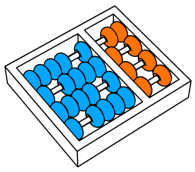
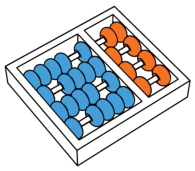


Gabriel Ferreira Teles Gomes

"Indirect Branch Emulation Techniques in Virtual Machines"

"*Técnicas para Emulação de Saltos Indiretos em Máquinas Virtuais*"

CAMPINAS
2014



University of Campinas
Institute of Computing

Universidade Estadual de Campinas
Instituto de Computação

Gabriel Ferreira Teles Gomes

"Indirect Branch Emulation Techniques in Virtual Machines"

Supervisor:
Orientador(a): **Prof. Dr. Edson Borin**

"Técnicas para Emulação de Saltos Indiretos em Máquinas Virtuais"

MSc Dissertation presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a Master degree in Computer Science.

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

This volume corresponds to the final version of the Dissertation defended by Gabriel Ferreira Teles Gomes, under the supervision of Prof. Dr. Edson Borin.

Este exemplar corresponde à versão final da Dissertação defendida por Gabriel Ferreira Teles Gomes, sob orientação de Prof. Dr. Edson Borin.

Edson Borin

Supervisor's signature / *Assinatura do Orientador(a)*

CAMPINAS
2014

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

G585i

Gomes, Gabriel Ferreira Teles, 1985-
Indirect branch emulation techniques in virtual machines / Gabriel Ferreira Teles Gomes. – Campinas, SP : [s.n.], 2014.

Orientador: Edson Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de computação. 2. Máquinas virtuais. 3. Tradução binária dinâmic. I. Borin, Edson,1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

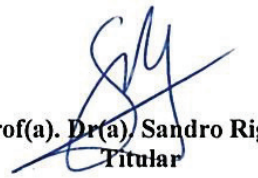
Informações para Biblioteca Digital

Título em outro idioma: Técnicas para emulação de saltos indiretos em máquinas virtuais
Palavras-chave em inglês:
Computer systems
Virtual machines
Dynamic binary translation
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Anderson Faustino da Silva
Sandro Rigo
Data de defesa: 07-07-2014
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Defesa de Dissertação de Mestrado em Ciência da Computação, apresentada pelo(a) Mestrando(a) **Gabriel Ferreira Teles Gomes**, aprovado(a) em **07 de julho de 2014**, pela Banca examinadora composta pelos Professores Doutores:

Prof(a). Dr(a). Anderson Faustino da Silva
Titular



Prof(a). Dr(a). Sandro Rigo
Titular

Prof(a). Dr(a). Edson Borin
Presidente

Institute of Computing /*Instituto de Computação*
University of Campinas /*Universidade Estadual de Campinas*

Indirect Branch Emulation Techniques in Virtual Machines

Gabriel Ferreira Teles Gomes[1]

July 07, 2014

Examiner Board/*Banca Examinadora*:

- Prof. Dr. Edson Borin (Supervisor)
- Prof. Dr. Sandro Rigo
  IC - UNICAMP
- Prof. Dr. Anderson Faustino da Silva
  DIN - UEM
- Prof. Dr. Luiz Fernando Bittencourt
  IC - UNICAMP
- Prof. Dr. Fernando Magno Quintão Pereira
  DCC - UFMG

[1]Financial support: CNPq (143517/2011-2) 2011–2012 / FAPESP (2011/16468-6) 2012–2013

Abstract

Dynamic binary translation is an emulation technique commonly employed in the implementation of virtual machines. One of the main sources of overhead that hinder the applicability of dynamic binary translators is that caused by the emulation of indirect branch instructions. This master thesis describes several techniques that try to improve the performance and efficiency of indirect branch emulation in efficient virtual machines. DynamoRIO is one of such machines and it implements features used by several of those techniques. In this master thesis, we present current implementations of DynamoRIO, modify its code to include two new techniques (Inline Caching and IBTC) and compare it with other techniques described in the literature.

Resumo

Tradução dinâmica de binários é uma técnica de emulação comumente utilizada na implementação de máquinas virtuais. Neste contexto, a emulação de saltos indiretos é uma das principais fontes de perda de eficiência, o que atrapalha a aplicabilidade de tradutores dinâmicos de binários. Essa dissertação descreve diversas técnicas que tentam melhorar o desempenho e a eficiência da emulação de saltos indiretos em máquinas virtuais eficientes. O DynamoRIO é uma máquina virtual que se enquadra nessa categoria e que utiliza características de diversas dessas técnicas. Nessa dissertação, nós apresentamos a implementação atual do DynamoRIO, modificamos seu código para incluir duas novas técnicas de emulação de saltos indiretos (Inline Caching e IBTC) e as comparamos com outras técnicas descritas na literatura.

Contents

Abstract **ix**

Resumo **xi**

1 Introduction **1**
1.1 Philosophical Approach 1
1.2 Technical Approach 1
1.3 Dynamic Binary Translation Commons 2
1.4 Same-ISA Process Virtual Machines 3

2 An Overview of the DynamoRIO Infrastructure **4**
2.1 The Emulation Manager 4
2.2 Fragment Lookup 5
2.3 Translation 5
2.4 Patching 6
2.5 Dispatch 7
2.6 Fragment optimization 7

3 A Detailed View of DynamoRIO 4.1 **9**
3.1 Hash Tables 9
3.2 Fragment Lookup 10
3.3 Translation 11
3.4 Patching 12
3.5 Emission 12
3.6 Indirect Branch Lookup Routines 13

4 Software Techniques for Indirect Branch Emulation **15**
4.1 Indirect Branches in Static Translators 15
4.2 Inline Caching 15
4.3 Speculative Chaining 16

4.4 Code expansion . . . 17
4.5 Indirect Branch Translation Cache . . . 17
4.6 Sieve . . . 18
4.7 Fast Returns . . . 19
4.8 Shadow Stack . . . 19
4.9 Return Cache . . . 20
4.10 Indirect Branches in DynamoRIO . . . 21

5 Implementation of the techniques **23**
5.1 Inline Caching . . . 24
5.2 IBTC . . . 25

6 Related Work **28**

7 Methodology and Results **36**

8 Conclusion **43**

Bibliography **45**
Bibliography . . . 48

List of Tables

2.1 Breakdown of the slowdown over native execution 8

4.1 Indirect branch emulation techniques 22

7.1 Hit rates for the IBTC technique 40

List of Figures

2.1 Dispatcher: the central hub of DynamoRIO 5
2.2 Fragment translation routines 6
2.3 Schematic view of DynamoRIO 7

3.1 Simulation of the open addressing collision mechanism 10
3.2 Internal fragment lookup routine 11
3.3 Fragment emission loop snippet 13
3.4 Shared Indirect Branch Lookup Routine 14

4.1 Inline Caching 16
4.2 Speculative Chaining 17
4.3 The Sieve 19
4.4 Fast Returns 20

5.1 API usage example 24
5.2 Inline Caching implementation 25
5.3 IBTC implementation 27

7.1 Native execution and emulation with DynamoRIO 37
7.2 IBTC and Inline Caching 38
7.3 Vanilla, Inline Caching, and IBTC 39
7.4 Hit path removed from IBTC 41
7.5 Basic blocks and Traces 42

Chapter 1

Introduction

1.1 Philosophical Approach

Software is the basis of modern society. It comes in many flavors: free or proprietary, in the form of binaries or source code, simply compiled or optimized, with or without graphical interfaces, interactive or autonomous. In any case, modern society depends on it for more efficient management of resources and relations, faster communication and the advance of science, just to name a few.

These are some of the aspects that make humanity free, as they allow us to control nature to a certain extent, overcome great distances in less or no time, and create our own version of what we call World. But the very technology that makes us free, also enslaves us, because the software that supports all these activities can only run on the hardware that it has been developed or compiled to.

Fortunately, our freedom is not entirely lost. Virtual machines are our allies in this battle against the evils of hardware dependency. With virtual machines, programs can run on computers that they have not been developed to. And although this ability usually penalizes efficiency, we can employ software techniques to improve it and achieve near-native execution performance.

1.2 Technical Approach

The key concept behind virtual machines is the act of emulation and the principal methods used in it are interpretation and dynamic binary translation (DBT). Whether a virtual machine should employ one or the other as emulation technique is a question of purpose. As a matter of fact, some virtual machines employ both methods.

Interpreters are simpler, more portable, and typically less efficient than dynamic binary translators [34]. In interpretation, instructions in a program are emulated one at a time, in a cycle that mimics the behavior of an actual hardware (*i.e.* instruction fetch, decode, and execution, all in software). On the other hand, in dynamic binary translation, instructions are analyzed and translated in chunks, then kept in memory – the "code cache" – for future re-execution, possibly speeding up the execution.

However, DBT systems can only garner better efficiency when the execution frequency of the translated code is high enough (*i.e.* the code is "hot") to amortize the initial costs of translation, which are higher than that of interpretation. And even if the translated code is indeed hot, other factors contribute to the loss of emulation performance. One of the greatest sources of overhead is the emulation of indirect branches.

Several techniques have been proposed to improve the emulation efficiency of indirect branches. In his Ph.D. thesis [8], Derek Bruening presents the virtual machine DynamoRIO and analyzes the impact on emulation efficiency caused by these techniques. In this master thesis we present the current implementation of DynamoRIO, we provide details about the way it emulates indirect branches, and we compare it to other techniques described in the literature.

1.3 Dynamic Binary Translation Commons

Virtual machines based on DBT translate instructions from the emulated application ("guest") in bundles. Possible choices for bundling include dynamic basic blocks, traces, and superblocks. We collectively refer to them as "fragments". Dynamic basic blocks are sequences of instructions that have a single entry-point and end in a single control transfer, such as branches and calls. Superblocks expand this definition by allowing multiple exit-points. Traces relax the constraints even further and allow both multiple entries and exits.

Whichever units of processing a DBT employs, the translator parses the guest code into fragments, translates, and saves them to the code cache. The complexity of the actual process of translation depends on the architectures and operating systems of the guest and the host and on their differences. However, all translators share the fact that control transfer instructions must be patched in order to keep control within the virtual machine.

As translated fragments are written to the code cache, the virtual machine transfers control to them so that they can be executed. Since exits have been previously patched, control always returns to the emulation manager. The emulation manager is the core of the virtual machine. It is responsible for switching execution between already translated fragments, fetching new chunks of guest code, and for translations *per se*.

Because guest code is discovered dynamically, fragments are stored in the code cache in an order that does not reflect the organization of fragments in the guest code. Thus, the emulation manager is responsible for keeping track of the locations of translated fragments and of the correspondences between them and the guest code. A map table provides this functionality and is implemented with hash tables in most virtual machines [34]. Whenever a fragment returns control to the emulation manager, it indicates the next address that would be executed should the program be running natively. The emulation manager then searches the map table for an existing translation and then decides if a new translation must be created or if an existing one may be used to continue execution.

1.4 Same-ISA Process Virtual Machines

User-level programs depend basically on two interfaces to execute: part of the instruction-set (ISA) and operating system calls. Process virtual machines provide these interfaces to programs. Potentially, the interfaces presented to the guest application are different than that used by the virtual machine itself, for instance, an emulator may run on top of an x86 computer with Windows and present the ARM ISA with Android for mobile application development.

But sometimes the purpose of a virtual machine is not to provide different interfaces for the execution of programs, but to optimize and provide introspection for the execution. DynamoRIO is one of such virtual machines developed for the x86 architecture. There are two available versions of DynamoRIO, one for the Windows operating system and other for Linux. In this master thesis we will use DynamoRIO to evaluate the efficiency of indirect branch emulation techniques.

Our objective is to determine which indirect branch emulation technique, or combination of techniques, provides the best emulation performance for the SPEC CPU2006 benchmark suite. This master thesis is organized as follows: Chapters 2 and 3 present the DynamoRIO infrastructure. Chapter 4 describes the functionality of several indirect branch emulation techniques, then Chapter 5 describes how we implemented some of the techniques in the code of DynamoRIO. Chapter 6 presents the related work. Finally, Chapter 7 presents the experimental results and Chapter 8 concludes the master thesis.

