Analysis of Performance Overheads in DynamoRIO Binary Translator

©2020

Sandip Dey

Submitted to the graduate degree program in Electrical Engineering and Computer Science Department and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Prasad Kulkarni, Chairperson

Dr. Jerzy W. Grzymala-Busse

Dr. Esam Eldin Mohamed Aly

Date defended: February 03, 2020

The Thesis Committee for Sandip Dey certifies
that this is the approved version of the following thesis :

Analysis of Performance Overheads in DynamoRIO Binary Translator

Dr. Prasad Kulkarni, Chairperson

Date approved: February 03, 2020

Abstract

Dynamic binary translation is the process of translating instruction code from one architecture to another while it executes, i.e., dynamically. As modern applications are becoming larger, more complex and more dynamic, the tools to manipulate these programs are also becoming increasingly complex. DynamoRIO is one such dynamic binary translation tool that targets the most common IA-32 (a.k.a. x86) architecture on the most popular operating systems - Windows and Linux. DynamoRIO includes applications ranging from program analysis and understanding to profiling, instrumentation, optimization, improving software security, and more. DynamoRIO uses several optimization techniques like code caching, trace creation, optimized software technique to emulate indirect branch instructions, etc. to reduce the translation overhead and enhance program performance in comparison to native execution. However, even considering all of these optimization techniques, DynamoRIO still has the limitations of performance and memory usage, which restrict deployment scalability. The goal of this thesis is to break down the various aspects which contribute to the overhead burden and evaluate which factors directly contribute to this overhead. This thesis will discuss all of these factors in further detail. If the process can be streamlined, this application will become more viable for widespread adoption in a variety of areas. We have used industry standard MI benchmarks in order to evaluate in detail the amount and distribution of the overhead in DynamoRIO. Our statistics from the experiments show that DynamoRIO executes a large number of additional instructions when compared to the native execution of the application. Furthermore, these additional instructions are involved in building the basic blocks, linking, trace creation, and resolution of indirect branches, all of which in return have contributed to the frequent exiting of the code cache. We will discuss in detail all of these overheads, show statistics of instructions for each overhead and finally show the observations and analysis in this experiment.

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Kulkarni for his unwavering guidance and support throughout this endeavor. His patience and wisdom have been greatly appreciated. Without him, completing this thesis may not have been possible. I would also like to extend my gratitude to all of the professors under whom I have worked and studied throughout my time at the University of Kansas. I am indebted to them for their teaching and their counsel. I am deeply grateful to my thesis committee members for their time and investment in my education, making my final completion of this degree possible. Special appreciation is due to my colleagues in the lab for their encouragement, humor, and hardworking enthusiasm.

Finally, I want to thank my Dad, who through his life and example inspires me every day to work towards being the very best that I can be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's modern applications are now assembled and defined at runtime, taking advantage of shared libraries, plugins, dynamically-generated code, and other dynamic mechanisms. The amount of information available statically is shrinking. As the demand for runtime management tools are growing, dynamic binary translators (DBT) [20], because of their dynamic code translations from original (guest) binary code to cached translated binary code, are becoming more and more indispensable.

Dynamic binary translators translate instructions from the emulated application ("*guest*") in fragments. It acts in a *sandboxed* environment in which these fragments are created by building dynamic basic blocks [1], creating patches and traces. Sandboxing in DBTs is a mechanism to control and monitor code execution. DBTs retain complete control during the execution of the guest code. To retain control, application code is copied into the code cache with control transfers so that it consistently returns to the emulation manager. DBT systems work much more efficiently when the execution frequency of the translated code is high ("*hot*") [4] to compensate for the cost of translating the code every single time. However, even if the translated code is hot, other factors contribute to the loss of emulation performance. One of the greatest sources of performance overhead is the resolution of indirect branches.

DBT systems have numerous uses in program instrumentation, profiling [6],[19], program optimization [10], binary portability [2],[22],[24] and secure execution [13],[16]. Poor performance restricts the scalability and effectiveness of DBT systems. There are many current research projects working to optimize DBT performance. One of the most popular DBT systems, DynamoRIO [4], employs code caching technique [21] so that the already translated code does not need to be trans-

lated again, block-chaining technique [7] to retain the execution within the code cache, and the previously discussed hot code emulation technique[1]. It also uses different techniques to resolve indirect branches [11] . All of these are discussed in detail in the next chapter. All of these approaches have resulted in significant increase in the performance of DBT systems over other emulation techniques, but they still come with a cost of performance and memory overhead. Therefore, further work must be done in order to streamline these systems and allow for wider adoption of DBT systems. Previously, similar studies have been performed in order to assess the effects of individual optimization techniques [4], [11] in a DBT. Other, similar research has also taken place evaluating DBTs on microarchitectural structures [21]. This project confirms these past findings, but looks at the topic more deeply by performing an instruction-level analysis of exactly where and how the DBTs resources are spent. Here we have assessed and identified the major areas of performance overload as well as identified the primary factors which contribute to this overload. Our hope is that this research will contribute to further work on DBT process improvement.

We talk further in the following chapters about the overview of the DynamoRIO infrastructure, various overheads associated with it, our experimental setup, and then finally present our results and analysis on the overheads of the DynamoRIO running on MI benchmarks. We also talk about any future work and our conclusion in the final chapters.

Our objective is to determine all of the performance overheads in the most popular and sophisticated x86 to x86 dynamic binary translator called DynamoRIO [4] for the MI Benchmarks suite. The goal of this thesis is to study the inner workings of DynamoRIO in order to understand the primary reasons contributing to its performance overheads.

Chapter 2

An Overview of the DynamoRIO Infrastructure

DynamoRIO is a fully-implemented runtime code manipulation system that supports code transformations on any part of the application program on the fly. It employs itself in the most common x86 architecture on Windows and Linux Operating System and hence, unlike other dynamic binary translators, DynamoRIO spends little time in translating from application code to target cached code. In fact, the cached application code looks just like the original code with the only exception of control transfer instructions. DynamoRIO ensures transparency and retains control over the cached code through these transfer instructions. As usual, the translated code is kept in the *code cache* [4] for future re-execution hence optimizing execution of application code. We describe below the different operations of DynamoRIO and the flow of control between them. Figure 2.1, from [4], shows the overview architecture of DynamoRIO.

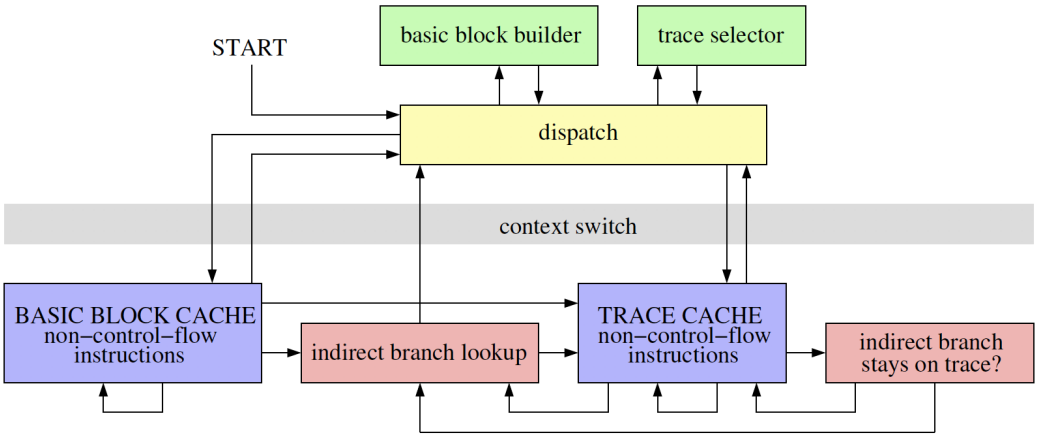

Figure 2.1: Flowchart of DynamoRIO [4]

2.1 The Emulation Manager a.k.a *dispatch*

The central hub of control flow in DynamoRIO is the "*dispatcher*" [4]. It is reached at the beginning of the execution and at every time that control leaves the code cache. Its main job is to retain control throughout the execution and redirect the control to different components of DynamoRIO. Its main loop [5] is responsible for checking if the current fragment of code that is next to be executed has been translated or not. A number of functions within the dispatcher helps to detect that and if they find no translation has been done yet then they translate a new one and *emits* it to the code cache. In addition to that, it also uses helper functions to maintain the statistics that identify *hot code* regions in the translated blocks of code and controls the building and installation of *trace* fragments. Ultimately, the dispatcher then transfers control to the translated code in the code cache. No further control is reachable to the dispatcher now and it will only be invoked again after the code cache exit. Figure 2.2, from [5], shows the main loop of *dispatch* from DynamoRIO.

```
1 void dispatch(dcontext_t *dcontext) {
2     fragment_t *targetf;
3     targetf = fragment_lookup_fine_and_coarse(dcontext, ...)
4     do {
5         if (targetf != NULL) {
6             targetf = monitor_cache_enter(dcontext, targetf);
7             break;
8         }
9         if (targetf == NULL)
10            targetf = build_basic_block_fragment(dcontext, ...);
11    } while (true);
12    dispatch_enter_fcache(dcontext, targetf)
13    ASSERT_NOT_REACHED();
14 }
```

Figure 2.2: Main loop in Dispatch [5]

2.2 Translation

2.2.1 Fragment Lookup

As discussed upon entry, the dispatcher checks if the next application code to be executed has already been translated or not through the lookup routine, *fragment_lookup_fine_and_coarse* [5].

In DynamoRIO, information about each translated block is kept in a central hash table. Each entry in the hash table is a pair of memory addresses where the guest or the application address serves as the key and its address of the corresponding translated fragment as value. If a match occurs, the lookup routine returns the address of the corresponding translated fragment, otherwise it returns an invalid address indicating that a new translation is required.

2.2.2 Building Basic Block

The creation of basic block is accomplished by the routine *build_basic_block_fragment* [5]. DynamoRIO considers each entry point to begin a new basic block, and copies it until a control transfer is reached, even if the tail of an existing basic block is duplicated. Then every decoded instruction is pushed to a linked list that represents the fragments of code being translated. Even the control transfer instruction goes into the list without patching. Unlike other DBT systems DynamoRIO spends little time translating from source to target architecture since it's the same x86 architecture. As a matter of fact, DynamoRIO copies and pastes most of the code that it executes.

2.3 Stubs or Patching

After every external API clients are given a chance to check and modify the decoded instruction code from the instruction list, DynamoRio calls its own patching routine, *mangle_bb_ilist* [5]. This routine ensures that the application code is never executed and modifies the targets of the direct branches so that they always transfer execution to the next translated block in the code cache or back to the dispatcher. This control is added to the end of the basic blocks as *stubs* and the process is called the block chaining technique.

2.4 Indirect Branch Lookup

Indirect branches, on the other hand, can have an unlimited number of targets and cannot be resolved through the block chaining technique. Hence, DynamoRIO uses different customized rou-

tines for each kind of indirect transfer but they all use the same kind of hashing technique to convert the guest application address to the code cache address. The hash table for indirect branch lookup resides within the code cache.

2.5 Trace Building

The trace building technique in DynamoRIO starts after translation and by marking certain basic blocks as potential trace heads. They receive a counter that is incremented after every execution of those blocks. So when a threshold is reached, that block and every subsequent blocks to be executed can be added as a new trace until an end of trace condition is reached.

Figure 2.3 shows the performance summary of the fundamental components of DynamoRIO as presented in the paper [4] for SPEC CPU2000 benchmark suite.

