SPIRE: Improving Dynamic Binary Translation through SPC-Indexed Indirect Branch Redirecting

Ning Jia  Chun Yang  Jing Wang  Dong Tong  Keyi Wang

Department of Computer Science and Technology, Peking University, Beijing, China
{jianing, yangchun, wangjing, tongdong, wangkeyi}@mprc.pku.edu.cn

Abstract

Dynamic binary translation system must perform an address translation for every execution of indirect branch instructions. The procedure to convert Source binary Program Counter (SPC) address to Translated Program Counter (TPC) address always takes more than 10 instructions, becoming a major source of performance overhead. This paper proposes a novel mechanism called SPC-Indexed REDirecting (SPIRE), which can significantly reduce the indirect branch handling overhead.

SPIRE doesn’t rely on hash lookup and address mapping table to perform address translation. It reuses the source binary code space to build a SPC-indexed redirecting table. This table can be indexed directly by SPC address without hashing. With SPIRE, the indirect branch can jump to the originally SPC address without address translation. The trampoline residing in the SPC address will redirect the control flow to related code cache. Only 2–6 instructions are needed to handle an indirect branch execution. As part of the source binary would be overwritten, a shadow page mechanism is explored to keep transparency of the corrupt source binary code page. Online profiling is adopted to reduce the memory overhead.

We have implemented SPIRE on an x86 to x86 DBT system, and discussed the implementation issues on different guest and host architectures. The experiments show that, compared with hash lookup mechanism, SPIRE can reduce the performance overhead by 36.2% on average, up to 51.4%, while only 5.6% extra memory is needed.

SPIRE can cooperate with other indirect branch handling mechanisms easily, and we believe the idea of SPIRE can also be applied on other occasions that need address translation.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors – Code generation, Optimization, Run-time environments.

General Terms Algorithms, Design, Performance

Keywords Dynamic Binary Translation, Indirect Branch, Redirecting

1. Introduction

Dynamic Binary Translation (DBT) is a technology that translates the source binary code to target binary code at run-time. DBT is widely used in many fields, such as ISA translator [1], dynamic instrumentation [2, 3], dynamic optimizing [4], program analysis [5], debugging [6], and system simulator [7].

In many cases, for a DBT system to be viable, its performance overhead must be low enough. How to handle control-transfer instructions is a key aspect to the performance. For conditional branch and direct jump instructions, whose branch targets are fixed, code block chaining [7] technique can significantly eliminate the overhead of transferring between code blocks. However, for Indirect Branch (IB) instructions, whose branch target cannot be determined until run-time, DBT system must perform an address translation from Source binary Program Counter (SPC) address to Translated Program Counter (TPC) address at every execution. Previous research [8, 9, 10] indicates that handling indirect branch instructions is a major source of performance overhead in DBT system.

DBT systems maintain an address mapping table to record the relationship between SPC and TPC, and a hash lookup routine is always used to perform the address translation. The SPC address is hashed to a key to select TPC from a hash table. However, even a carefully hand-coded hash lookup routine would cost more than 10 instructions [8], resulting in a non-trivial performance overhead.

This paper proposes a novel mechanism called SPC-Indexed REDirecting (SPIRE), which can significantly reduce the on-the-fly indirect branch handling overhead. SPIRE doesn’t rely on hash lookup and the mapping table to perform address translation. It reuses the source binary code space to build a SPC-indexed redirecting table, which can be indexed directly by SPC address without hashing. The memory at SPC address originally contains source binary instructions, and would be overwritten by a redirecting trampoline before first used. With SPIRE, the indirect branch can jump to the originally SPC address without address translation. The trampoline residing in the SPC address will redirect the control flow to related code cache. Only 2–6 instructions is needed to handle an indirect branch execution.

As some source binary code is corrupted by SPIRE table entries, a shadow page mechanism is explored to keep transparency for other code that may access source binary, such as self-modifying code. To decrease the memory overhead, an online profiling mechanism is developed to classify indirect branches, and only hot indirect branches will be optimized by SPIRE.

We implemented SPIRE mechanism on an x86 to x86 DBT system. The experiment shows that, compared with hash lookup mechanisms, SPIRE can reduce the performance overhead by 36.2% on average, up to 51.4%, while only 5.6% extra memory is consumed by shadow code pages. Although x86 isn’t the most suitable platform for SPIRE, we still gain a considerable performance improvement. We also discuss the implementation issues on different guest and host platforms in detail.
The rest of the paper is organized as follows. Section 2 provides an overview of indirect branch handling mechanisms and discusses related work. Section 3 shows the principle and main challenges of SPIRE. Section 4 implements SPIRE mechanism on x86 platform and discusses the implementation issues on different platforms. Section 5 describes our experimental results. Section 6 summarizes this work and discusses future directions.

2. Overview of Indirect Branch Handling

Figure 1 describes the general workflow of indirect branch handling in DBT systems. When the branch target is obtained, it’s a SPC address, and the PC dispatcher is used to convert SPC to related code cache address (TPC). A mapping table is probably needed during address translation. The dispatcher will jump to the TPC after address translation. Because indirect branch always has multiple targets, so the PC dispatcher is a single-entry/multi-exits routine.