Chapter 2

An Overview of the DynamoRIO Infrastructure

DynamoRIO is a process virtual machine designed for the x86 architecture and is capable of emulating applications both on Windows and Linux. It employs dynamic binary translation, rather than interpretation, as its emulation technique. Unlike other dynamic binary translators (DBT), DynamoRIO spends little time translating from source to target architecture, since it is a same-ISA virtual machine. As a matter of fact, DynamoRIO copies and pastes most of the code that it reaches. As usual, translated code is kept in the code cache for future re-execution.

2.1 The Emulation Manager

The central hub of control flow in DynamoRIO is the "dispatcher". It is reached at the beginning of the execution and at every time that control leaves the code cache. Its main loop is responsible for looking up if the current fragment of code that must be executed has already been translated, or if it still needs to be. A couple of helper routines perform this task, and if they fail to locate a fragment, they translate a new one. In either case, the dispatcher then transfers control to the translated fragment in the code cache. No further code is reachable in the dispatcher, and it will only be invoked again in the event of a code cache exit. Figure 2.1 presents a simplified version of the dispatcher, in which the fragment lookup, translation, and control transfer routines are located. It also contains a routine that monitors the execution frequency of each fragment of code.

```
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 2.1: Dispatcher: the central hub of DynamoRIO

2.2 Fragment Lookup

Upon entry, the dispatcher calls `fragment_lookup_fine_and_coarse`, the lookup routine. In DynamoRIO, information about every fragment that has ever been translated is kept in central hash tables. Each entry in a table is composed of a guest and a translated address, which form a pair of corresponding targets for control transfer instructions. The guest address is used as input to the hashing function and as tag. When a match occurs, the lookup routine returns the address of the corresponding translated fragment. Otherwise, it returns an invalid address indicating that the fragment of code has never been reached and that a new translation must be created for it.

2.3 Translation

Whenever the dispatcher is unable to find a translation for some piece of code, it starts the creation of a new one. The function `build_basic_block_fragment` accomplishes this action, as shown in Figure 2.1. The creation of the new translation begins with the partial decoding of the instructions in the guest application and ends whenever a control transfer instruction is reached. Since x86 has variable-length instructions, the decoding consists of determining its *opcodes* and lengths. No further information, such as operands, needs to be decoded. Only the raw bits are stored in the internal structures of DynamoRIO.

Every decoded instruction is then added to a linked list that represents the fragment of code being translated. Even the last instruction – the control transfer – goes into the list without patching. Figure 2.2 summarizes all the steps involved in the creation of

```
1  build_bb_ilist(dcontext_t *dcontext, build_bb_t *bb) {
2      while (true) {
3          if (check_for_stopping_point(dcontext, bb))
4              break;
5          if (bb->full_decode) {
6              bb->cur_pc = decode(dcontext, bb->cur_pc, bb->instr);
7              instrlist_append(bb->ilist, exit_instr);
8          } else
9              bb->cur_pc = decode_cti(dcontext, bb->cur_pc, bb->instr);
10     }
11     if (!bb->full_decode) {
12         non_cti = instr_create(dcontext);
13         instr_set_raw_bits(non_cti, non_cti_start_pc, ..);
14         instrlist_append(bb->ilist, non_cti);
15     }
16     bb->exit_target = get_ibl_routine(dcontext, ...);
17     instr_t *exit_instr = INSTR_CREATE_jmp(dcontext, bb->exit_target));
18     instr_set_our_mangling(exit_instr, true);
19     instrlist_append(bb->ilist, exit_instr);
20     client_process_bb(dcontext, bb);
21     mangle_bb_ilist(dcontext, bb)
22 }
```

Figure 2.2: Fragment translation routines. The translation of guest code happens in several phases. This segment of code presents the decoding of instructions from the guest into the intermediate representation.

a new fragment, apart from the last two instructions, which pertain to the patching of control transfer instructions.

2.4 Patching

Patching happens potentially several times. First, the list is delivered to clients registered through the API of DynamoRIO (`register_bb_event` and `register_trace_event`) to handle the event of a fragment creation. There may be zero or more clients registered to handle these events. After every registered client is given a chance to observe and modify the instruction list, DynamoRIO calls its own patching routine, `mangle_bb_ilist`.

The mangling of control transfer instructions guarantees that the guest code is never executed. For direct branches, DynamoRIO uses the block chaining technique [34], which modifies all the targets of the branches, so that they always transfer control to translated fragments of code in the code cache, or back to the emulation manager. Indirect branches, on the other hand, have unlimited number of targets and may not be patched with the block chaining technique. There are several types of such indirect control transfer, such as

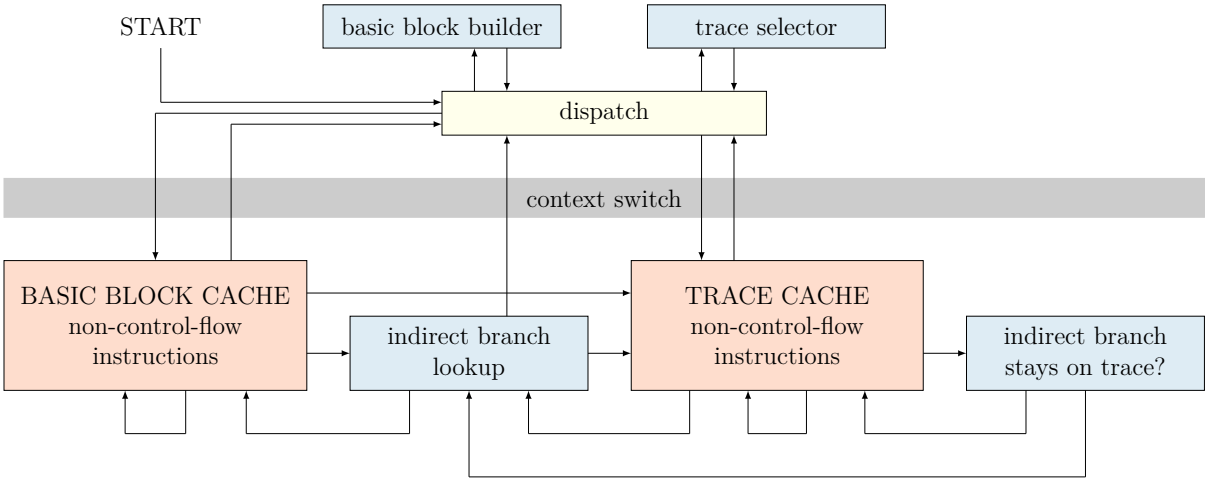


Figure 2.3: Schematic view of DynamoRIO

register-indirect branches, returns, and indirect calls. For each of them, DynamoRIO uses customized routines to determine where control must be transferred to. These routines are actually very similar to each other, in the sense that they receive a guest application address and, in order to convert it to a code cache address, they access a hash table of addresses correspondences.

2.5 Dispatch

Whether a new fragment had to be created, or had been found by the lookup routine, the dispatcher transfers control to it. Potentially, control will reach the dispatcher again, but as a new call. This process repeats until there is no more need to exit the code cache (for instance, if the guest program enters an infinite loop) or until the emulation is terminated.

Figure 2.3 presents a schematic view of the components in DynamoRIO. It graphically delineates the border between the code cache and the emulator. The components above the gray block pertain to the DynamoRIO code base, whereas those below, pertain to code emitted into the code cache.

2.6 Fragment optimization

Initially, DynamoRIO translates dynamic basic blocks. However, it also monitors the execution of each block in order to identify regions of “hot-code” and further optimize them. Optimization is achieved with the creation of larger segments of code, known as

System type	**crafty**	**vpr**
Basic emulation	300.0x	300.0x
+ basic block cache	26.1x	26.0x
+ link direct branches	5.1x	3.0x
+ link indirect branches	2.0x	1.2x
+ traces	1.7x	1.1x

Table 2.1: Breakdown of the slowdown over native execution

superblocks in the literature, but referred to as traces in the parlance of the DynamoRIO community.

The trace building mechanism in DynamoRIO starts by marking certain basic blocks as potential trace heads. Each of them receives a counter that is incremented upon each execution of that block. When a threshold is reached, that block and every subsequent block executed is added to the new trace, until an end-of-trace condition is reached [10].

Table 2.1 [10] depicts how the code caching, control transfer optimizations and trace formation reduces the overhead of DynamoRIO emulation. When none of them are used, the slowdown over native execution reaches a factor of several hundred. The use of a fragment cache reduces the slowdown significantly. The linking of fragments and the use of traces bring the execution times close to native execution.

Chapter 3

A Detailed View of DynamoRIO 4.1

DynamoRIO is a currently active open-source project, and still receives several contributions from its community of developers. In our project, we used DynamoRIO version 4.1, which was the current version, by the time of the writing of this Thesis. This chapter details the indirect branch handling mechanisms present in such version.

3.1 Hash Tables

As stated in Section 2.2, DynamoRIO maintains the correspondences between guest and host addresses in hash tables. Shared hash tables keep track of the fragments shared among threads in the emulated application, whereas private tables keep track of private translations for each thread. Initially, every entry in the hash tables is empty. This means that the composing fields `tag_fragment` and `start_pc_fragment` are set to `NULL_TAG` (0) and to `HASHLOOKUP_NULL_START_PC`, respectively. The emulation manager fills the entries with newly created fragments. But as it translates more fragments, the table starts to fill up and entries start to collide. In order to avoid excessive collisions or capacity issues, DynamoRIO doubles the size of the table, whenever the occupation rate reaches a defined threshold.

Collisions are handled with the *open addressing* technique, *i.e.* colliding elements are stored in the hash table itself, not in a linked-list. The insertion of an element in the table begins with the generation of an index with a hash function. Afterwards, the table is probed to check if that entry is empty or if it has already been occupied. If it is indeed empty, the new element is simply added to the table, otherwise the index is successively increased, until an empty slot is found. The increment function verifies if the successive addresses do not overlap the boundaries of the table, and wrap-around if they do.

Figure 3.1 illustrates the insertion of a sequence of elements that cause a collision and a wrap-around. Let the hash index be the modulo operation between the tag and the

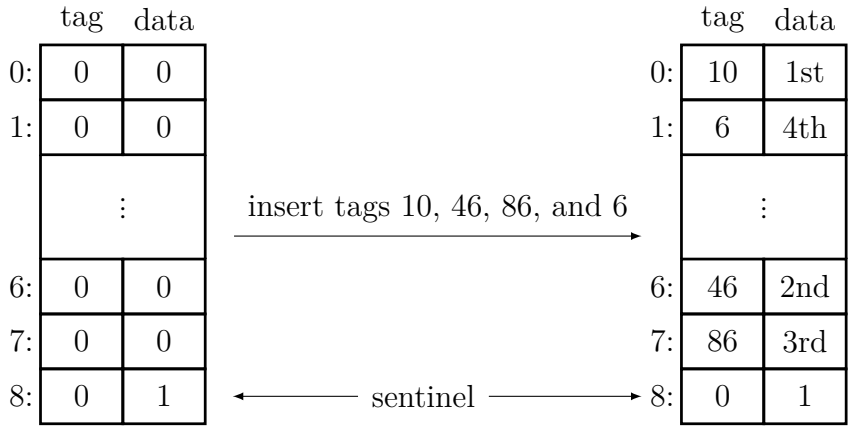


Figure 3.1: Simulation of the open addressing collision mechanism

constant number 10, and let the order of insertion of elements be 10, 46, 86, and 6. The first and second insertions do not cause collisions, as they are mapped to the previously empty slots 0 and 6, respectively. The third and fourth insertions do cause collisions. The open addressing technique walks through the collision chain and inserts the items in the first empty slots that it finds, *i.e.* slot 7 and, after wrapping around the end of the table, slot 1.

3.2 Fragment Lookup

In DynamoRIO, fragments may be either formed as dynamic basic blocks, or as the structures described as superblocks in Section 1.3. However, the developers of DynamoRIO refer to the superblocks as traces. In order to avoid misunderstandings, we will use their terminology throughout this master thesis. Separate hash tables are used to avoid unnecessary collisions. Moreover, fragments might be private to a single thread, or shared among them all. Therefore, the fragment lookup routine must search for the fragments in all hash tables, according to the following precedence: private traces, private basic blocks, shared traces, and, finally, shared basic blocks.

The process of looking up a fragment in the hash tables is straightforward. Figure 3.2 presents its code as it appears in DynamoRIO, with only few simplifications to make it more clear. The code begins by applying the hash function and fetching an entry from the table, in lines 7 and 8. If the tags match, the routine returns immediately. Otherwise it walks through the collision chain until the tags match or until it finds an empty entry, because empty entries indicate that the chain has ended.

Whether a hit occurs or not, a `fragment_t` structure is always returned. The calling routine – *i.e.* the dispatcher – analyzes the returned object. An empty entry indicates