System Components	Average slowdown	
	SPECFP	SPECINT
Emulation	~300x	~300x
Basic block cache	3.54x	17.16x
+ Link direct branches	1.32x	3.04x
+ Link indirect branches	1.05x	1.44x
+ Traces	1.02x	1.17x
+ Optimizations	0.88x	1.13x

Figure 2.3: Performance Summary presented before [4]

Chapter 3

Overheads in DynamoRIO

In the previous chapter, we reviewed the entire infrastructure of DynamoRIO and described in detail the various operations in a Dynamic Binary Translator(DBT) system [20]. As we have seen, DynamoRIO retains control of the translated application throughout the execution in the host operating system. In doing so, it ends up adding a significant amount of extra instructions from the original guest code which contribute to the performance overhead of the DBT systems. In fact, the instructions added by DynamoRIO to maintain transparency and retain control on the execution can go up to 30% more than the original native code. In this chapter, we will go through each one of these places where DynamoRIO is adding performance overhead. We will also point out the place in the code cache where the original guest code is executed till it reaches the control transfer. We are going to consider this region as the "no overhead" region since the instructions executed in this region would be the same as native execution. In Chapter 5 we will see the statistics for each overhead phase in DynamoRIO for the MI benchmark problems suite.

3.1 Initialization Overhead

From the start of the process, like all DBT systems, DynamoRIO uses a considerable amount of time and memory to initialize the system, to load and link the libraries and to prepare the system for execution. Furthermore, DynamoRIO occupies the same address space as the application, operating within the application process. As you can see in Figure 3.1, from [4], DynamoRIO interposes itself between an application and the underlying operating system and hardware.

DynamoRIO also implements runtime algorithm for adapting the cache size to match the ap-

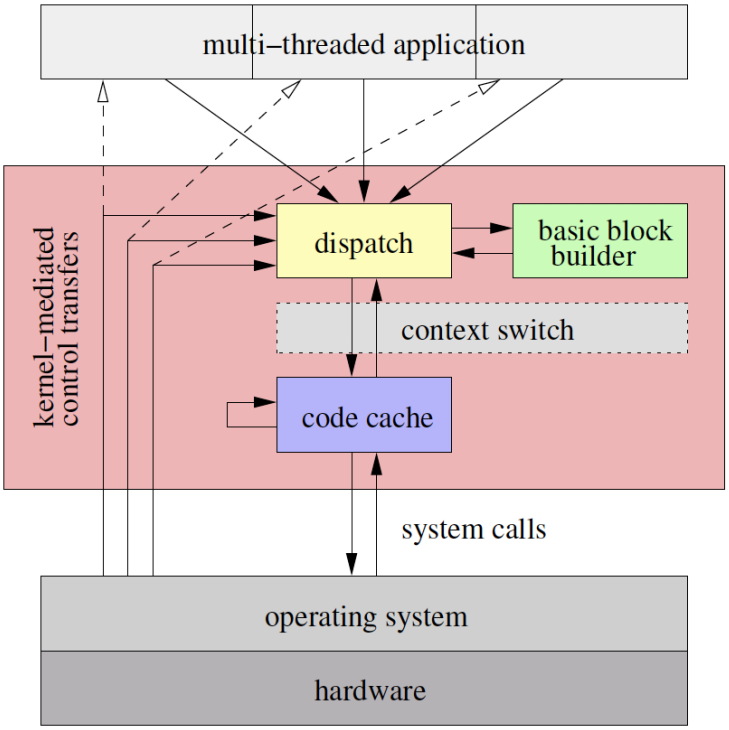

Figure 3.1: DynamoRIO runtime code manipulation layer [4]

plication's working set size. Finally, the runtime manipulation system must protect itself from erroneous application so that it can prevent the application from modifying its data and avoid the result of the runtime system failing. DynamoRIO also protects itself by dividing the execution into two modes, each with different privileges : *DynamoRIO mode* and *Application mode* where DynamoRIO mode corresponds to DynamoRIO code, while application mode corresponds to the code cache and the DynamoRIO-generated routines that are executed inside the cache without performing a context switch back to DynamoRIO. For the two modes, DynamoRio gives each type of memory page with privileges as shown in Figure 3.2 [4].

All of these initialization processes contribute to a large number of additional instructions which in turn cause a huge performance overhead for DBT systems. This begins when the program starts and continues until it reaches the central hub of DynamoRio, *the dispatcher*.

Page Type	DynamoRIO mode	Application mode
Application code	R	R
Application data	RW	RW
DynamoRIO code cache	RW	R (E)
DynamoRIO code	R (E)	R
DynamoRIO data	RW	R

Figure 3.2: Privileges of each type of memory page belonging to the two modes of DynamoRIO [4]

3.2 DR Dispatcher Control Overhead

We have already desrcibed in Chapter 2 about the importance of the dispatcher in DynamoRio and how it controls the entire execution of the application program in the system. It is the central hub of control flow in DynamoRio. It retains the control throughout the execution and redirects the control to different parts of DynamoRio. It is reached at the beginning of DynamoRio execution, just after all the initialization is complete, and every time after the control leaves the code cache. It controls everything outside the code cache and also prepares the code cache so that the control returns back to the dispatcher every time except at the end when the program terminates or when there is no need to exit the code cache (for example, if the application program goes into an infinite loop). It essentially delineates the border between the code cache and the emulator. It is responsible for verifying whether the current fragment of code that needs to be executed has already been translated or not. In order to do so, it utilizes a couple of helper functions, one of which is majorly responsible for the lookup of the translated block. If this is not found, the helper function proceeds to translate the untranslated block of code at that time. The other helper function is responsible for monitoring high frequency or "*hot*" code.

Hence, many of the performance overheads in the system are imposed in dispatcher control, as it is the command center for DynamoRio.

3.3 Translation Phase Overhead

3.3.1 Build Basic Block

Whenever the dispatcher is unable to locate some piece of code, new translation is initiated. This is done by *build_basic_block* routine. The application code is partially decoded and ends whenever a control transfer is reached. Every decoded instruction is then added to the linked list including the last control transfer instruction. The performance overhead associated with the basic block translation starts when the control enters the *build_basic_block* routine, translates the code, adds the translated code to the linked list, emits the code to the code cache and then finally returns back to the dispatcher again. However, despite these drawbacks, DynamoRio performs better in translation performance than other DBTs because it operates on the same x86 architecture and hence only copies and pastes the application code to the code cache with the control transfer at the end.

3.3.2 Build/Control Trace

After the translation is over and before entering the code cache, a one time check is always performed in DynamoRIO to see if the newly translated code can be a potential trace head and further optimize them. Each of the trace heads receives a counter that is incremented after every execution in the code cache and if a threshold is reached to the size of the counter, then that block and every subsequent block is added to become a new trace until an end-of-trace condition is reached. The whole process starts in the *monitor_cache_enter* where the trace is detected or built and ends when the control returns back to the dispatcher to finally enter the code cache with the newly translated basic block and any detected new trace as well.

3.4 Context Switch Overhead

In order to maintain transparency DynamoRIO creates no new thread, so each application thread is also DynamoRIO thread. This is achieved through context switching which saves and restores the application state as it exits and enters the code cache, creating no DynamoRIO only thread. DynamoRIO optimizes this operation by saving and restoring only general purpose registers, the condition codes(*eflags* register) and any operating system dependent state. The state preservation work both ways i.e., while entering and exiting the code cache. Hence, context switch in DynamoRIO acts as a gatekeeper to the code cache. Context switch imposes very little performance overhead to the system however it still counts towards the additional instructions used by DynamoRIO to execute the guest application code.

The below mentioned overheads are all added inside the code cache.

3.5 Stubs Execution Phase Overhead

Figure 3.3, from [4], shows what example basic block looks inside the DynamoRIO code cache. After the basic blocks are executed and before the targets of its exits have been decided, DynamoRIO puts two exit stubs at the end of these basic blocks to determine which exit was taken by the control. Each exit stub records a pointer to its own data structure (*dstub0* or *dstub1*) before transferring control to the context switch, so that DynamoRIO can figure out which branch was taken. Once an exit from a basic block is linked, the corresponding exit stub is no longer needed again unless the exit is later unlinked. Hence, DynamoRio keeps the direct exit stubs in a separate cache from the basic block body, so that it can delete and re-create direct exit stubs on demand if needed reducing the overall memory usage.

Typically, a basic block inside code cache contains 6-7 instructions including the exit stub taken and the 3 additional instructions for exit stubs executed every time a basic block is executed, if unlinked, causes a performance overhead in the code cache.

```
fragment7: add %eax, %ecx
           cmp $4, %eax
           jle stub0
           jmp stub1
    stub0: mov %eax, eax-slot
           mov &dstub0, %eax
           jmp context_switch
    stub1: mov %eax, eax-slot
           mov &dstub1, %eax
           jmp context_switch
```

Figure 3.3: Example of a Basic Block in Code Cache [4]

3.6 Indirect Branch Resolution Phase Overhead

We have till now, presented parts of the code in DynamoRIO that are written in the C language. They are responsible for translation, interpretation, code emission, dispatch, and fragment lookup. However, indirect branch lookup is critical to the execution performance, and so has an alternate implementation in DynamoRIO. Instead of invoking the dispatcher, during runtime, DynamoRIO places fast, specialized, lookup routines inside the code cache, enabling control to directly transfer to other fragments even when emulating indirect branches.

Figure 3.4, from [8], shows the assembly code of the optimized address translation routine for indirect jump emulation. This code is responsible for iterating over the hash tables described in Section 2.4. In label L0, the hash index is calculated, based on the target address received through register %ecx. Block L1 checks if the target address matches the contents of the hash table. If it does, a translation has been found, and the code in label L2 restores the machine state and transfer the execution control to the translated fragment. Otherwise, the algorithm iterates over the collision chain until it finds a translation, or until the chain is over. Block L3 checks for the end of the chain, whereas block L5 checks if the hash table itself has ended. Blocks L4 and L6 increment the pointer to the hash table entry and loop around. Blocks L7, L8, L9, L10, L11, and L12 prepare a return to

the dispatcher, because they are reached when a translation is not in the hash table. Disabling the indirect branch lookup routine and forcing the control to be transferred back to the dispatcher can be done in runtime, through the use of the runtime switch, *--no_ibl_link*.