There are many methods to implement PC dispatcher. Classified by address translation method, the PC dispatcher can be prediction-based or table-lookup based. The prediction-based method can achieve a better performance while prediction hits, but have to perform a full table lookup while prediction fails.

Classified by code placement, the PC dispatcher can be private or shared. The private dispatch is always inlined in the code cache to eliminate the overhead of context switch. The shared PC dispatcher is used by more than one indirect branch, which can reduce the total code size.

Classified by implementation, the PC dispatcher can be pure-software or SW/HW co-designed. The co-designed mechanisms always achieve a considerable performance improvement, but poor versatility.

2.1. Related Work

In this section, we will discuss the previous indirect branch handling mechanisms of DBT systems. For clarity, we classified these mechanisms by whether special hardware or instruction is needed.

2.1.1. Pure-Software Mechanisms

DBT systems without special hardware always use hash table or software prediction to reduce the overhead of handling indirect branch.

There are many design options for hash table. The hash lookup routine can be inlined or be shared. The address mapping data can be stored in several small private tables or a large public table [8]. Typical hash table implementations include Indirect Branch Translation Cache (IBTC) in Strata system [11], and SIEVE in HDTrans system [12]. Hash table is easy to implemented, but even a carefully hand-coded hash lookup routine still need more than 10 instructions. For example, the IBTC lookup routine take approximately 17 instructions [8].

Based on the locality of indirect branch’s targets, researchers propose another efficient technique called software prediction [13], also called indirect branch inlining [8], which is similar with de-virtualization [14] techniques in object oriented language. Software prediction uses a compare-jump chain to predict the branch target. When the compare instruction reports equal, the CPU continues to execute with the branch target.

There are many methods to implement PC dispatcher. Classified by address translation method, the PC dispatcher can be prediction-based or table-lookup based. The prediction-based method can achieve a better performance while prediction hits, but have to perform a full table lookup while prediction fails.

2.1.2. Co-Designed Mechanisms

Despite the pure-software techniques, researchers also propose a number of SW/HW co-designed techniques. Kim et al. design a Jump Target-address Lookup Table (JTLT)[13], which collaborates with the traditional Branch Target Buffer (BTB). JTLT is a hardware cache of the address mapping table. When an indirect branch is encountered, the CPU continues to execute with the prediction result of BTB, and queries the JTLT concurrently. The JTLT result is used to verify the prediction, if BTB prediction fails, the control-flow would jump to the JTLT result.
Hu et al. add a Content-Associated Memory (CAM) [17] to accelerate the address translation. Each CAM entry contains three fields: process ID, SPC, and TPC. When the branch target is determined, the system uses a special CAM lookup instruction to translate SPC to TPC. Li et al. propose a new CAM partition and replacing mechanism called PCBPC [18], which can reduce the CAM miss rate from 3.7% to 1.6%.

Guan et al. analyze the performance bottleneck of DBT systems, and design a SW/HW collaborative translation system called CoDBT [19]. CoDBT uses a specific hardware accelerator to deal with time-consuming operations, such as address translation. Michocka et al. evaluate several systems, and point out that a major overhead of address translation is caused by context save/restore [20]. They believe that adding light-weighted eflags save/restore instructions will be helpful.

Compared with our SPIRE, co-designed mechanisms need to modify the hardware or ISA, thus cannot be widely used on existing platforms.

There are many other mechanisms to improve the performance of DBT system, such as translation path selection scheme [21], specialization for target system [22], process-shared and persistent code cache [23, 24], individual optimizing thread on multi-core platform [25]. As they don’t focus on indirect branch handling, we won’t discuss them here.

3. How SPIRE works

Previous indirect branch handling mechanisms primarily contain two steps: 1) Translate the SPC to TPC. 2) Jump to the TPC. The address translation process results in the major overhead.

What would happen if the control-flow jumps to the original SPC address without address translation? As most of the DBT systems load source binary into the same memory address as native execution, the memory on SPC address always contains an un-translated source binary instruction. Thus the system would fail if control-flow jumps here directly. However, if this source instruction was overwritten by a trampoline jumping to the related TPC in advance, the control flow would be redirected to the expected code cache address.

Therefore, we propose a novel indirect branch handling mechanism called SPC-Indexed Redirecting. SPIRE reuses virtual address space of source binary code to build a redirecting table. This table can be indexed by the SPC address directly. The table entries reside in the source binary code space sparsely, and each entry only contains one filed: a redirecting trampoline to TPC.

Since the source binary code may be accessed by translator and self-modifying code, it should always be kept in the memory. So SPIRE doesn’t introduce too much extra memory overhead. For correctness, we develop a shadow page scheme to keep the corrupted source binary code transparency when being accessed.

3.1. Prototype of SPIRE

With SPIRE mechanism, each SPC address of branch target is regarded as a table entry of SPIRE. These entries initially contain source binary instructions, and would be filled with a redirecting trampoline before used.