```
1 static fragment_t
2 hashtable_fragment_lookup(dcontext_t *dcontext, ptr_uint_t tag,
3                            fragment_table_t *htable) {
4     fragment_t fe;
5     uint hindex;
6
7     hindex = HASH_FUNCTION(tag, htable);
8     fe = htable->table[hindex];
9     while (!ENTRY_IS_EMPTY(fe)) {
10        if (tag == fe.tag)
11            return fe;
12        hindex = WRAP_AROUND(hindex + 1, htable);
13        fe = htable->table[hindex];
14    }
15    return fe;
16 }
```

Figure 3.2: Internal fragment lookup routine. This code is located in the template file `hashtablex.h`. We simplified it by removing code used for debug and statistics, and by replacing template names properly.

that the lookup routine failed to find a translation, thus the dispatcher must start the creation of a new one. Non-empty entries provide the dispatcher with the host address of the translated fragment.

3.3 Translation

The translation of fragments happens in several phases: first, instructions are decoded from the guest code, converted to an intermediate representation (IR), and stored in instruction lists; afterwards, the instructions are patched by registered clients and by DynamoRIO itself; finally, the patched instructions are emitted into the code cache. We shall cover all but the patching of instructions in this section.

A loop performs the decoding of instructions from guest code into the IR. In each iteration, the loop checks if the fragment must be terminated, or if it can grow further. Several conditions are verified and force the decoding to stop, such as when too many instructions have been added to the list, or when a control transfer or invalid instruction is reached.

The intermediate representation of instructions in DynamoRIO comprises several levels of codification [10]. Level 0 stores the minimum amount of information. As a matter of fact, it stores only the raw bits of a sequence of instructions. Level 1 includes information

about the boundaries between instructions, whereas Level 2 includes both *opcode* and *eflags* information. Operands are only decoded in Levels 3 and 4.

When DynamoRIO decodes instructions from the guest code, it tries to keep the amount of decoded information to a minimum. Non-control-flow instructions are stored as a single strip of raw bits (Level 0) in the list. On the other hand, control transfer instructions are decoded up to Level 3, and appended as a single entry in the list. This behavior can be modified, through the use of the flag `full_decode`, so that every instruction is fully decoded (see Figure 2.2).

3.4 Patching

When DynamoRIO finishes the decoding of the fragment, it calls registered clients, so that they have an opportunity to observe and modify the instruction list. Afterwards, DynamoRIO applies its own patching routine to the list. As a matter of fact, in this step DynamoRIO cares only about control transfer instructions. Direct and indirect calls, far and indirect jumps, system calls, interruptions, and returns, each have a specialized mangling function. These functions are named in a systematic manner, by adding the prefix `mangle_` to the type of instruction that it patches. For instance, interruptions are mangled by `mangle_interrupt`.

When mangling system calls and interruptions, DynamoRIO only checks if their type is supported. If it is not, DynamoRIO simply removes them and uses a basic interpretation scheme. Direct calls are converted into a push of the return address. Indirect transfer of control – *i.e.* returns, indirect calls, and indirect jumps *per se* – are removed and have their target address stored in the register `%ecx`. This address will be used by the appropriate routine described in Section 3.6.

3.5 Emission

During code emission, DynamoRIO iterates over the instructions in the instruction list and emits them to the code cache, as executable code. Figure 3.3 presents the function `set_linkstub_fields`, which performs the loop. For every instruction in the list, it calls `instr_encode`, which actually writes the executable bytes to the code cache. The function returns a pointer to the memory address where subsequent emissions should be placed on.

```
1 cache_pc set_linkstub_fields(*dcontext, *fragment, *ilist, ..) {
2     cache_pc pc;
3     instr_t *inst;
4
5     pc = FCACHE_ENTRY_PC(fragment);
6     for (inst = first(ilist); inst; inst = next(inst))
7         if (instr_ok_to_emit(inst))
8             pc = instr_encode(dcontext, inst, pc);
9     return pc;
10 }
```

Figure 3.3: Fragment emission loop snippet. Features the function `instr_encode`, which is the function that actually writes the executable bytes to the code cache.

3.6 Indirect Branch Lookup Routines

So far, we have 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, fragment lookup is critical to the execution performance [8], thus it has an alternate implementation in DynamoRIO. Instead of invoking the dispatcher, the runtime places fast, specialized, lookup routines inside the code cache, enabling fragments to directly transfer control to other fragments even when emulating indirect branches.

Figure 3.4 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 3.1. 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`.

```
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.

Chapter 4

Software Techniques for Indirect Branch Emulation

On lowly optimized virtual machines, fragments of translated code always return to the emulation manager when they reach their ends. They do not try to translate any guest address into a host address from within the code cache, instead they rely on the emulation manager. On the one hand, this is the simplest mechanism to handle address translation, on the other hand, it is also the slowest. Highly optimized dynamic binary translators implement more advanced techniques of indirect branch emulation. The next sections describe these techniques.

4.1 Indirect Branches in Static Translators

Virtual machines based on static binary translation, such as VEST and MXR [33], rely on runtime support to translate indirect branch targets. During the static translation, indirect jumps are converted into calls to the interpreter. These virtual machines continue emulation using interpretation, until they reach a point in the program, for which there is a known translation. Then, they transfer control back to the translations, avoiding the low performance associated with interpretation. This process repeats indefinitely.

4.2 Inline Caching

The Inline Caching technique, also known as Indirect Branch Inlining [21], or Software Indirect Jump Prediction [34] replaces any indirect jump instruction with a sequence of tests that compare the target address of the jump with previously known targets – for which a translation already exists in the code cache. When a hit occurs, control is

```
1 mov ecx, <target>        ; copy target to ecx
2 cmp ecx, <prediction 1>  ; compare ecx with a predicted target
3 je <translation 1>       ; jump to the equivalent translation
4 cmp ecx, <prediction 2>  ; compare ecx with another predicted target
5 je <translation 2>       ; jump to the equivalent translation
6 jmp <emulation manager>  ; return to the emulation manager
```

Figure 4.1: Inline Caching

transferred to the equivalent translated fragment. Only when every test fail, control is delivered to the emulation manager. Figure 4.1 illustrates the technique. The amount of comparisons in the sequence varies across implementations.

Typically, the most frequent targets are inlined, thus the virtual machine must rely on a profiling mechanism. The profiling phase might happen while code is being interpreted, such as in virtual machines that use both interpretation and translation, or it might happen after translation. In the later case, the already translated fragment needs to be patched when the profiling threshold is reached.

Several authors [10,12,21,36] compare this technique to the Inline Caching mechanism developed for the object-oriented language Smalltalk-80. In object-oriented languages it is not always possible to determine, at compile time, to which class an object belongs, neither which implementation of a method it should call. Deustch [15] describes how Inline Caching finds the correct address of the implementation of a method in a class, during runtime.

4.3 Speculative Chaining

The Speculative Chaining technique has few differences when compared to the Inline Caching (Section 4.2). During translation, every indirect branch is simply converted into an unconditional jump to the translated fragment of a known target of that branch. Since it jumps speculatively, *i.e.* to a destination that is not guaranteed to be the correct one, the target fragment must assert that the speculation holds. The verification is performed by a comparison between the target address and a constant stored as immediate in the code. Figure 4.2 illustrates the process.

This behavior is similar to the branch predictor in modern computer architectures, in the sense that it tries to guess the destination address, before asserting that it is correct. Branch predictors take advantage of this eager behavior by being able to fetch code from the instruction memory earlier, whereas the Speculative Chaining technique removes indirect jumps from the code, potentially rendering it better guessable by the branch predictor in the underlying hardware.

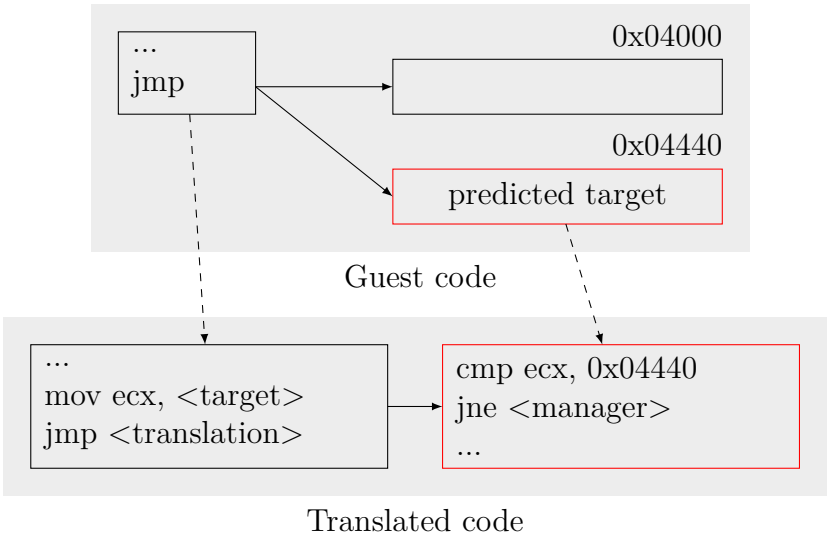


Figure 4.2: Speculative Chaining

4.4 Code expansion

Daisy [18] is a virtual machine based on binary translation that uses a unique technique to emulate indirect branches. For each byte in the guest application, Daisy reserves N bytes in memory to hold its translation. For instance, when $N = 4$ and a guest fragment is composed of 12 bytes, the translation of that fragment will have 48 bytes available to be fitted in. In order to convert a guest address into a host address, Daisy can use Equation 4.1, where n is the target address of the jump, N is the expansion constant and $VLIW\_BASE$ is a pointer to the beginning of the code cache. The ability to calculate the translation address with a formula avoids all the overhead associated with address lookups and the maintenance of map tables.

$$Address = n * N + VLIW\_BASE \tag{4.1}$$

To the extent of our knowledge, no other virtual machine used code expansion. As a matter of fact, even Daisy abandoned this idea in 2001 [17], on behalf of the more conventional code cache organizations.

4.5 Indirect Branch Translation Cache

In the Indirect Branch Translation Cache (IBTC) technique, every fragment that ends with an indirect jump is equipped with a small hash table of mappings between guest

and host addresses. It is a cache of the global map table and holds only the particular addresses that have been accessed by the indirect jump in the fragment. This arguably reduces the collision rate of the hashing function, possibly leading to faster execution times. Moreover, the hash table, though small, may also handle collisions. Whether it uses linked-lists, open-addressing, or other collision handling mechanisms, depends on implementation choices.

The indirect jumps themselves are converted into segments of code that compare the target of the jump with the guest address in the table. When they match, control is transferred to the equivalent host address. Otherwise, the collision chain is iterated over until a match occurs or until the end of the chain is reached. Only when all the comparisons fail, the IBTC technique gives up and falls back to the emulation manager.

Since the IBTC is a cache with limited size, indirect jumps with sufficient targets might hit the maximum capacity of the table. When that happens, and control is transferred back to the emulation manager, a translation might still exist for the target fragment. Thus, the dispatcher searches the global map table. Only when the global search fails, the dispatcher starts the translation of a new fragment.