All these asm level code are put by DynamoRIO into the code cache to resolve indirect branches. They improve the performance over the other emulator but impose a performance overhead for DynamoRIO.

```
1 L0:
2     movabs %eax,%gs:0x0
3     lahf
4     seto   %al
5     mov    %ebx,%gs:0x8
6     mov    %ecx,%ebx
7     and    %gs:0x48,%ecx
8     add    %ecx,%ecx
9     add    %ecx,%ecx
10    add    %ecx,%ecx
11    add    %ecx,%ecx
12    add    %gs:0x50,%ecx
13 L1:
14    cmp    %ebx,(%ecx)
15    jne    <L3>
16 L2:
17    mov    %edi,%gs:0x58
18    mov    %gs:0x20,%edi
19    mov    0x390(%edi),%edi
20    mov    0x3d0(%edi),%edi
21    incl   0xf0(%edi)
22    mov    %gs:0x58,%edi
23    mov    %gs:0x8,%ebx
24    jmpq   *0x8(%ecx)
25 L3:
26    cmpq   $0x0,(%ecx)
27    je     <L5>
28 L4:
29    mov    %edi,%gs:0x58
30    mov    %gs:0x20,%edi
31    mov    0x390(%edi),%edi
32    mov    0x3d0(%edi),%edi
33    incl   0xf8(%edi)
34    mov    %gs:0x58,%edi
35    lea    0x10(%ecx),%ecx
36    jmpq   <L1>
37 L5:
38    cmpq   $0x1,0x8(%ecx)
39    jne    <L8>
40 L6:
41    mov    %edi,%gs:0x58
42    mov    %gs:0x20,%edi
43    mov    0x390(%edi),%edi
44    mov    0x3d0(%edi),%edi
45    incl   0x100(%edi)
46    mov    %gs:0x58,%edi
47    mov    %gs:0x50,%ecx
48    jmpq   <L1>
```

```
49 L7:
50    mov    %ebx,%gs:0x8
51    mov    (%ecx),%ebx
52 L8:
53    mov    %ebx,%ecx
54    mov    %edi,%gs:0x58
55    mov    %gs:0x20,%edi
56    mov    0x390(%edi),%edi
57    mov    0x3d0(%edi),%edi
58    incl   0xfc(%edi)
59    mov    %gs:0x58,%edi
60    mov    %gs:0x8,%ebx
61    add    $0x7f,%al
62    sahf
63    movabs %gs:0x0,%eax
64 L9:
65    mov    %edi,%gs:0x18
66    mov    %gs:0x20,%edi
67    mov    %eax,0x38(%edi)
68    mov    %ecx,0x2d8(%edi)
69    movabs $0x71311bd0,%eax
70    mov    0x38(%edi),%ecx
71    mov    %ecx,%gs:0x0
72    mov    %gs:0x10,%ecx
73    mov    %gs:0x18,%edi
74    jmpq   <out of range>
75 L10:
76    mov    %edi,%gs:0x58
77    movabs %eax,%gs:0x0
78    lahf
79    seto   %al
80    mov    %gs:0x20,%edi
81    mov    0x390(%edi),%edi
82    mov    0x3d0(%edi),%edi
83    incl   0x108(%edi)
84    add    $0x7f,%al
85    sahf
86    movabs %gs:0x0,%eax
87    mov    %gs:0x58,%edi
88    jmpq   <L9>
89 L11:
90    jmpq   <L0>
91 L12:
92    jmpq   <L10>
93    nop
94    nop
95    nop
96    nop
```

Figure 3.4: Shared Indirect Branch Lookup Routine code as emitted to the code cache [8]

3.7 Resolution of *fcache* Exits Overhead

After the control comes out of the code cache through the context switch to the dispatcher, the dispatcher navigates it to resolve the reason for the *fcache* exit. Once it figures out the reason from where it came out and why, it begins resolving the reason and goes on building the basic blocks and traces again, hence repeating the already discussed processes till it re-enters the code cache. There can be several reasons for the code cache to exit. We have highlighted those reasons in chapter 5 with statistics of instruction count. The control can come out of the code cache normally when it can't find anymore translated block to execute inside the code cache, system call execution, client redirection, native execution, indirect branch exit for no translated blocks in the lookup table, coarse grain fragment etc.

These additional instructions created by DynamoRIO in order to resolve the code cache exit contribute further to the performance overhead.

3.8 No Overhead Phase

The only aspect of the DynamoRio execution that causes no performance overhead in the system is the process of executing the original guest code. DynamoRio, unlike other DBTs, copies and pastes the original guest code into the code cache until the control transfer is reached. So this original code executed within the code cache has the same performance impact in the system as that of native execution and thus this is considered as the no overhead phase in our experiment. As a matter of fact, instructions executed in the no overhead phase in DynamoRio are 65% of the overall instruction count required to run an application in DynamoRio.

Chapter 4

Experimental Setup

In order to gather the required scientific data for the different overheads present in DynamoRIO, we had to put our custom code in DynamoRIO that is mainly responsible for capturing the memory addresses of the different overhead locations. We then build our own custom script which creates breakpoints at those memory locations and then measure the total number of instructions executed at each breakpoint, which in return tells us the total additional instructions DynamoRIO uses at different overhead locations. To setup the breakpoints we used the *ptrace* [18] linux tool and to calculate the instructions at the register level, we used *libpfm4* [23] library. We compile both MiBenchmark [9] and DynamoRIO with the GNU/GCC compiler, in its 4.4.3 version, using the -O2 optimization flag. We used the most recent 7.0 version of DynamoRIO during our experiment setup and whatever overheads we show in the following chapter are calculated in its vanilla version, i.e., without any modifications.

We present our analysis by compiling different sets of data to harvest the different factors that possibly lead to the run-time overhead (or speed-up) of different benchmarks run through DynamoRIO.

4.1 DynamoRIO setup

As we have already described the different locations of overheads in DynamoRIO in the previous chapter, we will now discuss how the exact memory locations were detected so that individual breakpoints could be set at each of these locations. Following this process, it will then be possible to find out the total number of instructions executed at each of these locations.

We started with the *core/dispatch.c* file which is the main working file of DynamoRIO. We get the *d_r_dispatch* sub routine over here which as previously discussed is the control manager of the entire DynamoRIO operation. We take a dump of its starting memory address in a text file and later on when a breakpoint is set at this memory segment we can then know the *dispatcher control* overhead has started. Similarly, we also take a note of the starting memory addresses of the two other subroutines, *build_basic_block* and *monitor_cache_enter*, present inside the main loop of *d_r_dispatch* , which as previously discussed help us to monitor the translation overhead in DynamoRIO. Using the command *nm -n* on the DynamoRIO shared library we can access a list of all these memory addresses together in one text file *"nm_maps.txt"*.

```
system("nm -n /projects/zephyr/Sandip/dynamorio/debug_build/lib64/release/libdynamorio.so.debug |
            grep 'dynamorio_so_start\\|d_r_dispatch\\|build_basic_block_fragment\\|monitor_cache_enter\\|sd_' > nm_maps.txt");
```

Figure 4.1: Code snippet showing the *nm - n* command to get the starting memory addresses of the mentioned DynamoRIO subroutines

In order to mark the starting memory addresses of the other overheads in DynamoRIO, we put our own code inside the DynamoRIO source files. Our custom code will create a text file, *memory_mapping.txt*, from DynamoRIO that contains the list of starting memory addresses for *context switch, stubs* and *indirect branch resolution* overheads from three other DynamoRIO source files namely *core/emit.c, arch/emit_shared_utils.c* and *arch/x86/emit_utils.c*. For each translated basic block and trace block, it's corresponding memory addresses in the code cache are also taken note of so that anytime control reaches these memory addresses, we can count that under the "no overhead phase" as these are the original block of code from the source application. Furthermore, in order to mark the different entry points in DynamoRIO after the exit from the code cache, dummy functions have also been inserted in the DynamoRIO source code at appropriate places to mark their starting memory addresses for breakpoint creation later.

```
754             cache_pc sd_stub_start_pc;
755
756             sd_stub_start_pc = EXIT_STUB_PC(dcontext, f, l);
757
758             file_t fs = INVALID_FILE;
759             const char *file = "memory_mapping.txt";
760
761             fs = os_open_protected(file, OS_OPEN_WRITE | OS_OPEN_ALLOW_LARGE | OS_OPEN_APPEND);
762             if (fs != INVALID_FILE){
763               print_file(fs, "%llx STUBS\n", sd_stub_start_pc);
764               os_close(fs);
765               fs = INVALID_FILE;
766             }
767             else
768               print_file(STDOUT,"Saving the file was unsuccessful for STUB from emit.c\n");
```

Figure 4.2: Code snippet showing how the starting memory address of each *stubs* is saved in the *memory_mapping.txt* text file

4.2 Instruction Count through single stepping

Initially we took an approach of counting the total number of instructions executed for each overhead by single stepping through every instructions executed by DynamoRIO and keeping a count of it in a separate custom script. In order to do so we took a dump of the ending memory addresses as well along with the starting memory addresses of each overhead location in DynamoRIO discussed in the previous section. We built a separate custom script to use the *ptrace* Linux tool to single step the entire execution phase in DynamoRIO. In order to do so our custom script was forked with a child and a parent process where the child process was responsible for running the DynamoRIO and the parent process was monitoring the child process. In the child process, the *ptrace* command with the PTRACE_TRACEME is executed, asking the parent process to monitor the child process execution of DynamoRIO. We will discuss in detail about the *ptrace* Linux command in the next section.

So how our single stepping method would work is by comparing the contents of the *rip* register gathered with the *ptrace* command executed with the request PTRACE_GETREGS, with the memory locations gathered from the DynamoRIO source files. Each section of memory segments gathered from the DynamoRIO source code into the *memory_mapping.txt* was considered as the different overhead phase and every time our custom script will compare the contents of the *rip*

register with the overhead section and mark that as the start of the particular overhead phase. After the overhead phase has been marked, the parent sends the *ptrace* command with the request PTRACE_SINGLESTEP to the child process, so that for each instruction executed a counter can be incremented which would essentially give us the instructions count of the overhead phase.

However, the single stepping process seems most accurate but it is difficult to implement for large test input, since single stepping through the entire execution for large test input is not possible as it might take weeks to complete the execution. So we implemented the method of creating a breakpoint instead of single stepping where a custom script was created that will act as a debugger and create breakpoints at each overhead phase starting location. Every time control stops at each of these breakpoint location, that particular overhead phase is marked and instructions are counted from the hardware level using the *rdpmc* [14] instruction counter until the counter stops again at the next overhead phase breakpoint. This method speeds up our experiment by several times as the same test input that took 1.5 weeks to complete using the single stepping method now takes 40 minutes to complete. Hence, for our experiment we used the breakpoint method.

4.3 Breakpoints implemented manually

We build a separate custom script in order to create breakpoints at the different memory segments corresponding to each overhead. This custom script is essentially forked with a child and a parent process where the child process is responsible for executing our benchmark suites through DynamoRio and the parent process helps in monitoring the entire process. The memory addresses from DynamoRIO are therefore essential for the parent process to monitor the content of rip (instruction pointer) register during the execution of the child process, and thereby mark the different phases of execution that correspond with the different overhead. To understand our agenda, we need to understand how does a debugger do its work [3]. A debugger can start some process and debug it, or attach itself to an existing process. It can single-step through the code, set breakpoints and run to them, examine variable values and stack traces. Our custom script will create a similar debugger that will attach itself to the child process, set breakpoints at the different overhead start-

ing memory locations and single step through the code to count the total number of instructions executed until it reaches the next breakpoint or overhead.

To set a breakpoint at some target address in the traced process, the debugger does the following:

1. Remember the data stored at the target address

2. Replace the first byte at the target address with the int 3 instruction (we will talk about the int 3 instruction in detail later in this chapter)

```
253  /* Enable the given breakpoint by inserting the trap instruction at its
254  ** address, and saving the original data at that location.
255  */
256  static void enable_breakpoint(pid_t pid, debug_breakpoint* bp)
257  {
258      assert(bp);
259      bp->orig_data = ptrace(PTRACE_PEEKTEXT, pid, bp->addr, 0);
260      ptrace(PTRACE_POKETEXT, pid, bp->addr, (bp->orig_data & ~0xFF) | 0xCC);
261  }
```

Figure 4.3: Code snippet showing how to create a breakpoint with int3 instruction

Then, when the debugger asks the OS to run the process, the process will run and eventually hit upon the int 3, where it will stop and the OS will send it a signal. This is where the debugger comes in again, receiving a signal that its child (or traced process) was stopped. It can then:

1. Replace the int 3 instruction at the target address with the original instruction

2. Roll the instruction pointer of the traced process back by one. This is needed because the instruction pointer now points after the int 3, having already executed it.

3. Allow the user to interact with the process in some way, since the process is still halted at the desired target address. This is the part where we do our math to count the total number of instructions executed between each breakpoint.

