this approach, we could eliminate most of the cost of a context switch. Performing register reallocation would be a large task requiring integration with the core of DynamoRIO in order to accurately handle signal translation.

We should also consider providing general-purpose building blocks for common tasks instead of asking tool authors to use plain C. For example, we are currently considering integrating some of the memory access collection routines from DrMemory into a DynamoRIO extension. With this support, tools would be able to efficiently instrument all memory accesses without worrying about whether their instrumentation meets our inlining criteria.

Another building block we could provide is general purpose buffer filling similar to what was done in PiPa.[15] With a general purpose buffering facility, tool authors do not need to worry about whether their calls were inlined, and can be confident that DynamoRIO has inserted the most efficient instrumentation possible.

Another improvement we could make is to allow the tool to expose the check to us explicitly. The idea is to have the tool give us two function pointers for conditional analysis: the first returns a truth value

indicating whether the second should be called, and the second performs analysis when the first returns true. Pin uses this approach. It requires the tool author to realize that this API exists in order to take advantage of it, but it provides more control over the inlining process. We could provide a mechanism for requesting that a routine be inlined regardless of criteria and raise an error on failure, allowing the tool to know what routines were inlined.

7.2 Contributions

Using the optimizations presented in this thesis, tool authors are able to quickly build performant dynamic instrumentation tools without having to generate custom machine code.

First, we have created an optimization which performs automatic inlining of short instrumentation routines. Our inlining optimization can achieve as much as 50 times speedup as shown by our instruction count benchmarks.

Second, we have built a novel framework for performing partial inlining which handles cases where simple inlining fails due to the complexity of handling uncommon cases. Partial inlining allows us to maintain the same callback interface, while accelerating the common case of conditional analysis by almost four fold.

Finally, we present a suite of standard compiler optimizations operating on instrumented code. With these optimizations, we are able to ameliorate the register pressure created by inlining and avoid unnecessary spills. Without our suite of optimizations, we would not be able to successfully inline many of our example tools.

Once these facilities have been contributed back to DynamoRIO, we hope to see more tool authors choose DynamoRIO for its ease of use in building fast dynamic instrumentation tools.

References

[1] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *The International Symposium on Code Generation and Optimization*, CGO, (Chamonix, France), Apr 2011.

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, PLDI, pp. 89–100, 2007.

[3] "Helgrind: a thread error detector." http://valgrind.org/docs/manual/hg-manual.html.

[4] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA, (New York, NY, USA), pp. 62–71, ACM, 2009.

[5] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building workload characterization tools with Valgrind," in *Invited tutorial, IEEE International Symposium on Workload Characterization*, IISWC, (San José, California, USA), 2006.

[6] M. Carbin, S. Misailovic, M. Kling, and M. Rinard, "Detecting and escaping infinite loops with jolt," in *25th European Conference on Object-Oriented Programming*, ECOOP, 2011.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, (New York, NY, USA), pp. 190–200, ACM, 2005.

[8] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *USENIX Security Symposium*, (San Francisco), Aug 2002.

[9] D. Bruening, *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[10] S. P. E. Corporation, "Spec cpu2006 benchmark suite," 2006.

[11] K. Adams, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pp. 2–13, ACM Press, 2006.

[12] D. A. Solomon and M. E. Russinovich, *Inside microsoft windows 2000.* 2000.

[13] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *Proceedings of ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, PLDI, pp. 121–133, 2009.

[14] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *The International Symposium on Code Generation and Optimization*, CGO, (Toronto, Canada), Apr 2010.

[15] Q. Zhao, I. Cutcutache, and W. F. Wong, "Pipa: pipelined profiling and analysis on multi-core systems," in *The International Symposium on Code Generation and Optimization*, CGO, (New York, NY, USA), pp. 185–194, 2008.