4.6 Sieve

The Sieve can be thought of as a technique that uses instructions, rather than data memory, to store the mappings of guest to host addresses. During translation, indirect branch instructions are converted into jumps to chains of *sieve buckets*, which are tiny segments of code responsible for comparing the target address of the indirect jump with constants stored as immediate in the buckets themselves. When the addresses match, execution control is transferred to the equivalent translated fragment, which have its host address stored also as immediate in the code of the buckets.

Sieve buckets are arranged in chains, dynamically allocated as linked-lists. When the address comparison in a bucket fails, the next bucket receives the control of the execution. This process continues until a translation is found, or until the end of the chain is reached. As usual, when the later happens, control is transferred to the emulation manager so that it decides whether a translation already exists but could not be found by the technique, or whether a new translation must be created.

The virtual machine maintains several sieve chains, which are selected by the result of a hash function between the guest address and a predefined mask. Figure 4.3 illustrates the process and structures related to the Sieve. This behavior is similar to the hash tables used by IBTC and by the central map tables of DynamoRIO. The differences reside in the facts that: the Sieve handles collisions with linked-lists, rather than with open-addressing; it stores data as code; and it converts indirect jumps into direct branches,

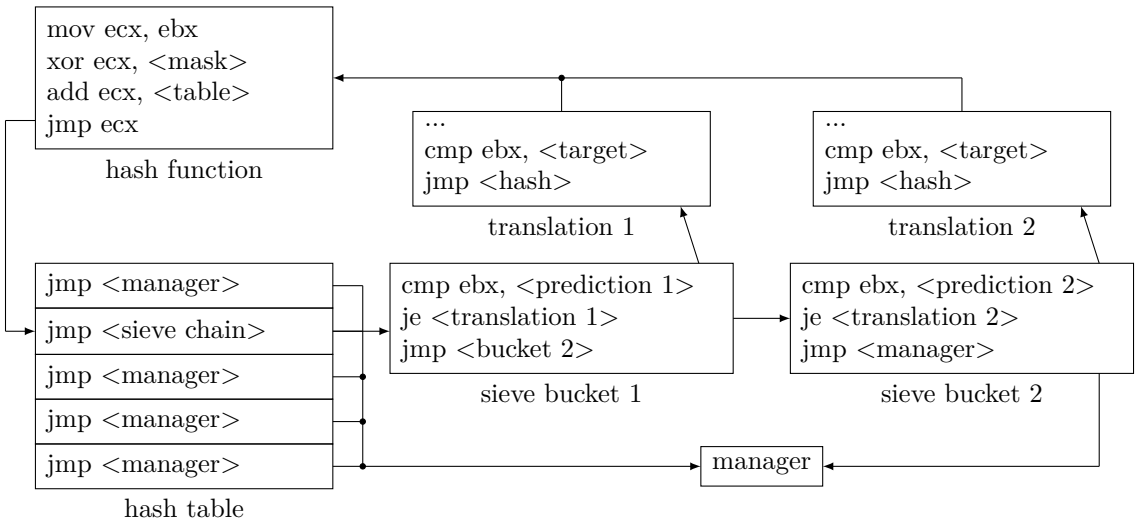


Figure 4.3: The Sieve

potentially rendering it easier for the branch predictor, on the underlying hardware, to guess correctly.

4.7 Fast Returns

The Fast Returns technique handles the specific case of the emulation of return instructions. Typically, return instructions transfer the execution control to the address previously saved by the corresponding call instruction. This is a guest address, so the return instruction cannot jump directly, instead it should translate and jump to a host address, in the code cache. With Fast Returns, call instructions are modified so that they store the address of the translation, *i.e.* a host address, instead of the guest address. This allows of return instructions to be left untouched, during translation. Figure 4.4 illustrates the technique.

This mechanism benefits performance, because it removes the necessity of an address lookup. But it poses an issue to the emulation of the program, because it changes the contents of the guest program memory and might modify the behavior of the execution, thus violating transparency, and potentially leading to wrong results.

4.8 Shadow Stack

The Shadow Stack technique handles the emulation of return instructions, while still maintaining transparency. During translation, call instructions are converted into segments of code that push the guest return address into the program stack, and also push

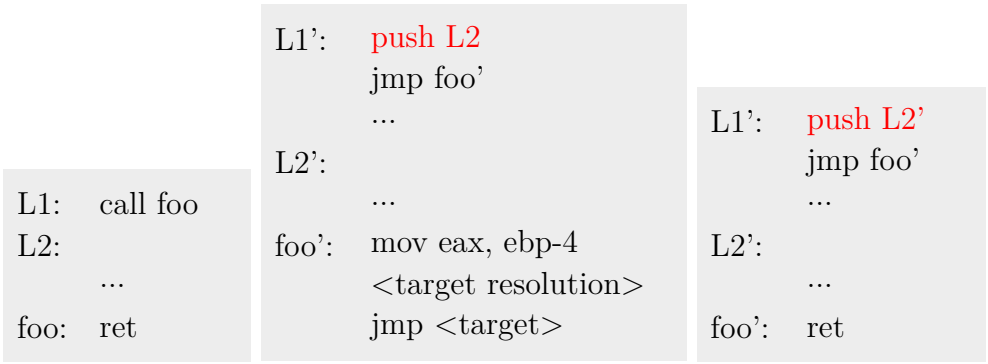


Figure 4.4: Fast Returns

the same guest address along with the host address of the corresponding translation, into an alternative stack – the *Shadow Stack*. Since these steps do not touch data from the application, this technique maintains transparency, avoiding errors in the execution of the program.

On the other hand, return instructions cannot be left untouched. They are converted into segments of code that pop an entry from the Shadow Stack, verify if the target of the return instruction and the entry obtained from the stack match, and jump to the corresponding translation when they do. When the addresses do not match, the emulation falls back to the emulation manager, or to other indirect branch emulation technique.

Hazelwood and Klauser [20] describe the same technique, but name it differently, as software RAS. Hiser *et al.* [21] also give an alias to the technique, RATS, for Return Address Translation Stack.

4.9 Return Cache

The Return Cache technique maintains a hash table of recently used return targets, in order to exploit the regularity of return instructions. During translation, call instructions are converted into segments of code that store, into the hash table, the host address of the return target, *i.e.* the address of a translation in the code cache. Afterwards they jump to the translated body of the function. Return instructions are also modified so that they jump to a host address loaded from the hash table, unconsidered of the validity of the translation.

The hash table is indexed by a function of the call target, rather than of the return target. Thus, multiple call points to the same function update the same entry with their

corresponding return site addresses. Return instructions in the function access the same single entry in the hash table.

Since the return address of a function can change between the call and the return, such as in recursive calls and in some *glibc* functions, a validation code is added to the return site. The validation code compares the guest address obtained during execution against a constant stored as immediate. The constant represents the guest address that should have been taken if no changes to the return address had occurred. If the addresses match, execution may continue, otherwise, a backup mechanism must be used. Usually, this means falling back to the emulation manager.

4.10 Indirect Branches in DynamoRIO

Bruening showed in his PhD thesis, that DynamoRIO spends only 4.5% of the time in routines related to the translation of guest to host addresses. In order to do so, DynamoRIO uses optimized lookup routines to translate the target guest address of indirect jumps into host addresses in the code cache. During the translation of a fragment, indirect jumps are converted into segments of code that store the target address of the jump into the register `ecx`, then transfer control to the appropriate lookup routine. Section 3.6 describes how the optimized routines look a target up in the mapping tables. They convert the target address into a hash index, check for a match, and iterate over the collision chain when the check misses. When a hit occurs, the routines transfer the execution control to the target fragment in the code cache. Otherwise, they fall back to the dispatcher.

Each type of indirect branch, *i.e.* returns, indirect calls, and register-indirect jumps, have a specialized routine. So do each type of fragment, *i.e.* basic blocks and traces, and each level of thread-awareness, *i.e.* shared and private. Thus, there are several lookup routines. However, these routines only differ in the fact that each of them searches in a distinct hash table.

The IBTC is similar to this technique in two aspects. First, they use hash tables to store the mappings of addresses. Second, they can handle index collisions in the hash function. On the other hand, they differ in two aspects. First, the lookup routines of DynamoRIO are shared among indirect branches, thus leading to less code being emitted to the code cache, whereas in the IBTC, the translation of each fragment has its own inlined address resolution code. Second, the IBTC allocates extra memory for each indirect branch, whereas in DynamoRIO, a single table is shared by several fragments.

The potential advantage of the IBTC technique is that collisions might be less frequent, due to the fact that each indirect branch has its own hash table, and provided that the tables are large enough to keep the collision rate low, though they may be smaller than the global table. This potentially leads to better performance, since hits in the lookup

Technique	**Virtual Machine**	**Class**
Emulation Manager	VEST, MXR, Shade, FX!32, UQDBT, Bintrans, QEMU	Generic
Inline Caching	Daisy, Dynamo	Generic
Speculative Chaining	Embra, Walkabout	Generic
Code expansion	Daisy	Generic
IBTC	Strata, Pin	Generic
Sieve	HDTrans, Pin	Generic
Fast Returns	Strata	Return-specific
Shadow Stack	FX!32, Pin	Return-specific
Return Cache	HDTrans	Return-specific
DynamoRIO's	DynamoRIO	Generic

Table 4.1: Indirect branch emulation techniques, their use throughout the literature, and their classification. A generic technique is capable of handling any type of indirect branch, whereas return-specific techniques may only be used in the emulation of return instructions

routines happen faster. On the other hand, DynamoRIO consumes less memory, which might also lead to better performance, since it produces less pressure in the processor cache. Chapter 7 analyzes these trends.

Table 4.1 summarizes the techniques described in this chapter and correlates them to the virtual machines presented in Chapter 6.

Chapter 5

Implementation of the techniques

Section 3.4 presents the points where code patching happens. Clients may register themselves to receive the opportunity to modify the code on the event of fragment creation. When they do, DynamoRIO calls the registered routines before applying its own patches. Ideally, we would insert our indirect branch emulation techniques using a client, but we need access to functions from the DynamoRIO code base that are only accessible from within DynamoRIO itself. Therefore, we apply our modifications right after the clients return. The API of DynamoRIO provides a rich set of functions to ease code modification. Five types of functions are particularly useful for our implementation:

Instruction decoding functions (`instr_get_target` and `instr_get_src`) ease the parsing of jump targets. Both return the first operand of a jump instruction, which is its target.

Instruction creation macros enable the creation of new instructions from scratch. They receive the machine state as a parameter, followed by the list of desired operands. For instance, the macro `INSTR_CREATE_add` creates a new `add` instruction.

Operand creation macros facilitate the creation of operand structures. These are particularly useful, because they obviate the need to know details about immediate and memory pointer representation on the x86 architecture. For instance, the macro `OPND_CREATE_INTPTR` automatically determines how many bits are required to represent an immediate.

Instruction list handling functions make it easy to iterate over the instruction lists that DynamoRIO uses to form basic blocks and traces. They provide methods to get the first and last instruction in a list, to append instructions and to remove them.

```
1 instr_t *instr;
2 opnd_t target;
3 opnd_t immed;
4
5 instr = instrlist_get_last(ilist);
6 target = instr_get_target(instr);
7 immed = OPND_CREATE_INTPTR(1);
8 dr_insert_clean_call(ilist, instr, routine, 2, target, immed);
```

Figure 5.1: API usage example.

Clean calls enable the use of code written in C while still executing code from within the code cache. They prepare the calls by saving the processor context and loading a new stack pointer. Afterwards they call the desired routines, and upon return, restore the processor context.

Figure 5.1 shows an example of the use of the API routines. In it, we parse the target of an instruction, create an immediate operand, and insert a clean call to a routine in C.

The implement of each emulation technique uses these functions and macros, and they are contained in a single pair of code (.c) and header (.h) files. As a matter of fact, they are all implemented in the same function, and selected through the use of preprocessor conditionals. In this project, we evaluate two indirect branch emulation techniques: Inline Caching and IBTC.