4. When the user wants to keep running, the debugger will take care of placing the breakpoint back (since it was removed in step 1) at the target address, unless the user asked to cancel the breakpoint.

```
263  /* Disable the given breakpoint by replacing the byte it points to with
264  ** the original byte that was there before trap insertion.
265  */
266  static void disable_breakpoint(pid_t pid, debug_breakpoint* bp)
267  {
268      assert(bp);
269      long data = ptrace(PTRACE_PEEKTEXT, pid, bp->addr, 0);
270      assert((data & 0xFF) == 0xCC);
271      ptrace(PTRACE_POKETEXT, pid, bp->addr, (data & ~0xFF) | (bp->orig_data & 0xFF))
272  }
```

Figure 4.4: Code snippet showing how to disable a breakpoint

All of this is achieved through the ptrace call. ptrace is declared thus (in *sys/ptrace.h)*:

*long ptrace(enum _ptrace_request request, pid_t pid, void *addr, void *data)*

The first argument is a request, which may be one of many predefined PTRACE_* constants. The second argument specifies a process ID for some requests. The third and fourth arguments are address and data pointers, for memory manipulation.

In our case we have used different *ptrace_request* to achieve the functionality of a debugger such as PTRACE_TRACEME to trace the child process, PTRACE_PEEKTEXT to remember the original data stored at the target address(Step 1), PTRACE_POKETEXT to replace the first byte at the target address with the int 3 instruction(Step 2) and PTRACE_GETREGS to ensure if we indeed stopped every time at the breakpoint by gathering the content of the *rip* register and comparing it with the breakpoint location. Finally PTRACE_CONT request to let the process run.

4.3.1 More on int 3

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. We can now simply say that breakpoints are implemented on the CPU by a special trap called int 3. int is x86 jargon for "trap instruction" - a call to a predefined interrupt

```
318  int resume_from_breakpoint(pid_t pid, debug_breakpoint* bp)
319  {
320      struct user_regs_struct regs;
321      int wait_status;
322
323      ptrace(PTRACE_GETREGS, pid, 0, &regs);
324      /* Make sure we indeed are stopped at bp */
325      // assert(regs.rip == (long) bp->addr + 1);
326
327      /* Now disable the breakpoint, rewind RIP back to the original instruction
328      ** and single-step the process. This executes the original instruction that
329      ** was replaced by the breakpoint.
330      */
331      regs.rip = (long) bp->addr;
332      ptrace(PTRACE_SETREGS, pid, 0, &regs);
333      disable_breakpoint(pid, bp);
334      if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0) < 0) {
335          perror("ptrace");
336          return -1;
337      }
338      wait(&wait_status);
339
340      if (WIFEXITED(wait_status)) {
341          return 0;
342      }
343
344      /* Re-enable the breakpoint and let the process run.
345      */
346      enable_breakpoint(pid, bp);
347
348      if (ptrace(PTRACE_CONT, pid, 0, 0) < 0) {
349          perror("ptrace");
350          return -1;
351      }
```

Figure 4.5: Code snippet showing how to resume from a breakpoint by single stepping using *ptrace*

handler. x86 supports the int instruction with a 8-bit operand specifying the number of the interrupt that occurred, so in theory 256 traps are supported. The first 32 are reserved by the CPU for itself, and number 3 is the one we're interested in here - it's called "trap to debugger" . The definition of int 3 from Intel's manual [3]:

" The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code"

int instructions on x86 occupy two bytes - 0xcd followed by the interrupt number. int 3 could've been encoded as cd 03, but there's a special single-byte instruction reserved for it - 0xcc. This allows us to insert a breakpoint without ever overwriting more than one instruction. Having a special 1-byte encoding for int 3 solves this problem. Since 1 byte is the shortest an instruction can get on x86, we guarantee that only the instruction we want to break on gets changed.

4.4 Instruction Count through *rdpmc*

So every time we reach a breakpoint we start calculating the total number of instructions executed in between which add towards the performance overhead. In order to calculate these instructions we take the help of an external library *"perform2"*. This library uses the *"read performance monitoring counters (rdpmc)"* to count the number of instructions executed for a particular event. The RDPMC instruction allows application code running at a privilege level of 1, 2, or 3 to read the performance-monitoring counters if the PCE flag in the CR4 register is set. This instruction is provided to allow performance monitoring by application code without incurring the overhead of a call to an operating-system procedure. At the end of the script we add our own maths to calculate the total instructions executed for each overhead by summing the *rdpmc* count for each overhead at each interval.

```
#if defined(__x86_64__) || defined(__i386__)

#ifdef __x86_64__
#define DECLARE_ARGS(val, low, high)  unsigned low, high
#define EAX_EDX_VAL(val, low, high) ((low) | ((uint64_t )(high) << 32))
#define EAX_EDX_ARGS(val, low, high)  "a" (low), "d" (high)
#define EAX_EDX_RET(val, low, high) "=a" (low), "=d" (high)
#else
#define DECLARE_ARGS(val, low, high)  unsigned long long val
#define EAX_EDX_VAL(val, low, high) (val)
#define EAX_EDX_ARGS(val, low, high)  "A" (val)
#define EAX_EDX_RET(val, low, high) "=A" (val)
#endif

#define barrier() __asm__ __volatile__("": : :"memory")

static inline int rdpmc(struct perf_event_mmap_page *hdr, uint64_t *value)
{
unsigned a, d, c;
c = (1<<30);
__asm__ volatile("rdpmc" : "=a" (a), "=d" (d) : "c" (c));
*value = (uint64_t)(((unsigned long)a) | (((unsigned long)d) << 32));

  return 0;
}
```

Figure 4.6: Code snippet showing how to count instructions using *rdpmc*

Chapter 5

Experimental Results and Analysis

In this project we used MIBenchmark suite to evaluate DynamoRIO and the performance overhead caused by it. MiBench has many similarities to the EEMBC benchmark suite as described on their web site (http://www.eembc.com). However, MiBench is composed of freely available source code. Where appropriate, we provide a small and large data set. The small data set represents a light-weight, useful embedded application of the benchmark, while the large data set provides a more stressful, real world application. MiBench consists of six categories including: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. Finally, an automated script was written to compile and execute the benchmarks.

For each of the experiments described in Chapter 5, we prepare the environment for the execution of the benchmarks by isolating the machine from the network, setting the processor power states to maximum performance, and clearing eventual zombie processes. Then we invoke the automated scripts on MIBench which run each application three times, with the reference input. After the experiments are complete, we collect the data produced by the automated scripts and select the mean value of several runs to statistically represent the true central index of dispersion in the computer science experiments.

All the experiments are run on a single machine, featuring a pair of Intel processors at 2.4 GHz, 32 GiB's of RAM, and a 64-bits Ubuntu LTS 10.04 operating system. We compile both MIBench and DynamoRIO with the GNU/GCC compiler, in its 4.4.3 version, using the -O2 optimization flag.

The remainder of this chapter describes how we evaluate the techniques described in Chapter 3, and DynamoRIO itself. We also present our experiments and their results, as well as our analysis

of the obtained results. We will have two graphs for each section for both small and large input test data.

5.1 x86 Results and Analysis

Our MI benchmark has a shorter running time over the SPEC benchmarks so the Initialization overhead takes the most number of instructions to initiate the DynamoRIO. In the Appendix we have presented the tables that show the instruction count of each overhead phase in DynamoRIO. The initialization phase has a static number of instructions that it requires in order to startup a DBT system. We have already discussed in Chapter 3 exactly what the processes are that the Initialization phase is responsible for. Since these processes are fixed and all DBT systems require more or less all of these processes to initialize, hence the DynamoRIO has a fixed number of instructions before it is able to reach the *dispatcher*.

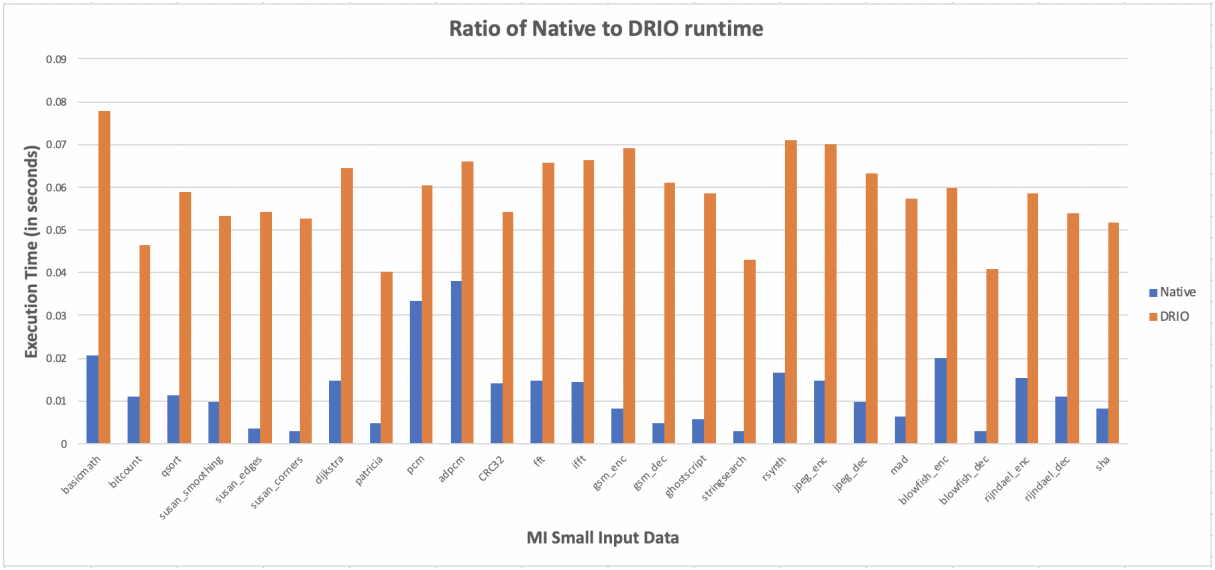

Figure 5.1: Ratio of DynamoRIO run times to Native run times for different small MI benchmark test inputs

Figure 5.1 and Figure 5.2 shows the ratio of the DynamoRIO execution time to native run of different benchmark test inputs. This tells us approximately how much performance overhead DynamoRIO puts on a running application. Our MI benchmark suite has two types of test input data

- small and large. The performance overhead of DynamoRIO is more prominent in small running applications since it takes a higher number of additional instructions to execute a small application there than in its native execution. As previously discussed DynamoRIO has a fixed number of instructions to startup the process and translate the application code, hence these additional instructions becomes a larger overall overhead for small applications. Figure 5.1 from our experiment validates that fact. Figure 5.2 although shows a considerable overhead from DynamoRIO but since the application size is large, the running time of the application in native mode is still closer to the running time with DynamoRIO.

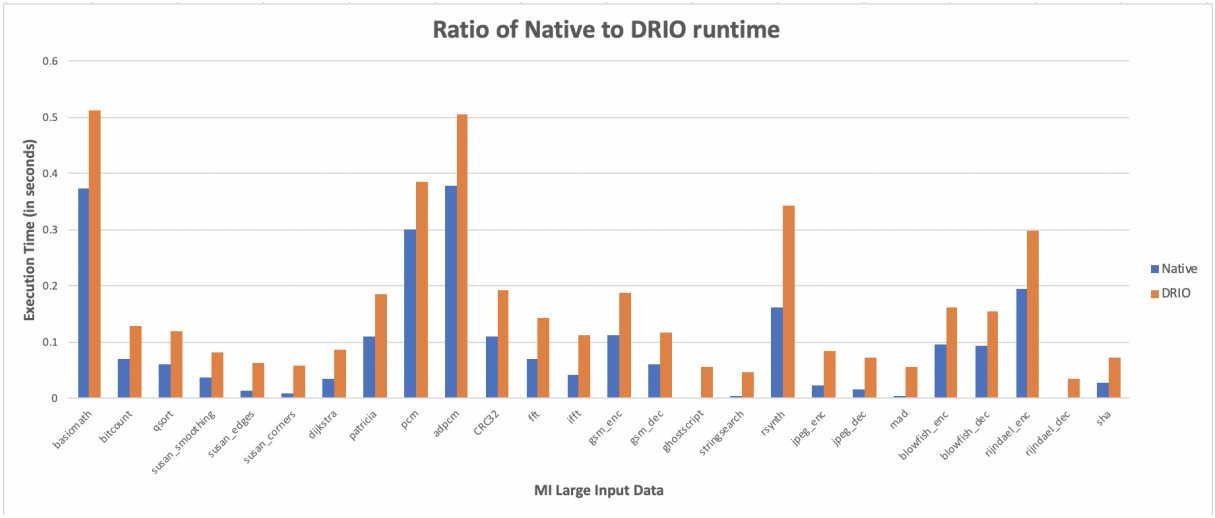

Figure 5.2: Ratio of DynamoRIO run times to Native run times for different large MI benchmark test inputk

Figure 5.3 and and 5.4 shows the performance summary graph of the MI Benchmark without including the Initialization phase. This makes the overhead phases more prominent. We can see from the graphs that the "Translation Phase" overhead now dominates the graph. Again, this is because the number of instructions required for creation of a basic block or a trace block are constant. Since we are dealing with the small test size benchmark input data the translation phase overhead dominates the graph after the Initialization phase. The translation phase is responsible for converting the source binary code to the target code. DynamoRIO is more of a dynamic binary optimizer than a binary translator which means it essentially copies the source application code to the target code cache with just adding the control transfer instructions at the end of each translated

block in order to retain control during execution in the code cache. As these process is constant for every translated code block, hence, the total number of instructions during the translation phase is almost constant. Our experimental data shows DynamoRIO imposes approximately 30% of overhead over the native execution.

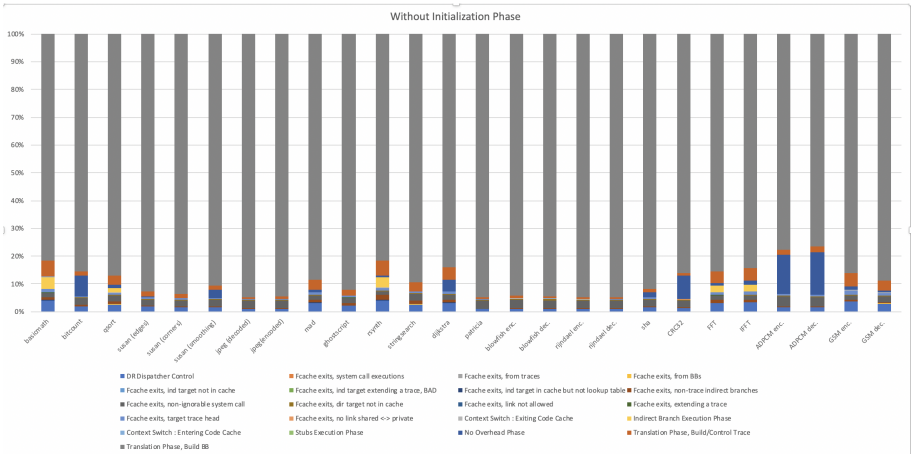

Figure 5.3: Performance Summary for MI Small Benchmark without the Initialization Phase

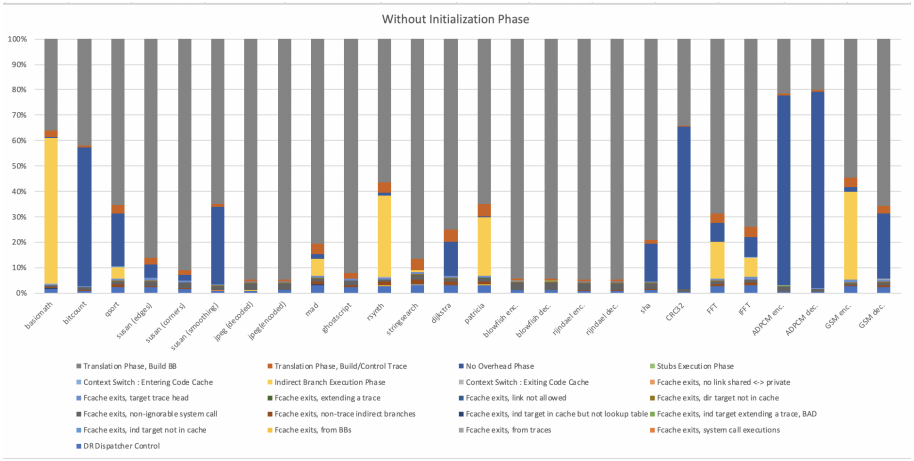

Figure 5.4: Performance Summary for MI Large Benchmark without the Initialization Phase

Figure 5.5 and Figure 5.6 shows the performance summary of the DynamoRIO with MI Benchmark suit without the "translation phase". The graph is more prominent now in showing most of the total performance overheads in DynamoRIO. As we have already discussed in Chapter 3 that the *indirect branch* overhead and *fcache exits* during execution affect the performance of DynamoRIO more than any other overhead - the results from our experiments validate that fact. We can see

from the graph the different *fcache exits* that affect the performance overhead in DynamoRIO. The "No Overhead" phase (green bar in the graph) is the actual time taken to execute the original application code. Although DynamoRIO has optimized the translation process, we still can see from our results that in some cases, mainly for the *basicmath, ghostscript, stringsearch, patricia* large test input benchmarks, performance overhead is more than the actual execution time. The *basicmath, rsynth, patricia, gsm encoding* large test input for MI benchmark shows the domination of the indirect branch overhead. Although DynamoRIO uses the *software technique* of mapping the source pc to target pc in a hash-table in order to resolve the indirect branches without exiting the code cache, it still proves to impose a high performance overhead for DynamoRIO.

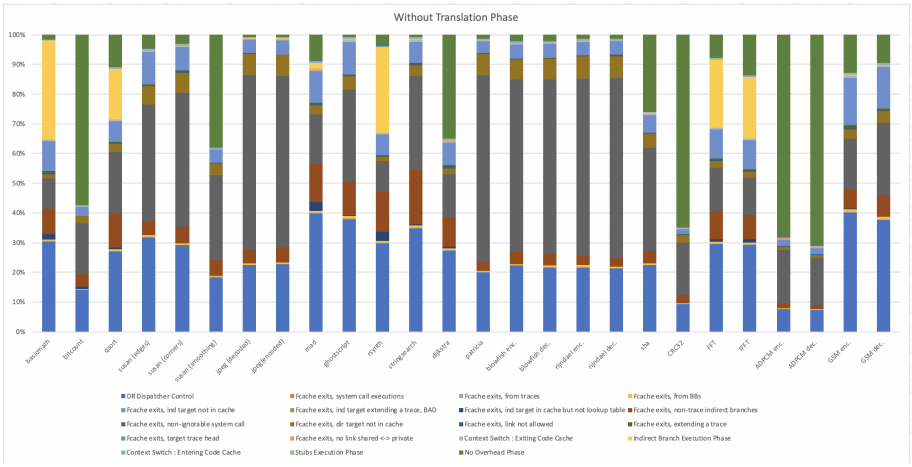

Figure 5.5: Performance Summary for MI Small Benchmark without the Translation Phase

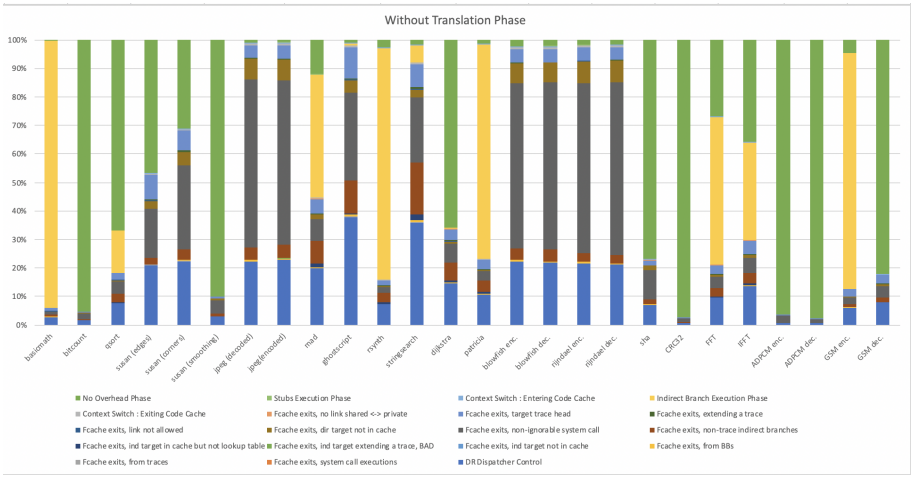

Figure 5.6: Performance Summary for MI Large Benchmark without the Translation Phase

Figures 5.7 and 5.8 show all of the DynamoRIO performance overheads. The graph shows how the *indirect branch resolution* and the *fcache exits* dominate the performance overhead. The optimization techniques incorporated in DynamoRIO including caching, linking, trace formation, and instruction inlining have been able to considerably improve the execution time spent in the fragment cache. However, the additional instructions executed could not be fully alleviated through the optimization techniques as an overhead of around 30% could still be seen in terms of run-time captured. The overhead would be accountable to the ability of the hardware to handle the excess instruction load.

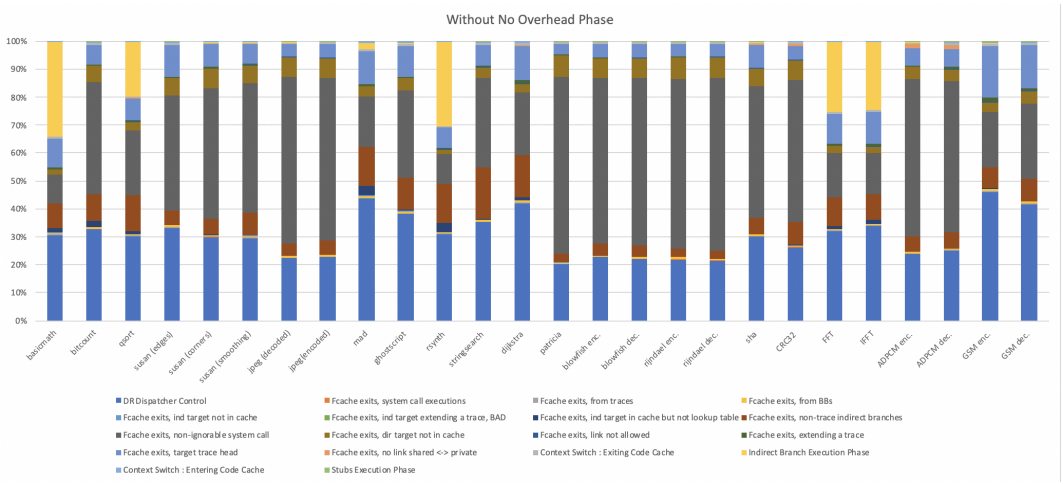

Figure 5.7: Performance Summary for MI Small Benchmark without the Overhead Phase

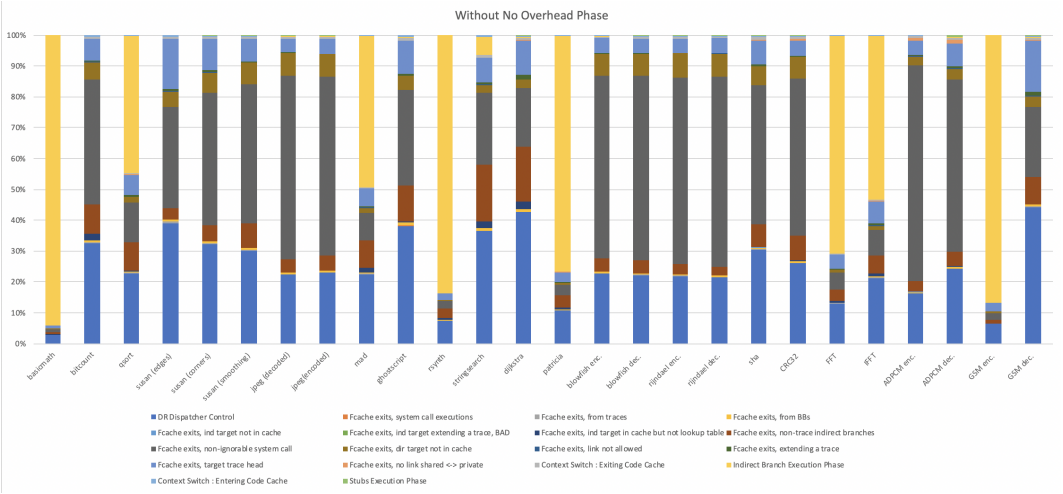

Figure 5.8: Performance Summary for MI Large Benchmark without the Overhead Phase

In summary, not just the *fcache exits* but also the *indirect branch resolution* impact the performance overhead or speed up. To address the issue of overhead, more research needs to be done on how to reduce the number of fcache exits and how to employ a better technique for indirect branch resolution. This can be done with larger hash-tables, page tables and more software techniques to be implemented for the indirect branch predictor. Options for parallelization within the DynamoRIO code base and eager translation to compile basic blocks ahead of time(analogous to pre-fetching) can be explored to improve speed.