5.1 Inline Caching

The Inline Caching technique, described in Section 4.2, has two implementation parameters: the amount of tests inlined in the code and whether translation happens before or after the profiling of targets. In this project, we set the number of tests to one and translation to happen before profiling. As a matter of fact, since DynamoRIO never interprets guest code, translation must happen before profiling.

Keeping the amount of tests to its minimum has one major advantage: the reduced time spent in profiling when compared to multiple tests. Bala *et al.* [3] reduce the effort applied to profiling with a scheme referred to as MRET (most recently executed tail) that works as follows. Each loop head in the program is classified as a profiling point and receives a counter. Every time that point is executed, the counter is incremented. When a threshold is reached, the current state of the program is said to be hot, *i.e.* frequently executed. The key concept behind the idea, is that when a path becomes hot, it is statistically likely that the previous and next iterations were and will be hot, as well.

```
1 init:
2         <save state (6 intructions)>
3         mov     %ebx, %edi
4         cmp     %ebx, (GUEST_SLOT)
5         jne     profile
6 hit:
7         <restore state instructions (6 instructions)>
8         jmp     (HOST_SLOT)
9 profile:
10        cmp     (COUNTER), $THRESHOLD
11        ja      stop
12        jb      continue
13        call    update      ; clean call to update, a function in C.
14 continue:
15        inc     (COUNTER)
16        mov     (LAST), %ebx
17 stop:
18        <restore state (6 instruction) and fall back>
```

Figure 5.2: Inline Caching. Assembly code generated by the translation of a jump instruction. The original jump used register %edi as operand. The clean call is actually composed of 43 instruction, apart from the C routine itself.

We extend this concept to our implementation, by adding a counter to the translation of each indirect jump. Once the counter reaches a threshold, we use the last seen target of the jump as a hot target, and we update the Inline Cache with the guest and host pair of addresses. Figure 5.2 shows the assembly code generated by the translation of an indirect jump.

5.2 IBTC

The IBTC technique, described in Section 4.5, consists of several hash tables, one for each fragment, and of code that searches and updates these tables. The size of the table, as well as the method used to handle collisions in the hash function, are implementation dependent. As a matter of fact, collision handling is optional.

In our implementation, we set the cache size to 32 slots, and we handle collisions with the open-addressing technique [14], in exactly the same way that DynamoRIO does with its global hash tables of mappings, as described in Section 3.1.

Figure 5.3 shows the assembly code generated by the translation of an indirect jump with the IBTC technique. In label `init`, the target of the jump is loaded into the registers `ebx` and `ecx`. Afterwards, the hash index is calculated, based on the target address. Finally, the IBTC table base is added to the hash index and stored into register `ecx`. Label

`retry` checks if the jump target matches the current entry in the table, then transfers control accordingly. When a hit occurs, the virtual machine state is restore and control is transferred to the translation of the targeted fragment, which is stored in `0x4(%ecx)`, *i.e.*, the next word in the hash table. When a miss occurs, the code in label `miss` checks if the chain is over, and falls back if it is. Otherwise, it must iterate over the collision chain. Label `used` increments the table pointer and loops back to the label `retry`. Label `unused` checks if the table itself is over, by checking if the entry is the sentinel, in which case it also loops back to the beginning of the table and retries.

```
1  init:
2          <save state (6 intructions)>
3          mov     %ebx, %edi
4          mov     %ecx, %ebx
5          and     %ecx, $MASK
6          shl     %ecx, 3
7          add     %ecx, $TABLE_BASE
8  retry:
9          cmp     %ebx, (%ecx)
10         jne     miss
11 hit:
12         mov     %ebx, 0x4(%ecx)
13         mov     %ecx, $HOST_SLOT
14         mov     (%ecx), %ebx
15         <restore state instructions (6 instructions)>
16         jmp     (HOST_SLOT)
17 miss:
18         cmp     (%ecx), $0
19         je      unused
20 used:
21         add     %ecx, $8
22         jmp     retry
23 unused:
24         cmp     0x4(%ecx), $1
25         jne     fallback
26 sentinel:
27         mov     %ecx, $TABLE_BASE
28         jmp     retry
29 fallback:
30         call    update      ; clean call to update, a function in C.
31         <restore state (6 instruction) and fall back>
```

Figure 5.3: IBTC. Assembly code generated by the translation of a jump instruction. The original jump used register %edi as operand. The clean call is actually composed of 43 instructions, apart from the C routine itself.

Chapter 6

Related Work

Sites *et al.* [33] describe VEST and MXR, static binary translators that rely on runtime support for the resolution of indirect branch targets. When an indirect jump is reached during the execution of the translated code, a lookup is performed in the static table of address mappings. If the address is found, control is directly transferred to the corresponding host target, which has been previously and statically translated. Otherwise, the emulation continues using the interpretation technique, until it reaches a point in the guest code to which a translation is known. The mechanisms used by the address lookup are not clearly described in the paper.

Cmelik and Keppel [13] present Shade, a virtual machine for code translation and introspection. It allows the user to monitor the execution of selected types of instructions, and to select the level of detail the monitoring should be performed. Shade employs dynamic binary translation to emulate guest code, and uses basic blocks, rather than traces or superblocks, as its unity of translation. To each basic block, Shade adds a prologue and an epilogue. The prologue is responsible for code introspection, whereas the epilogue, for the chaining of basic blocks ended with direct branches. However, indirect jumps always transfer control back to the emulation manager.

Bedichek [5,6] presents Talisman, a system virtual machine that uses interpretation as its emulation technique. The main goal of Talisman is to model the memory management unit (MMU) of processors, thus it keeps track of memory pages, as well as it handles the conversion from virtual to physical addresses. Talisman pre-decodes pages, thus when a control transfer instruction is emulated, the target address must be monitored in order to determine whether the jump stays on the same page, or if it crosses pages boundaries. When a direct jump stays on the same page, it is said to be an *on-page* branch. During pre-decoding, *on-page* direct branches have their target address converted into a decoded target. *Off-page* direct branches, on the other hand, does not receive similar treatment, because Talisman must also verify if the targeted page is present in memory. Since the

target of indirect jumps in unknown during pre-decoding, they are treated as *off-page* branches. Talisman keeps a target address cache for faster resolution of *off-page* branch targets, which is flushed whenever a modification happens to the page tables.

Witchel and Rosenblum [39] present Embra, the first system virtual machine to employ dynamic binary translation. The authors observed that return instructions were responsible for a large fraction of the total register-indirect branch count, and that the register values were often the same, across executions. Thus, Embra uses the *Speculative Chaining* technique, which chains blocks ended with indirect jumps as if they ended with unconditional direct branches. Since the chaining is based on speculation, a validation code is added to the target block which determines if the speculation was correct. If it fails, the emulation manager of Embra receives the control of the execution for correct address resolution.

Ebcioğlu and Altman present Daisy, a dynamic binary translator capable of translating from several architectures to a VLIW machine. It is a system virtual machine that efficiently handles interruptions. In its first version [18], the basic unity of translation was a virtual memory page, which was translated into a region of memory four times larger than the source. This allows of the use of the Code Expansion technique, described in Section 4.4. Nonetheless, in 2001, the authors presented the new version of Daisy [17], which replaced the page as unity of translation with the more conventional basic block. Moreover, the new version abandoned the concept of Code Expansion and adopted the Inline Caching technique with multiple, update-able, comparison tiers.

FX!32 [11, 22] is a binary translator that incrementally converts x86 binaries into Alpha code. On the first time that an application is executed, FX!32 uses only the interpretation technique. Meanwhile, it monitors the execution and generates profiling logs. When the application is terminated, a resident process of FX!32 reads the log file, which contains the addresses of executed basic blocks, and translates these blocks into native Alpha code. This method does not guarantee that the entire code of the emulated application is covered by translation, but it reduces the amount of code that must be interpreted in future executions. Even though the targets of indirect branches do get profiled and translated, FX!32 does not employ advanced techniques for regular register-indirect branch target resolution, and always falls back to the emulation manager. On the other hand, return instructions do have a special target resolution technique based on the fact that procedure calls on the x86 and Alpha architectures behave in orthogonal ways. Call instructions are converted into segments of code that push the return address onto the program stack, in exactly the same way that the x86 hardware does, then jump to the translated routine using the native call of the Alpha architecture. On Alpha computers, the call instruction saves the return address into the `ra` register, which can then be used by the return instruction. However, some applications modify the return address of a

procedure call, but the modification is only visible on the program stack and not in the `ra` register. In order to avoid corrupted execution, FX!32 uses the Shadow Stack technique.

Ung and Cifuentes [37] describe UQDBT, a framework for the generation of dynamic binary translators. Its main goal is to ease the implementation of translators for diverse architectures, through the provision of an architecture description language. UQDBT does not use any special technique for indirect branch emulation, thus always falling back to the emulation manager.

Bala *et al.* [3] present Dynamo, a dynamic binary translator that uses traces as its basic translation unit. Traces enable the removal of return instructions, because a trace may span the whole body of the function all the way until the return site. In Dynamo, the target of indirect jumps are resolved by the Inline Caching technique. Additionally, Dynamo maintains a cache of the global table of mappings, which it consults whenever the Inline Caching fails. When both methods fail, the control of the execution is returned to the emulation manager, which performs a full address lookup.

Bruening *et al.* [9] present a framework for dynamic binary optimization of Windows applications. Through the use of a set of dynamic-link libraries (DLL), it takes control of an application and optimizes traces of frequently executed code. Since it translates between identical guest and host architectures, the actual process of translation consists merely of the copy and paste of the instructions in a trace. Control transfer instructions are an exception to this trend and must be patched in order to keep the control of the emulation within the virtual machine. The framework uses the Speculative Chaining technique to speed up the emulation of indirect branches. However, when it fails, control is not immediately transferred to the emulation manager, because the framework maintains, as well as Dynamo, a cache of the global table of address mappings, which it may use to resolve the translation from guest to host addresses.

BOA is a VLIW architecture designed for the emulation of PowerPC code at high clock frequencies [2,19]. The work has been inspired by the Daisy project [18], however its main goal was to maximize the operating frequency of the VLIW, rather than the instructions per cycle (IPC) count. BOA employs both interpretation and dynamic binary translation. During interpretation, it collects profiling information, which it uses to detect frequently executed portions of the guest application. In BOA, the basic translation unit is a trace that may span indirect branches by following its most frequently executed target.

Scott *et al.* [29–32] present Strata, a framework for the generation of dynamic binary translators. Throughout its development, the authors introduced two novel techniques for the emulation of indirect branches. The IBTC technique, described in Section 4.5, can be used for the emulation of any type of indirect branch. Nevertheless, the authors developed Fast Returns, a technique specialized in the emulation of return instructions. Fast Returns

violates the concept of emulation transparency by replacing the guest return address of a call instruction, with its corresponding translated address. This mechanism yields better execution performance, but it does not work if the return address gets modified during the procedure call. The authors argue that this is a violation of the SPARC ABI.

Patel and Lumetta [26] present rePLay, a hardware framework for dynamic binary optimization of x86 applications. Its hardware can natively execute x86 instructions, and it does so by forming long sequences of successively executed basic blocks, referred to as *frames*. The formation of a frame is preceded by a profiling phase, which counts the number of times that the targets of direct branches, indirect jumps, or return instructions are taken. Once a threshold is reached, the selected basic blocks are grouped into a frame and optimized. Additionally, control transfer instructions are converted into assertions that verify if the execution stays within the frame, and abort it otherwise. rePLay relies on speculative hardware in order to be able to recover a precise architectural state when aborting the execution of a frame. It also features profiling hardware that stores the execution count of both taken and not-taken path of direct branches. Indirect branches and return instructions are also monitored, however only the most recently executed target is stored by the profiling hardware. Thereby, direct and indirect branches are optimized in a similar way, and since rePLay can natively execute x86 instructions, the resolution of the target of indirect branches can be done trivially.