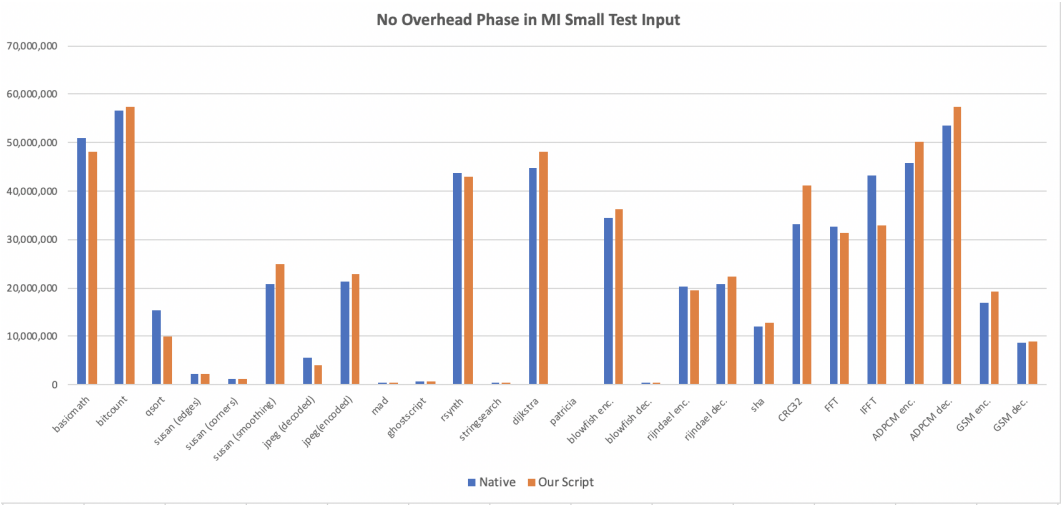

Figure 5.9: Ratio of native instructions count to our script count for small test input

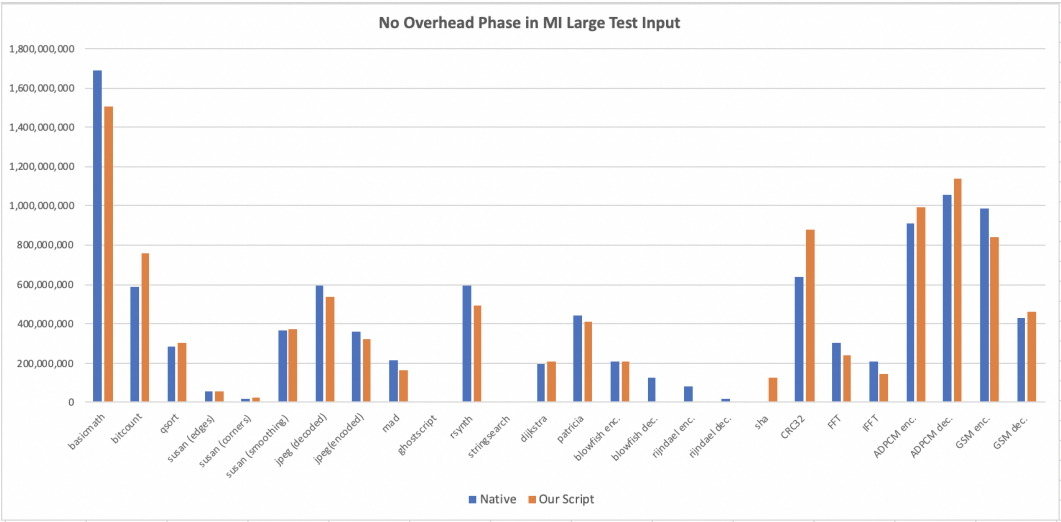

Figure 5.10: Ratio of native instructions count to our script count for large test input

Figures 5.9 and figure 5.10 show the ratio of the instructions count during native run to the instructions count which our experiment calculated as the "no overhead phase" for our MI benchmark test input. We have used the Linux command *perf* [25] to capture the instructions count statistics for the native run of our MI benchmark suit. Below is the sample *perf* command for native execution of our *bitcount test input*, for both small and large.

perf stat -B -e cycles, instructions bitcnts 75000 > bitcount_small.txt

perf stat -B -e cycles, instructions bitcnts 1125000 > bitcount_large.txt

Our "no overhead phase" count from the experiment shown in the tables in Appendix are almost equal to the instructions count from the native run. This validates that our experiment was performed with accuracy.

Chapter 6

Related Work

As these limitations of dynamic binary translation systems are commonly known, previous research has been conducted in order to evaluate the various causes of performance overhead in these systems. Some techniques have already been developed for improving performance related to the previously described limitations. One similar example is the work of Arkaitz Ruiz and Kim Hazelwood [21] which investigates the process of running applications within the DBT. They used both DynamoRIO [4] and also Pin [19], [17] for their experiments, but focused primarily on the impact within Pin of the various aspects of DBT execution. They employed the *perf* tool [25]to track the hardware counters for various hardware events during program execution. The results of their study indicate that the primary cause of the overhead is the greater number of instructions executed versus the native execution, causing high L1 instruction cache misses and iTLB misses for most of the benchmarks. Similarly, we also employed the *perf* tool to collect our own hardware statistics during the experiments and have confirmed similar results.

In addition to the impact of the DBT on the hardware, we have also explored the number of exits, basic blocks and traces, and runtime and compilation time in order to grasp a wider picture of how the DBT affects the overall application execution. Clearly, the root cause of the high overhead largely stems from the large instruction execution from the code base of DynamoRIO during translation. This occurs because of the control exiting the code cache for translation. As confirmed by our own research, it has been found that a big contributor to the number of exits is indirect branch execution and no linking between basic blocks and trace heads. Thus, the fundamental cause of overhead relates to the source pc to target pc translation in executing basic blocks and traces from within the code caches.

SPIRE [15] is one known approach for handling hot indirect branches during source pc translation. Research on its use has found that, rather than exiting the code cache for translation, it is helpful to place a trampoline at the source PC address that causes redirection of the control to the code cache in order to execute the translated code. Since SPIRE system modify the original source code with a trampoline, there is a need to maintain code transparency to the application. In order to do this a code space is created with the same size as the original source binary and the trampolines are added in the new space, without touching the original source code.

While SPIRE uses probe-based method, DynamoRIO uses JIT-based compilation with *software prediction* to resolve indirect branches. DynamoRIO uses a compare-jump list for the indirect branch to jump to the appropriate target pc, as the source pc stays unknown until the branch instructions have been executed. With this technique, only the corresponding mapping of source pc to the target pc has to be checked from the hash-table which has all the source to target pc mappings. However, if the mapping is not found then the whole lookup-table has to be checked to resolve the indirect branching. It is proven before[15] that the probe-based method in SPIRE is more efficient than the *software based* approach of DynamoRIO for indirect branch extensive benchmark.

The work of HQEMU [12] has also shown to help in reducing the exits from code cache when executing the trace code. The main goal of HQEMU is to improve the quality of translated code by additional compilation of intermediate code from QEMU and LLVM translation. LLVM is an optimization intensive compiler. Apart from improving the code quality HQEMU also implements a technique called *trace merging* to reduce the exits while executing the traces in code cache. The idea is to merge the traces that are frequently executed into a single trace avoiding the switching between code cache and DBT for the trace address resolution. These would reduce the large number of exits from code cache and therefore reducing a considerable amount of performance overhead.

Various other techniques have been discussed, created, and evaluated for reducing the performance impact of auxiliary tasks in DBT systems. Various optimization techniques have been

shown to improve different aspects, for example; the introduction of a basic block code cache or trace generation, can help to improve both startup and steady-state performance. Trace generation can help improve performance in long-running programs.

Other work has focused on whether persistent code caching can be used to reduce startup overhead in DBT systems. Persistent code caches enable code reuse by making translations available for storage and reuse across executions. Some techniques even handle dynamically generated code. These techniques have demonstrated improvement in performance, but do raise some security concerns as well as need to be warmed up and lack full effectivity when an unseen program without representation in the code cache is executed.

Chapter 7

Future Work

Dynamic Binary Translator is a dynamic area of study with much potential for future study and application. For this project, we have focused on testing and analyzing the primary contributions to performance overhead in DynamoRio as compared to those in native program execution. Looking ahead, there are multiple aspects of this work that would benefit from further study.

The high instruction volume and loss of instruction locality in current DBTs are issues that significantly impact hardware condition and efficiency, particularly for applications with large code bases. In order to successfully and efficiently run DBTs, improved hardware techniques will be necessary. It may also prove helpful to combine improved hardware and software techniques in order to optimize DBT performance. Ideally, these various areas of improvement will eventually combine and applications may even operate better than they do in their native performance environments.

A more immediate and pressing intervention will focus on reducing the number of exits from the code cache while performing various tasks. There are three primary causes for exits from the code cache - these include indirect branches, basic block translation, and trace formation. Together, the exits required from these three processes contribute a substantial proportion of overall performance overhead sustained by DBT program execution. We believe that the most applicable and immediately helpful future work will consist of efforts to reduce the currently high number of exits. When mechanisms to reduce the number of exits are developed, these tasks will be parallelized and thus will allow the main program execution to remain in the code cache, smoothing the process and positively impacting program efficiency.

Chapter 8

Conclusions

Dynamic binary translation is a promising technology with many potential applications across a wide variety of industries. It has the potential to actualize portable program execution, improve overall performance, and improve monitoring and instrumentation in order to ensure a secure execution environment. The major drawback still limiting widespread DBT adoption is the fact that program execution still causes some amount of performance overhead, sometimes to a significant level. This thesis has employed various experiments in order to better describe and quantify the various specific causes for this performance overhead, to support future research efforts directed toward reducing this overhead burden.

DynamoRIO is an x86 to x86 dynamic binary translator that is available today. It is advanced and popular, functioning as a binary translator as well as an instrumentor and optimizer. The performance overhead of a DBT is due to a combination of factors. Our Master Thesis conducts a series of experiments to measure the effect of program execution through DynamoRIO and gather all the information to calculate the performance overhead caused by DynamoRIO which are mostly consistent with earlier results. In particular, we confirmed that program execution in a DBT exerts greater pressure on the instruction cache due to a higher volume of executed instructions compared to native program execution.

Additionally, we designed and implemented additional experiments to further understand exactly what contributes to the higher volume of executed instructions and the associated performance overheads. These experiments confirm earlier research showing that much of the performance overhead burden is due to frequent code cache exits which occur in order to enable additional DBT tasks. Our experiments and associated graphics illustrate the number of exits for each

category and the total number of exits performed throughout program execution. As described in detail above, the major categories of code cache exits include block translation, indirect branch resolution, and trace formation. In the future we hope to develop techniques of parallelization which will enable these processes to be conducted faster and in advance to reduce the total number of code cache exits and thus reduce overall DBT performance overhead.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[3] Eli Bendersky. How debuggers work: Part 2 - breakpoints. accessed from https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints.

[4] Derek L. Bruening and Saman Amarasinghe. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, USA, 2004. AAI0807735.

[5] Q. Z. Derek L. Bruening. Dynamorio: Dynamic instrumentation tool platform. accessed from https://github.com/DynamoRIO/dynamorio, April 2018.

[6] H. K. Cho, T. Moseley, R. Hank, D. Bruening, and S. Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, Feb 2013.

[7] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, page 128–137, New York, NY, USA, 1994. Association for Computing Machinery.

[8] Gabriel F. T. Gomes and Edson Borin. Indirect branch emulation techniques in virtual machines, 2014.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, page 3–14, USA, 2001. IEEE Computer Society.

[10] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 68–78, USA, 2015. IEEE Computer Society.

[11] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, page 61–73, USA, 2007. IEEE Computer Society.

[12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 104–113, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, page 2–12, New York, NY, USA, 2006. Association for Computing Machinery.

[14] Intel. Rdpmc: Read performance-monitoring counters. accessed from https://www.felixcloutier.com/x86/rdpmcl.

[15] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. *SIGPLAN Not.*, 48(7):1–12, March 2013.

[16] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, page 191–206, USA, 2002. USENIX Association.

[17] Osnat Levi. Pin - a dynamic binary instrumentation tools. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2012.

[18] LINUX. Ptrace - process trace. accessed from http://man7.org/linux/man-pages/man2/ptrace.2.html.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[20] Mark Probst. Dynamic binary translation, 2003.

[21] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *2008 IEEE International Symposium on Workload Characterization*, pages 131–140, Sep. 2008.

[22] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical report, USA, 2001.

[23] seranian. perfmon2. accessed from https://git.code.sf.net/p/perfmon2/libpfm4 perfmon2-libpfm4, 2011.

[24] David Ung and Cristina Cifuentes. Dynamic binary translation using run-time feedbacks. *Sci. Comput. Program.*, 60(2):189–204, April 2006.

[25] wiki. perf: Linux profiling with performance counterss. accessed from https://perf.wiki.kernel.org/index.php/Main_Page.

Appendix A

Instructions Count for Each Overhead in DynamoRIO for Small and Large MI Benchmark

Overheads	basicmath	bitcount	qsort	susan(edges)	susan(corners)	susan(smoothing)
DR Dispatcher Control	64177699	14002512	24337837	15698093	12372997	11995503
Fcache exits, system call executions	11150	11372	24119	17344	17306	17024
Fcache exits, from traces	569153	51131	114360	130732	56576	30493
Fcache exits, from BBs	1078879	284900	462697	272651	204087	264062
Fcache exits, ind target not in cache	62380	3172	19700	15106	15095	21167
Fcache exits, ind target extending a trace, BAD	1903	104	1102	74	65	117
Fcache exits, ind target in cache but not lookup table	3805617	856250	636089	12525	12987	34887
Fcache exits, non-trace indirect branches	18297837	4065555	10320092	2391779	2407129	3361413
Fcache exits, non-ignorable system call	21570755	17198960	18631928	19475955	19124918	18687605
Fcache exits, dir target not in cache	3248024	2432536	2553472	2939579	2936622	2632901
Fcache exits, link not allowed	412921	78227	125886	116402	69163	66798
Fcache exits, extending a trace	1543379	92722	449606	220566	183372	133925
Fcache exits, target trace head	21579513	3008515	6218591	5376703	3345273	2826072
Fcache exits, no link shared <-> private	183119	20703	53745	7609	7582	7582
Translation Phase, Build/Control Trace	91738921	11134536	31043430	16064012	12052481	11122159
Translation Phase, Build BB	1359477444	656677384	816402419	827041206	803920844	744860336
Initialization	140738346366412	140737519339414	140737519217977	140737519647931	140737519732664	140737519488486
Context Switch : Exiting Code Cache	1137299	289014	445755	303419	245362	239340
Indirect Branch Execution Phase	70612640	83596	15533869	54886	36169	63110
Context Switch : Entering Code Cache	713851	153258	274482	169043	132061	127469
No Overhead Phase	3405889	57422508	9857839	2338023	1345923	24806958
Stubs Execution Phase	188049	91908	89183	78976	60356	60659

Table A.1: Instruction count for each overhead for MI Auto/Industrial small benchmark

Overheads	jpeg (decoded)	jpeg(encoded)	mad
DR Dispatcher Control	6096506	6372832	102598652
Fcache exits, system call executions	11141	11138	65728
Fcache exits, from traces	12960	19657	572585
Fcache exits, from BBs	137685	158983	1777931
Fcache exits, ind target not in cache	23677	18643	91335
Fcache exits, ind target extending a trace, BAD	0	31	3333
Fcache exits, ind target in cache but not lookup table	6849	4982	7934001
Fcache exits, non-trace indirect branches	1198276	1382751	33049167
Fcache exits, non-ignorable system call	16105938	16252214	42198367
Fcache exits, dir target not in cache	1984936	1995474	8240327
Fcache exits, link not allowed	46889	24008	527636
Fcache exits, extending a trace	42571	51693	1604506
Fcache exits, target trace head	1200028	1333092	27646562
Fcache exits, no link shared <-> private	1600	1600	243912
Translation Phase, Build/Control Trace	4358531	5149048	117220370
Translation Phase, Build BB	564010560	577364483	2923340998
Initialization	140737519401061	140737519398803	140737519910897
Context Switch : Exiting Code Cache	95442	139592	1808888
Indirect Branch Execution Phase	21721	29659	5064622
Context Switch : Entering Code Cache	66010	68716	1141509
No Overhead Phase	200999	227629	22930610
Stubs Execution Phase	64182	45558	290542

Table A.2: Instruction count for each overhead for MI Consumer small benchmark

Overheads	ghostscript	rsynth	stringsearch
DR Dispatcher Control	31792378	58190978	16923140
Fcache exits, system call executions	27159	11533	11374
Fcache exits, from traces	127871	436416	66712
Fcache exits, from BBs	621212	963287	354139
Fcache exits, ind target not in cache	42590	52396	13346
Fcache exits, ind target extending a trace, BAD	234	2242	451
Fcache exits, ind target in cache but not lookup table	325162	5953724	205181
Fcache exits, non-trace indirect branches	9647228	26540727	8821809
Fcache exits, non-ignorable system call	25800297	20028589	15351787
Fcache exits, dir target not in cache	3752840	2701617	1815677
Fcache exits, link not allowed	178667	263214	78026
Fcache exits, extending a trace	322348	1123828	324630
Fcache exits, target trace head	9130100	13697957	3412252
Fcache exits, no link shared <-> private	7833	3786	0
Translation Phase, Build/Control Trace	29239416	80876171	21318770
Translation Phase, Build BB	1322999584	1235865359	587218346
Initialization	140737519808871	140737519781118	140737519398049
Context Switch : Exiting Code Cache	572973	1017254	318661
Indirect Branch Execution Phase	354395	56101390	149164
Context Switch : Entering Code Cache	355347	677771	192495
No Overhead Phase	687877	7394316	437314
Stubs Execution Phase	103720	159690	84469

Table A.3: Instruction count for each overhead for MI Office small benchmark

Overheads	dijkstra	patricia
DR Dispatcher Control	37413872	4829670
Fcache exits, system call executions	11144	11141
Fcache exits, from traces	184816	12831
Fcache exits, from BBs	708135	113064
Fcache exits, ind target not in cache	7114	13144
Fcache exits, ind target extending a trace, BAD	3179	0
Fcache exits, ind target in cache but not lookup table	896600	1618
Fcache exits, non-trace indirect branches	13456310	692928
Fcache exits, non-ignorable system call	19990729	15147223
Fcache exits, dir target not in cache	2847859	1816869
Fcache exits, link not allowed	227036	21965
Fcache exits, extending a trace	1077606	31763
Fcache exits, target trace head	10686092	938698
Fcache exits, no link shared <-> private	201150	0
Translation Phase, Build/Control Trace	54274034	3317003
Translation Phase, Build BB	1012976574	502698877
Initialization	140737519443785	140737519479251
Context Switch : Exiting Code Cache	678419	111422
Indirect Branch Execution Phase	217716	22041
Context Switch : Entering Code Cache	418241	50840
No Overhead Phase	48150071	303675
Stubs Execution Phase	84545	41378

Table A.4: Instruction count for each overhead for MI Network small benchmark

Overheads	blowfish enc.	blowfish dec.	rijndael enc.	rijndael dec.	sha
DR Dispatcher Control	5816569	5492009	5702168	5355890	10924672
Fcache exits, system call executions	11141	11138	21419	11141	14681
Fcache exits, from traces	12104	11904	11659	18358	46580
Fcache exits, from BBs	103593	158711	180349	121109	216605
Fcache exits, ind target not in cache	3257	12673	12946	4181	23499
Fcache exits, ind target extending a trace, BAD	13	13	0	0	13
Fcache exits, ind target in cache but not lookup table	5377	12944	1846	1619	18259
Fcache exits, non-trace indirect branches	1105136	1037985	801687	713452	2025219
Fcache exits, non-ignorable system call	15088879	14929962	15808920	15324905	17066645
Fcache exits, dir target not in cache	1781619	1736868	2013138	1853199	2306775
Fcache exits, link not allowed	34239	35392	25487	47252	71363
Fcache exits, extending a trace	37232	36795	33433	31678	68239
Fcache exits, target trace head	1264557	1189846	1199160	1143823	2851795
Fcache exits, no link shared <-> private	1600	0	1600	0	158073
Translation Phase, Build/Control Trace	4114814	4025250	3736913	3628221	8235579
Translation Phase, Build BB	504420305	494812324	553317673	514844006	637639330
Initialization	140737519506905	140737519196811	140737519351353	140737519340474	140737519328936
Context Switch : Exiting Code Cache	119557	152238	124890	101389	206769
Indirect Branch Execution Phase	17313	22220	11820	9653	41248
Context Switch : Entering Code Cache	62115	59204	59860	55965	115374
No Overhead Phase	614102	536939	396132	343139	12732204
Stubs Execution Phase	42795	43867	58786	63052	57541