Probst [28] describes Bintrans, a dynamic binary translator generator based on an architecture description language also developed by the author. The basic unit of translation in Bintrans is a basic block, and it does not feature any special indirect branch emulation technique, falling back to the emulation manager when it encounters such control transfer instructions.

Cifuentes *et al.* [12] present Walkabout, a framework for the generation of dynamic binary translators based on an architecture description language. Walkabout generated translators initially emulate instructions using interpretation and profiling, until they determine that some portion of the guest application is hot. Afterwards, they form traces and optimize them for future re-execution. Indirect branches are emulated with the Speculative Chaining technique.

Bruening *et al.* [10] present DynamoRIO, a dynamic binary optimization and introspection system, based on Dynamo [3]. Both systems form traces for code optimization and use the Inline Caching technique to resolve the target of indirect branches. However, Dynamo inlines a single address comparison per branch, whereas DynamoRIO inlines multiple comparison tiers. Furthermore, DynamoRIO features a target profiling mechanism that is able to update the comparison data, thus increasing the effectiveness of the technique. In his Ph.D. thesis [8], Bruening discusses the use of the Shadow Stack mechanism for the emulation of return instructions, but he discards it since it does not

provide better execution performance when compared to treating returns as generic indirect jumps. Finally, DynamoRIO implements the indirect branch emulation technique based on central tables of mappings described in Section 3.6. It consists of small lookup routines that perform an address lookup in the global tables from within the code cache, and that may be either inlined for each indirect branch or shared among them all. DynamoRIO is a currently active open-source project that received modifications in the last ten years. A more recent version of it is described in Chapter 3.

Baraz *et al.* [4] describe IA-32 EL (Execution Layer), a dynamic binary translator that applies distinct levels of optimization to distinct portions of the guest code and that never relies on interpretation to detect hot regions. Initially, IA-32 EL treats every basic block from the guest code as cold, *i.e.* not frequently executed, thus keeping the optimization effort to its minimum. Besides, it inlines profiling code into each translation, in order to monitor the execution frequency. Once IA-32 EL detects hot code, it forms fragments longer than basic blocks and re-translates them with a higher optimization effort. The authors state that the targets of indirect branches are resolved with a fast lookup in a table of mappings, however they do not present details about the lookup, nor about the profiling mechanisms.

Kumar *et al.* [24] present an upgrade to Strata [30] based on compile-time profiling. Before the actual execution of the application, training inputs are used, in order to generate profiling information. Afterwards, when actually emulating the application, the profiling data is used, thus reducing the overhead related to runtime profiling. When translating basic blocks, the system already knows, based on the profiling data, one target of an eventual indirect jump, which it may then follow to form a trace. However, since the target of the jump may vary during execution, the translator adds a segment of code that verifies if the current and predicted targets are the same, similarly to the Speculative Chaining technique.

Bellard [7] describes Qemu, a system virtual machine capable of emulating several architectures. Qemu employs dynamic binary translation as its emulation technique and it first converts fragments of code from the guest application into an intermediate representation. Afterwards, it converts them into native code. The author states that adding a new architecture to Qemu is similar to adding a new architecture to the GCC compiler. The basic unit of translation in Qemu is a basic block and fragments ended with direct branches may be chained. However, whenever the MMU emulator modifies the page table, Qemu flushes all the chaining between basic blocks. The resolution of indirect branch targets, although not addressed in the paper, always rely on the emulation manager, as we could identify by code inspection.

Sridhar *et al.* [35,36] describe HDTrans, a dynamic binary translator that is performance-efficient, although it does not employ any code optimization technique. One of the reasons

for its efficiency is the introduction of two novel indirect branch emulation techniques: the Sieve, for register-indirect jumps, and the Return Cache for return instructions (see Section 4.6 and Section 4.9).

Luk *et al.* [25] present Pin, a dynamic binary translator tuned for code instrumentation. Pin employs two mechanisms for indirect branch emulation. First, the code generated by the translation of a jump instruction iterates over small segments of code, similar to the buckets in the Sieve technique (see Section 4.6). However the bucket chains are local to each indirect jump, whereas in the original Sieve technique, the chains are global and indexed by a hash function. When the bucket chains fail to find an address correspondence, control is transferred to a routine that looks the target up in local tables, similar to the IBTC.

Hazelwood and Klauser [20] describe the implementation of the ARM version of Pin [25], and also discuss the resource shortage faced by developers of translators for embedded systems, mostly due to memory constraints. Pin uses the IBTC technique for the emulation of regular indirect branches and the Shadow Stack for return instructions. However, the ARM architecture does not feature regular call and return instructions. Instead it provides a branch-and-link instruction that stores the return address in a link register and requires returns to be implemented with regular register-indirect jump instructions. In order to enable the use of the Shadow Stack, Pin assumes that every indirect jump might be a return and have them pop an entry from the Shadow Stack, even though this could remove entries that would be useful later.

Wang *et al.* [38] present StarDBT, a dynamic binary translator tuned for the emulation of home and business applications, such as Office Suites and Web Browsers. The authors used the metrics *wall time* and *duty cycle* to characterize the response time of the emulated applications, since long delays are easily noticed by the users. Since it translates between similar architectures, namely IA-32 and Intel64, the translation effort may be reduced to operations as simple as decoding and copying instructions. However, control transfers must be patched. The emulation manager maintains a global table of guest to host address translations, as well as a cache of the table. Indirect branches are converted into segments of inline code that search this table for a correspondence and jump to it when a match occurs, or fall back to the emulator, otherwise.

Dhanasekaran and Hazelwood [16] present a modification to the Inline Caching technique, which exploits the temporal locality of indirect branches targets. The authors argue that for the benchmarks in the SPEC CPU2006 suite, whenever an indirect jump target is executed, there is a 74% probability that the next execution of that same jump will target the same address. Their algorithm consists of an update scheme that keeps the most recently used (MRU) target of an indirect jump in the first position of the com-

parison chain of the Inline Cache. The results indicate an improvement in the hit rate of the first comparison, for all benchmarks.

Payer and Gross [27] describe an adaptive scheme that tries to dynamically select the best technique for each indirect branch in the emulated application. The scheme adds a counter per indirect branch, which keeps track of the number of mispredictions caused by the Inline Caching technique. When this number becomes higher than a threshold, the Inline Cache is replaced with a hash table lookup. The authors also present a novel technique, the Shadow Jump Table. In this technique, for the subset of the indirect jumps that look as if they use a jump table (*e.g.* `jump *addr(, %reg, 4)`), a new jump table is constructed that contains only the addresses of translated fragments. The indirect jumps themselves are then converted into segments of code that check for the boundaries of the table and use the new jump table as base.

Jia *et al.* [23] present SPIRE, an indirect branch emulation technique that completely removes the translations from guest to host addresses, through the reuse of guest code space. Indirect branches are left untouched and transfer control to the untranslated guest code, however the SPIRE technique avoids the execution of such untranslated code with a page-protection and an instruction-protection mechanism. Initially, every memory page that holds guest code is marked as not-executable, thus triggering a software trap whenever control is transferred to it. When that happens, SPIRE calculates the translated address of the target and inserts a trampoline to it. In order to protect the rest of the page, SPIRE populates the entire page, apart from the recently installed trampoline, with software traps (*e.g.* `INT3` instructions on the x86 platform).

Hiser *et al.* [21] present a detailed analysis of several indirect branch emulation techniques. The authors conclude that no technique is absolutely better than the other, and that the selection of the best technique is highly dependent on the underlying host architecture. They analyze the techniques IBTC, Sieve, Inline Caching, Shadow Stack, Return Cache, and Fast Returns. We summarize their findings for each technique in the following paragraphs.

Regarding return-specific techniques, *i.e.* techniques that are specialized in the emulation of return instructions, the authors discovered that the Fast Returns technique always outperforms the Shadow Stack and the Return Cache. Moreover, Fast Returns introduces no overhead when compared to native execution. However, it does violate transparency, as we described in Section 4.7. The remaining return-specific techniques provide poorer performance results, but do not present the transparency issues that could lead to wrong emulation results.

For the Sieve technique, the authors concluded that the only parameter that affects performance is the number of Sieve Buckets available for emulation. They also show that

for the UltraSPARC architecture the optimal number of buckets is 1K, whereas for the Pentium 4 Xeon and AMD Opteron architectures, this number is 16K.

Their analyzes of the Inline Caching technique revealed that an exact optimal number of inlined targets does not exist, nevertheless, inlining 0 to 3 targets provides the best performance results. Moreover, the authors discovered that profiling the targets of each indirect jump before inlining, provides better results than naively inlining the first seen targets. They also observed that for register-indirect jumps, profiling 30 executions is the most beneficial option, whereas for indirect calls, inlining the first two targets provides the best results.

For the IBTC technique, the authors show that inlining the lookup code into each fragment or calling a shared lookup routine provides similar results. They also show that handling collisions in the hash table with a replacement strategy, *i.e.* replacing old entries with newer ones, yields better hit rates on subsequent queries, but does not benefit performance. Finally, the authors compare the distributed approach of having an individual cache with a centralized approach, where all the fragments share the same hash table, and conclude that a shared table with inlined lookup code provides the best results.

To achieve such results, Hiser *et al.* extended the Strata framework to include all the above mentioned indirect branch emulation techniques. Strata would have been an invaluable resource to our project, however, we learned that it is not Open Source anymore and is owned by Zephyr Software LLC.

From the other virtual machines presented in Table 4.1, DynamoRIO, QEMU, and HDTrans were still good options, because their source code is indeed available and they run on x86 machines. The remaining virtual machines are either proprietary software or designed for other architectures.

The disadvantage of QEMU lies on the fact that it adds more overhead to the emulation than HDTrans and DynamoRIO do, thus potentially hindering our ability to evaluate the gains provided by the indirect branch emulation techniques. The disadvantage of HDTrans over QEMU and DynamoRIO lies on the facts that it does not have a currently active community of developers and our efforts to make it work failed. Therefore, we decide to use DynamoRIO. In our work, we analyze the IBTC and Inline Caching techniques and compare them to the unique indirect branch emulation technique used in DynamoRIO.

Chapter 7

Methodology and Results

In this project we used SPEC CPU2006 [1] to evaluate DynamoRIO and the techniques Inline Caching and IBTC. SPEC CPU2006 is a benchmark suite composed of compute-intensive applications. It is designed to stress the system's processor, memory subsystem, and compiler. The applications in the suite are classified as integer or floating-point intensive. Finally, the suite provides automated scripts to compile and execute the benchmarks. In this project we use the integer subset and the automated scripts.

For each of the experiments described in Chapter 7, 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 from SPEC CPU2006, which run each application three times, with the reference input.

After the experiments are run, we collect the data produced by the automated scripts, which automatically select which of the three iterations should be reported. The authors of SPEC advocate that the median value of several runs is the most statistically representative of the true central index of dispersion in computer science experiments [1]. We abide to their recommendation.

All the experiments are run in a single machine, featuring a pair of Intel E5645 processors at 2.4 GHz, 32 GiB's of RAM, and a 64-bits Ubuntu LTS 10.04 operating system. We compile both SPEC 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 5, and DynamoRIO itself. We also present our experiments and their results, as well as our analysis of the obtained results.

Figure 7.1 shows the overhead caused by emulation with DynamoRIO 4.1, in its vanilla version, *i.e.* without modifications. DynamoRIO can be thought of as a Dynamic Binary Optimizer, since it does not translate code between different architectures. Regardless,

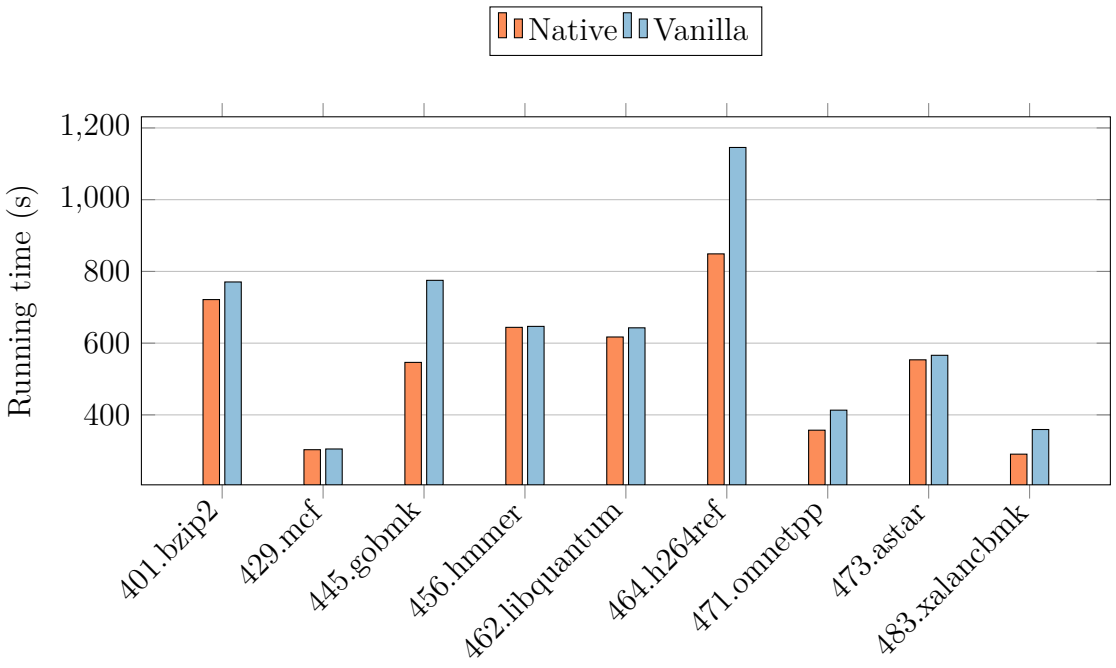


Figure 7.1: Native execution and emulation with the vanilla version of DynamoRIO.

it adds overhead to the execution of all but one of the integer benchmarks in the SPEC CPU2006 suite, due to indirect branches and eflag changes handling [10]. We set this version of DynamoRIO as our baseline.

Initially, we compare the execution performance of the techniques IBTC and Inline Caching. We do so by implementing them in the DynamoRIO framework and analyzing the execution times of the benchmarks. We implemented two versions of the IBTC technique: in the first, the address lookup is performed inside a function written in C, whereas in the second, it is implemented using the helper functions described in Chapter 5, thus emitting code directly into the code cache. The later approach should benefit from the fact that it does not need to prepare the execution of the C code, which involves saving and restoring the context, as well as loading a safe and transparent stack pointer.

Figure 7.2 shows that the C version of the IBTC penalizes performance, when compared to the version which emits code directly into the code cache, on four benchmarks: *hmmer*, *h264ref*, *omnetpp*, and *xalancbmk*. It also shows that the Inline Caching technique provides better results than the IBTC for the same benchmarks. Finally, it shows that for the other benchmarks the resulting running times do not differ as much. We know that the overhead in the C version is caused by the additional steps related to the save and restore of the processor state, because nothing else has changed between the two versions of the IBTC. But we cannot conclude anything about the differences between the IBTC

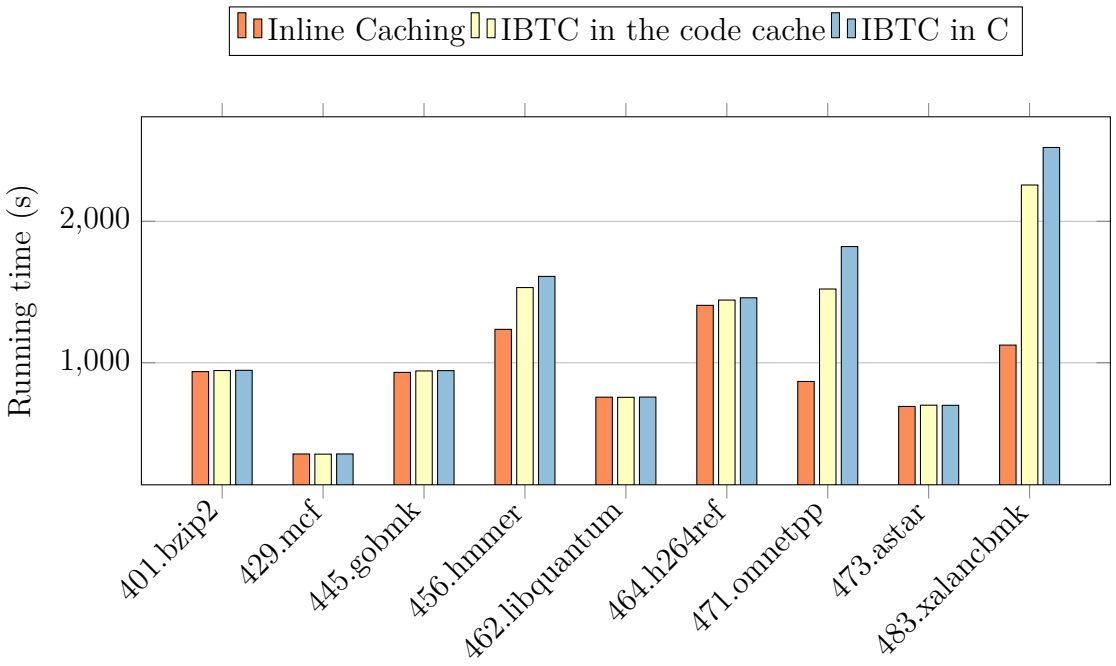


Figure 7.2: Emulation of indirect branches with the Inline Caching and IBTC techniques. The Inline Caching implementation emits code directly into the code cache. One version of the IBTC is also implemented that way, but it also has a version that calls functions written in C.

and the Inline Caching versions. Before we make such conclusions, we shall examine how the techniques Inline Caching and IBTC behave compared to the base implementation of DynamoRIO, *i.e.* the vanilla version. Since the C version of the IBTC introduces higher overhead, we discard it from our subsequent experiments.

Since DynamoRIO also implements its own indirect branch target resolution technique, we compare its base implementation with the addition of the IBTC and Inline Cache techniques. Surprisingly, the addition of the two never benefit performance. As a matter of fact, they introduce overheads of up to 530%, as shown in Figure 7.3. We argue that the addition of the Inline Cache and IBTC techniques to the DynamoRIO code base can be thought of as an overlaying of techniques. Thus, they might interfere with other parts of DynamoRIO, such as the hotness prediction algorithms, rather than just with its indirect branch emulation technique.

Our first conjecture about the source of the overhead was that the caching of entries in the local storage of the IBTC technique was not sufficiently large to hold the several targets that an indirect jump might have, and that this could lead to the poor performance results. In order to remove this uncertainty, we measured the hit rate of the techniques, finding results as high as 99.99% and as low as 91.54%. Table 7.1 shows the hit rates and

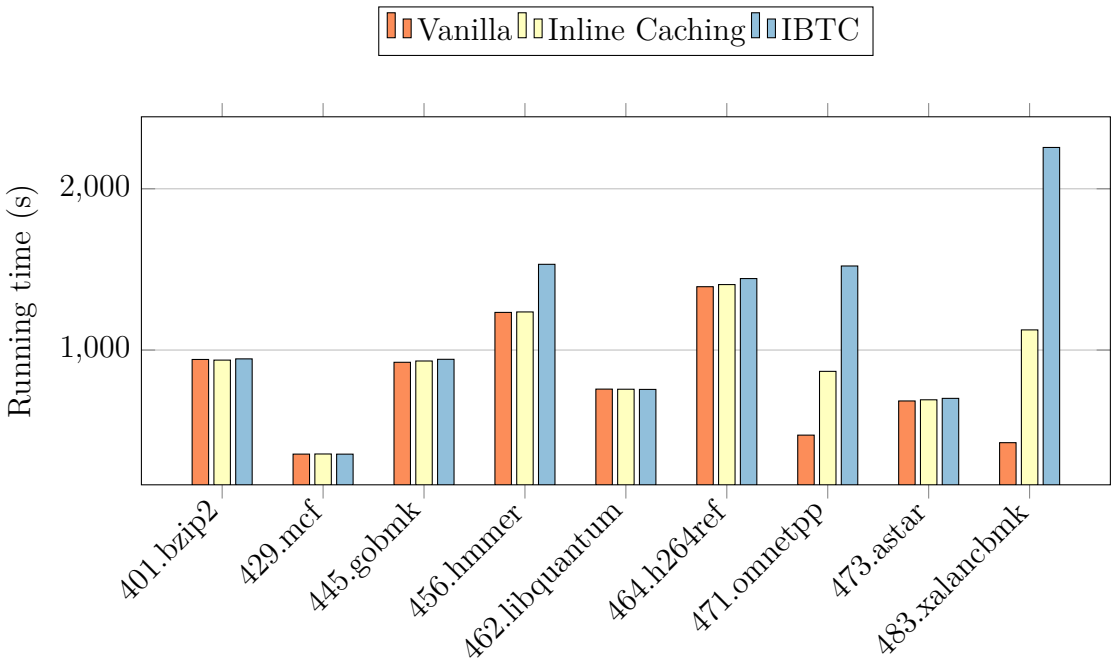


Figure 7.3: Running times of the overlaying of the Inline Caching and IBTC techniques to DynamoRIO.

the total number of executed indirect jumps. The lowest hit rate, for *libquantum*, happens due to the reduced number of total indirect jumps executed, since the first execution of each jump always misses. Therefore, the lack of space in the IBTC tables does not cause low hit rates.

Still uncertain about the causes of the poor performance, we modified our implementation in order to determine if the lookup code was the sole responsible for the surprisingly high overhead in the benchmarks *hmmer*, *h264ref*, *omnetpp*, and *xalancbmk*. The modification consists of removing the hit path of the IBTC technique, *i.e.* we still look the guest target address up in the local caches, but we never follow the corresponding host address. Even if a hit occurs, we fall back to DynamoRIO, as if we had not found the target. Figure 7.4 shows that the running times of the four benchmarks were reduced after this modification, which indicates that something else, other than the cache lookup and update times, is also hindering performance.

Regardless, not following the hit path did not remove all the overhead from the emulation, although it did remove the largest part of it. This means that the time spent in the address lookup and in the table update is not negligible. Table 7.1 shows the absolute number of indirect jumps executed by the benchmarks. Unsurprisingly, the benchmarks

Benchmark	**Misses**	**Hits**	**Hit rate (%)**
401.bzip2	3287	5265054	99.93
429.mcf	133	1807573	99.99
445.gobmk	4717	13672112	99.96
456.hmmer	2165	719647704	99.99
462.libquantum	47	509	91.54
464.h264ref	6268	151560350	99.99
471.omnetpp	9350	3527523910	99.99
473.astar	1490	11007350	99.98
483.xalancbmk	9854	2800900967	99.99

Table 7.1: Hit rates for the IBTC technique

hmmer, *omnetpp*, and *xalancbmk* are the benchmarks with the highest count of indirect jumps.

We based our second conjecture about the source of the overhead on the ability that DynamoRIO has to translate code either as basic blocks or as traces. Section 3.2 describes how DynamoRIO forms fragments of code from the guest application. First, it forms basic blocks, which have a single entry-point and a single exit-point. Afterwards, when it determines that a basic block is hot, it starts to form optimized traces, which are collections of sequential basic blocks. We formulated that since we store references to basic blocks in the local storage of the IBTC and Inline Caching techniques, we will never jump to the optimized translations. DynamoRIO, on the other hand, actively updates its global tables of mappings on the event of trace creation, thus benefiting from the more optimized code.

There are two modifications to the IBTC and Inline Caching techniques that may prevent them from using these unoptimized fragments. First, on the event of trace creation, we could examine all the local caches, and replace old references to basic blocks with references to newly created traces. Second, we could store only traces in the local caches.