Table A.5: Instruction count for each overhead for MI Security small benchmark

Overheads	CRC32	FFT	IFFT	ADPCM enc.	ADPCM dec.	GSM enc.	GSM dec.
DR Dispatcher Control	8656906	42719308	48536203	5531415	5828387	60221711	35373333
Fcache exits, system call executions	14681	11186	19120	0	0	14695	14668
Fcache exits, from traces	23752	344704	439124	44318	56258	501657	237445
Fcache exits, from BBs	191346	744691	813398	118033	136700	1088319	645748
Fcache exits, ind target not in cache	16316	53967	42233	16456	7841	17644	12397
Fcache exits, ind target extending a trace, BAD	113	2488	2311	65	65	2881	343
Fcache exits, ind target in cache but not lookup table	14497	1115634	2134389	14652	5759	343734	56230
Fcache exits, non-trace indirect branches	12662036	13695855	13195269	1310165	1330462	9791502	6957353
Fcache exits, non-ignorable system call	16775178	21384938	20976943	13042713	12564739	25780144	22784378
Fcache exits, dir target not in cache	2262632	3120335	3195329	982099	973521	4343583	3570005
Fcache exits, link not allowed	60066	293523	325643	52852	54224	503460	236555
Fcache exits, extending a trace	110670	1007039	1116278	105627	133848	1650992	729049
Fcache exits, target trace head	1575073	13949655	16588524	1378231	1474282	24260457	13110015
Fcache exits, no link shared <-> private	194142	53004	84910	386227	376898	22747	18712
Translation Phase, Build/Control Trace	8395073	59263097	67706416	6647790	7377192	84323875	43084875
Translation Phase, Build BB	630273638	1204888427	1257132008	280279415	287692000	1438027284	1092327102
Initialization	140737519225820	140737519656374	140737519777398	140737519324871	140737519352863	140737519575217	140737519549742
Context Switch : Exiting Code Cache	185852	809532	868156	118790	138748	1018098	626835
Indirect Branch Execution Phase	31630	33309914	34562231	23284	39837	488429	89512
Context Switch : Entering Code Cache	94874	483595	543414	61541	64165	640092	373920
No Overhead Phase	61218084	11453083	22803410	50142240	57390403	19261628	8973716
Stubs Execution Phase	34356	117481	161522	37640	66456	219627	131950

Table A.6: Instruction count for each overhead for MI Telecomm. small benchmark

Overheads	basicmath	bitcount	qsort	susan(edges)	susan(corners)	susan(smoothing)
DR Dispatcher Control	70631432	14016476	34504342	25269002	15394540	12649537
Fcache exits, system call executions	11412	11372	24097	34843	49807	25333
Fcache exits, from traces	682601	50101	158882	308872	99036	41210
Fcache exits, from BBs	1129953	286593	691382	330323	278836	295907
Fcache exits, ind target not in cache	55177	15719	10775	5403	19279	13378
Fcache exits, ind target extending a trace, BAD	3359	104	1109	74	78	391
Fcache exits, ind target in cache but not lookup table	4179430	866066	665687	33256	24756	58711
Fcache exits, non-trace indirect branches	19540904	4075720	13722568	2493366	2489924	3310005
Fcache exits, non-ignorable system call	22519214	17387394	19529767	21093866	20470001	18964443
Fcache exits, dir target not in cache	3361955	2412526	2652470	3185223	3080194	2869762
Fcache exits, link not allowed	475916	79265	245966	174746	81115	59229
Fcache exits, extending a trace	1763230	85201	933830	422015	379838	140187
Fcache exits, target trace head	24362011	3024793	9690861	10512625	4759904	3019703
Fcache exits, no link shared <-> private	181989	19638	373453	22805	23181	22046
Translation Phase, Build/Control Trace	103783559	11131577	50866626	28540013	18809326	11657940
Translation Phase, Build BB	1482280286	657042966	957442573	924872194	880476437	795899047
Initialization	140737519635316	140737519478902	140737519679459	140737519658518	140737519771841	140737519799847
Context Switch : Exiting Code Cache	1224345	282487	587451	401086	264753	280208
Indirect Branch Execution Phase	2359839944	70594	67148323	51758	47189	56562
Context Switch : Entering Code Cache	785027	153491	385523	270190	164574	134603
No Overhead Phase	5048316	856567299	305507011	56708882	21483683	372821560
Stubs Execution Phase	195638	50980	121456	81096	77591	56395

Table A.7: Instruction count for each overhead for MI Auto/Industrial Large benchmark

Overheads	jpeg (decoded)	jpeg(encoded)	mad
DR Dispatcher Control	6077768	6439430	99543571
Fcache exits, system call executions	11144	11144	67512
Fcache exits, from traces	11820	28122	503124
Fcache exits, from BBs	122454	114782	1804956
Fcache exits, ind target not in cache	5897	6023	49150
Fcache exits, ind target extending a trace, BAD	0	8567	4611
Fcache exits, ind target in cache but not lookup table	5985	3208	5989087
Fcache exits, non-trace indirect branches	1202818	1375591	40295114
Fcache exits, non-ignorable system call	16127637	16251718	38692381
Fcache exits, dir target not in cache	1964392	2016322	7302881
Fcache exits, link not allowed	44008	23943	469349
Fcache exits, extending a trace	48773	51666	1892012
Fcache exits, target trace head	1209272	1346678	25102877
Fcache exits, no link shared <-> private	1600	1600	406525
Translation Phase, Build/Control Trace	4314191	5155129	127348551
Translation Phase, Build BB	563996082	577415131	2627232963
Initialization	140737519303144	140737519396171	140737520162029
Context Switch : Exiting Code Cache	146751	160303	1717567
Indirect Branch Execution Phase	39939	42278	216178911
Context Switch : Entering Code Cache	66010	68716	1124056
No Overhead Phase	239864	222762	60468950
Stubs Execution Phase	39707	49523	238073

Table A.8: Instruction count for each overhead for MI Consumer Large benchmark

Overheads	ghostscript	rsynth	stringsearch
DR Dispatcher Control	31811249	76789802	25461562
Fcache exits, system call executions	26953	43546	11374
Fcache exits, from traces	121239	639802	150813
Fcache exits, from BBs	644251	1199761	459530
Fcache exits, ind target not in cache	36823	34623	6863
Fcache exits, ind target extending a trace, BAD	234	2467	759
Fcache exits, ind target in cache but not lookup table	323962	7680137	1528063
Fcache exits, non-trace indirect branches	9647655	33079980	12934801
Fcache exits, non-ignorable system call	25836063	23533532	16175279
Fcache exits, dir target not in cache	3730053	3394079	1902309
Fcache exits, link not allowed	193413	394391	104200
Fcache exits, extending a trace	324117	1529411	556492
Fcache exits, target trace head	9127092	19508397	5510705
Fcache exits, no link shared <-> private	7625	26502	41646
Translation Phase, Build/Control Trace	29192537	107911619	35298404
Translation Phase, Build BB	1321604457	1533332319	676365780
Initialization	140737519903663	140737519867453	140737519630193
Context Switch : Exiting Code Cache	609275	1441931	491511
Indirect Branch Execution Phase	353305	867442438	4211271
Context Switch : Entering Code Cache	355347	888593	294626
No Overhead Phase	745583	29241287	1058462
Stubs Execution Phase	93483	194337	50456

Table A.9: Instruction count for each overhead for MI Office Large benchmark

Overheads	dijkstra	patricia
DR Dispatcher Control	45690113	83631686
Fcache exits, system call executions	10880	419715
Fcache exits, from traces	256903	557222
Fcache exits, from BBs	849568	1466712
Fcache exits, ind target not in cache	23869	56360
Fcache exits, ind target extending a trace, BAD	2950	2931
Fcache exits, ind target in cache but not lookup table	2459717	4113217
Fcache exits, non-trace indirect branches	19146036	31289135
Fcache exits, non-ignorable system call	20502673	26613766
Fcache exits, dir target not in cache	2848854	3939051
Fcache exits, link not allowed	261464	485648
Fcache exits, extending a trace	1558666	2391009
Fcache exits, target trace head	11819782	24783416
Fcache exits, no link shared <-> private	204580	370590
Translation Phase, Build/Control Trace	75655953	129496729
Translation Phase, Build BB	1161086392	1689989442
Initialization	140737519381199	140737519415650
Context Switch : Exiting Code Cache	802183	1526424
Indirect Branch Execution Phase	268330	592088314
Context Switch : Entering Code Cache	521807	941196
No Overhead Phase	205964315	11132558
Stubs Execution Phase	168505	218235

Table A.10: Instruction count for each overhead for MI Network Large benchmark

Overheads	blowfish enc.	blowfish dec.	rijndael enc.	rijndael dec.	sha
DR Dispatcher Control	5784171	5542375	5698417	5347560	11585886
Fcache exits, system call executions	10920	11138	14736	11141	14681
Fcache exits, from traces	12893	11512	21310	12226	61441
Fcache exits, from BBs	118634	131979	128554	108721	212303
Fcache exits, ind target not in cache	3133	2925	4058	3956	10881
Fcache exits, ind target extending a trace, BAD	13	13	0	0	113
Fcache exits, ind target in cache but not lookup table	14781	2705	7542	840	16561
Fcache exits, non-trace indirect branches	1082620	1035257	817162	695231	2781459
Fcache exits, non-ignorable system call	15088106	14935631	15809497	15330382	17200690
Fcache exits, dir target not in cache	1767990	1765180	2018916	1871764	2327189
Fcache exits, link not allowed	29850	26121	39644	47641	67429
Fcache exits, extending a trace	36218	36217	31821	32014	170143
Fcache exits, target trace head	1252250	1178261	1192554	1152910	2943374
Fcache exits, no link shared <-> private	1600	0	1292	0	189265
Translation Phase, Build/Control Trace	4082302	4035369	3738651	3585642	11935266
Translation Phase, Build BB	504262589	494560910	553267813	514829807	668439150
Initialization	140737519327446	140737519337484	140737519372229	140737519328172	140737519191703
Context Switch : Exiting Code Cache	114534	101692	143569	96493	270941
Indirect Branch Execution Phase	24511	36191	20967	36270	48980
Context Switch : Entering Code Cache	62115	59204	59860	55965	123000
No Overhead Phase	603084	529015	422477	375040	126734555
Stubs Execution Phase	31984	77993	54630	64951	73168

Table A.11: Instruction count for each overhead for MI Security Large benchmark

Overheads	CRC32	FFT	IFFT	ADPCM enc.	ADPCM dec.	GSM enc.	GSM dec.
DR Dispatcher Control	8601985	50346900	54649383	6230927	6911515	71172218	43924862
Fcache exits, system call executions	14999	55926	54308	0	0	14748	14380
Fcache exits, from traces	19106	392408	450970	55034	85252	643359	359923
Fcache exits, from BBs	184467	880217	952548	152305	153295	1210227	749175
Fcache exits, ind target not in cache	4502	69692	45595	19106	1305	13382	10115
Fcache exits, ind target extending a trace, BAD	378	2156	854	61	65	1132	954
Fcache exits, ind target in cache but not lookup table	22423	1389065	2182058	24791	24680	607378	66314
Fcache exits, non-trace indirect branches	2670250	15285738	14872721	1315852	1336166	12148395	8798095
Fcache exits, non-ignorable system call	16784144	21316677	21599001	26861659	15997233	26475647	22426437
Fcache exits, dir target not in cache	2273304	3251975	3285372	1034587	999117	4382825	3556401
Fcache exits, link not allowed	48401	355118	376292	85244	56381	619924	357550
Fcache exits, extending a trace	106330	1177038	1264228	116132	160402	1975635	1022798
Fcache exits, target trace head	1590224	16958030	18847747	1825482	2133723	28634911	16546352
Fcache exits, no link shared <-> private	195990	180157	182343	366813	391930	304902	312955
Translation Phase, Build/Control Trace	8354087	71338431	76865657	7407960	8641404	101172055	57729855
Translation Phase, Build BB	629879963	1321586299	1349035942	286361868	294701256	1536360483	1172970780
Initialization	140737519314882	140737519662928	140737519773759	140737519341315	140737519282012	140737519504897	140737519564389
Context Switch : Exiting Code Cache	190188	903329	953962	185765	155949	1146452	767628
Indirect Branch Execution Phase	48901	274150978	136198045	29799	29655	972266382	152767
Context Switch : Entering Code Cache	94915	565431	609137	69905	76711	761780	468999
No Overhead Phase	1179860416	142140052	143456012	993731178	1138551239	53807657	459105228
Stubs Execution Phase	50741	168008	141307	41907	135440	253411	159287

Table A.12: Instruction count for each overhead for MI Telecomm. Large benchmark