The first approach diverts from the concept behind the IBTC and the Inline Caching. These techniques act passively towards the event of basic block and trace creation, *i.e.* they only update each local storage when the corresponding indirect jump is executed, and only when a hit does not occur. The second approach, on the other hand, does not alter the behavior of the techniques. In any case, our implementation of both aproaches shows that they also do not provide performance enhancements over the base implementation of DynamoRIO. Figure 7.5 shows that denying the insertion of basic blocks references in our IBTC tables does reduce the overhead introduced by our implementation, but it does

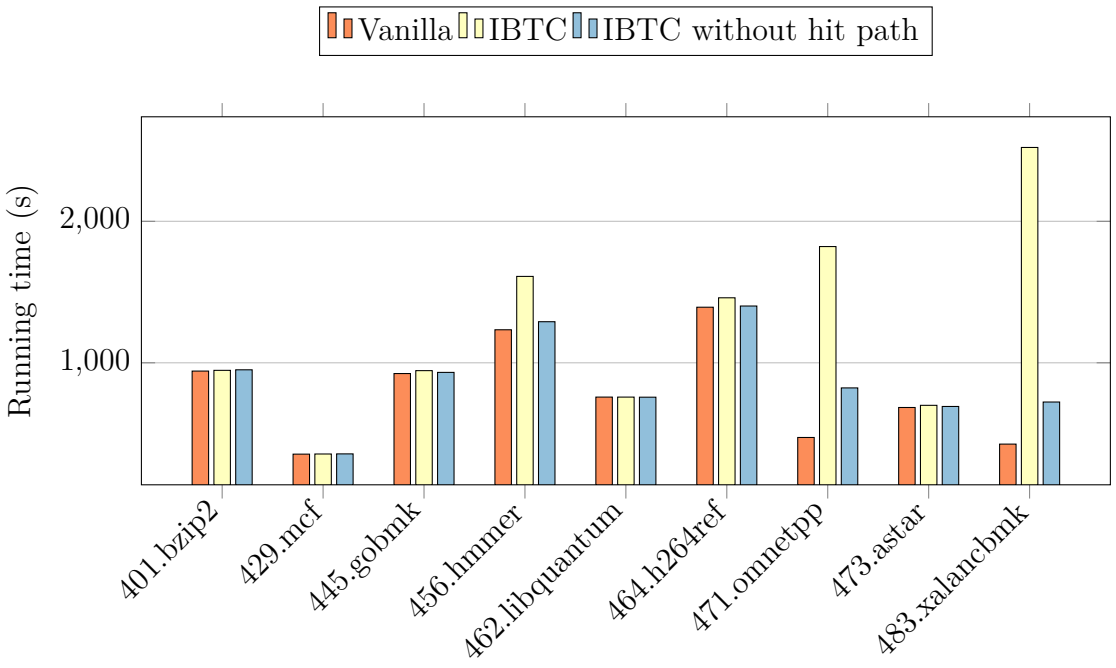


Figure 7.4: Hit path removed from IBTC.

not yet fully explain the loss of performance when compared to the base implementation of DynamoRIO.

Finally, we observe how overlaying DynamoRIO with the IBTC and the Inline Caching techniques modifies the creation of traces. When we jump to the basic blocks pointed to by the hit path of either the IBTC or the Inline Caching, we modify the addresses that get selected as trace heads, as described in Section 2.6. This means that our implementation of the Inline Caching and IBTC techniques has a side-effect on the hotness prediction algorithms of DynamoRIO, which is fundamental to the performance of DynamoRIO [8].

We tried to tackle this issue with the use of the *runtime* options, `disable_traces` and `no_indirect_stub`. However, the first option caused errors in most of the benchmarks, whereas the second did not alter the results. We also tried to solve the issue by removing from our caches the same fragments that DynamoRIO removes from theirs while building traces. Several points in the code base of DynamoRIO perform fragment removal. We inserted callbacks to our code in all of them, but this did not modify the performance of the execution. Our last option would be to fully understand the hotness prediction algorithms of DynamoRIO, and modify our techniques in order to take advantage from it. Unfortunately, we could not follow this line of work, due to time constraints.

Our experiments helped reveal how two parts of the DynamoRIO code base (indirect branch emulation and hotness prediction algorithms), that are apparently independent,

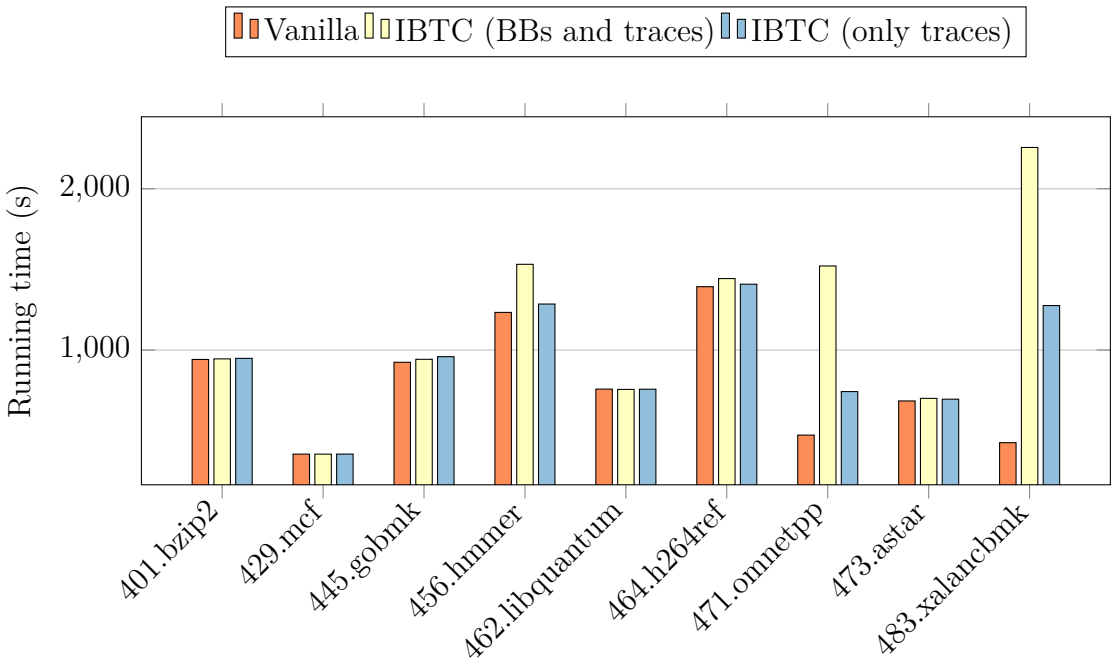


Figure 7.5: Reduced overhead obtained by denying the insertion of Basic Blocks (BBs) into the IBTC caches.

interfere with each other. We have shown how the indirect branch emulation technique employed by DynamoRIO correlates with the techniques described in the literature. Finally, our experiments have enabled a deeper understanding of the code base of DynamoRIO.

Chapter 8

Conclusion

In this master thesis, we evaluate DynamoRIO and the indirect branch emulation techniques Inline Caching and IBTC. DynamoRIO is a same-ISA process virtual machine that employs dynamic binary translation as its emulation technique. In order to provide optimized, near-native, execution performance, it features hotness prediction algorithms, as well as two levels of translation complexity (basic blocks and optimized traces), and optimized techniques to transfer control between fragments without leaving the code cache.

We have experimented with register-indirect control transfers emulation techniques, and observed that the built-in technique of DynamoRIO presents a major difference when compared to the techniques Inline Caching and IBTC. The later techniques maintain the mapping of guest to host addresses in small caches, individual to each indirect jump location, whereas, DynamoRIO maintains global tables of mappings.

Initially, we thought that the distributed nature of the Inline Caching and IBTC caches could benefit the performance of the indirect jumps emulation, because collisions in the local hash tables would be less frequent than with shared global tables, thus leading to faster hit times. But DynamoRIO solves this potential issue by increasing the size of its global tables, whenever they reach a defined occupation threshold.

Moreover, since DynamoRIO translates code in two levels of complexity, it should update the mapping tables whenever it switches between basic blocks and traces. The centralized nature of the global tables of DynamoRIO is better suited for this task, because it reduces the effort required by updates. When a basic block is converted into a trace, DynamoRIO must search and update an entry only in its global mapping tables.

On the other hand, in the techniques Inline Caching and IBTC, a full update would require searches in every table, which are as many as the total number of indirect jumps executed by the guest application. Moreover, the Inline Caching and IBTC techniques were primarily developed for dynamic binary translators with a single level of translation complexity. Hence, they are expected to act passively towards the translation of

fragments, and only actively update their entries on the event of the execution of the indirect branch. We modified the IBTC technique by denying basic blocks in its local caches, which did improve the execution performance, but still did not beat the built-in technique of DynamoRIO.

Finally, our implementation of the Inline Caching and IBTC techniques produced an undesired side-effect on the hotness prediction algorithms of DynamoRIO. By directly jumping to the targeted basic blocks of indirect jumps, they modify the portions of the guest application that are selected as trace heads, leading to poorer performance results.

Nonetheless, this master thesis have qualitatively described how DynamoRIO solves potential issues posed by the emulation of indirect branches. It provides an up-to-date documentation of a recent version of the framework, as well as it describes how the techniques employed by DynamoRIO correlate with that described in the literature, rendering it easier for future development of the research.

Bibliography

[1] SPEC: Standard Performance Evaluation Corporation.
http://spec.org.

[2] Erik Altman, Michael Gschwind, Sumedh Sathaye, Stephen Kosonocky, Arthur Bright, Jason Fritts, Paul Ledak, Craig Agricola, and Zachary Filan. BOA: the architecture of a binary translation processor. Technical report, IBM Research, 2000.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[4] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2003.

[5] Robert Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Technical Conference*, 1990.

[6] Robert C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1995.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[8] Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[9] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[10] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: a profile-directed binary translator. *IEEE Micro*, 1998.

[12] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout: a retargetable dynamic binary translation framework. In *Proceedings of the IEEE Workshop on Binary Translation*, 2002.

[13] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1994.

[14] Thomas Cormen, Clifford Stein, Ronald Rivest, and Charles Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[15] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1984.

[16] Balaji Dhanasekaran and Kim Hazelwood. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, 2011.

[17] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 2001.

[18] Kemal Ebcioğlu and Erik R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture*, 1997.

[19] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller. Dynamic and transparent binary translation. *Computer*, 2000.

[20] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[21] Jason D. Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Transactions on Architecture and Code Optimization*, 2011.

[22] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: combining emulation and binary translation. *Digital Technical Journal*, 1997.

[23] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. SPIRE: Improving Dynamic Binary Translation through SPC-Indexed Indirect Branch Redirecting. In *Proceedings of the International Conference on Virtual Execution Environments*, 2013.

[24] Naveen Kumar, Bruce R. Childers, Daniel Williams, Jack W. Davidson, and Mary Lou Soffa. Compile-time planning for overhead reduction in software dynamic translators. *International Journal of Parallel Programming*, 2005.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[26] Sanjay J. Patel and Steven S. Lumetta. rePLay: a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 2001.

[27] Mathias Payer and Thomas R. Gross. Generating Low-Overhead Dynamic Binary Translators. In *Proceedings of the Annual Haifa Experimental Systems Conference*, 2010.

[28] Mark Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the IEEE Workshop on Binary Translation*, 2001.

[29] Kevin Scott and Jack Davidson. Strata: a software dynamic translation infrastructure. Technical report, University of Virginia, 2001.

[30] Kevin Scott, Jack Davidson, and Kevin Skadron. Low-overhead software dynamic translation. Technical report, University of Virginia, 2001.

[31] Kevin Scott, Naveen Kumar, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Overhead reduction techniques for software dynamic translation. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2004.

[32] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[33] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 1993.

[34] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processors*. Morgan Kaufmann, 2005.

[35] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: a low-overhead dynamic translator. *ACM SIGARCH Computer Architecture News*, 2007.

[36] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the International Conference on Virtual Execution Environments*, 2006.

[37] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[38] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. StarDBT: an efficient multi-platform dynamic binary translation system. In *Proceedings of the Conference on Advances in Computer Systems Architectures*, 2007.

[39] Emmett Witchel and Mendel Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.