Figure 2 presents the stable state of SPIRE mechanism. After the table entry is filled, only two steps are needed to handle an indirect branch execution: 1) Table-entering: jumps to SPC address directly. 2) Redirecting: jumps to code cache. The latter step only needs 1 instruction. If the source binary is loaded into the same address space as native execution, which is the most common case, the table-entering routine only needs 1 instruction too.

It should be noted that the prototype shown in Figure 2 is just a common implementation of SPIRE. Based on the core idea, several variants can be derived when necessary. For example, some DBT systems may load the source binary into different memory address with native execution, so the table-entering routine should add an offset to the SPC address, and about 4 more instructions are needed. Meanwhile, the “instruction-based” redirecting table can be converted to “data-based”. Under the “date-based” variant, the redirecting instruction will be replaced with the TPC address, which can shorten the table entry. These variants will be discussed in the following section.

SPIRE leverages the relationship of memory address and its value to represent the mapping between SPC and TPC. SPC address of branch target is used to distinguish different branch occasions. Be different from the traditional translation-and-jump mechanisms, SPIRE can jump to the SPC address without address translation, and the trampoline will lead control-flow to expected code cache.

3.2. Main Challenges

The workflow of SPIRE seems simple and neat, but in order to keep the correctness and transparency while achieving a satisfied performance, there are several obstacles to overcome.

Table Filling. Since the branch target of indirect branch cannot be determined until executed, so SPIRE table is totally unfilled when system starts. The table entries will be filled after first used. Meanwhile, to ensure the correctness, there must be a scheme to distinguish whether the table entry is filled or not. An unfilled table entry should never be executed.

Table Entry Overlapping. SPIRE reuses the memory address of branch targets to build table entries, and the redirecting trampoline would take several bytes. If two branch targets are too close, a table entry may be overlapped with others. Such case becomes more complicated if the ISA of source binary is variable-length. The table entry overlapping must be avoided, or the system may fail.

Transparency. Due to the workflow of SPIRE, the source binary code may be corrupted by SPIRE table. However, even after being translated, source binary code still may be accessed by self-modifying code or re-translated routine. To make the program
run correctly in all cases, SPIRE should keep the corrupted source binary code totally transparency to other code.

**Architecture-Specific Issues.** There are also some implemented issues related to the features of source/host architectures. For example, SPIRE may leverage page protection scheme to keep correctness, which relies on the support of memory management unit; the redirecting trampoline should contain a jump instruction, which may be limited by the addressing mode of ISA.

Next, we will discuss the general solutions to these challenges, and explore more details of SPIRE implementations on several typical platforms.

### 3.3. On-Demand Table Filling

Since the SPIRE table is unfilled when system starts, an on-the-fly entry filling method should be provided to fill the table entries at an appropriate moment, while ensuring the unfilled entries wouldn't be executed. Checking the status of entry at run-time is too time-consuming. So we propose a two-level exception-based entry filling scheme: page-grained and instruction-grained.

Inspired by the on-demand paging mechanism in memory management scheme of operating system, we develop a page-protection mechanism to distinguish whether the entry is filled. After the source binary is loaded, SPIRE will unset the EXEC permission of all the code pages. When the control-flow jumps to an unfilled table entry, a page fault exception would be triggered. Then the handler can fill the related table entry and set this page executable. However, this mechanism only works at page-grained, and cannot deal with the case that several entries belong to a same page. So we still need an instruction-grained mechanism to classify table entries.

After the first table entry of a page is filled, the other space of this page will be filled with software trap instructions of host platform, for example, the INT3 instruction on x86 platform. When the control-flow jumps to an unfilled table entry of this page, a software exception will be triggered. The exception handler will fill the related table entry.

### 3.4. Transparency

A shadow code page mechanism is developed to achieve the transparency goal. When a code page is overwritten by SPIRE entry, it will be copied to shadow page space, and the translator would disable the READ and WRITE permissions of this corrupted page. The shadow page space is managed by the translator. If some other code tries to access the corrupted code pages, a read/write page fault exception will be raised, and the handler will redirect the access to related shadow code page.

On Linux operating system, SPIRE uses the signal mechanism to handle the exceptions. Some guest applications may register their own signal handler, thus SPIRE must keep signal transparency to guest application. SPIRE registers its own handlers to operating system, and intercepts the guest application’s signal registration. When a signal is raised, SPIRE’s handler will check whether it is sent to SPIRE or to guest application, and then select the correlated handler.

When application has bugs, the control flow may jump to a wrong address which doesn’t contain valid code. Then the behavior will be un-predicted and the program may fail. If such an application runs on a DBT system with SPIRE, when this type of bug exists, SPIRE cannot exactly reproduce same error states as native execution. It is hard for DBT systems to achieve 100% error transparency[3]. Current SPIRE implementation cannot keep transparency to such an illegal program counter bug, but SPIRE works well with stable applications.

Algorithm 1 describes the process of SPIRE entry filling. When all the table entries are filled, the control-flow will jump between code blocks without raising exceptions, achieving a satisfied performance.

**Algorithm 1: SPIRE table entry filling**

1. Unset the EXEC permission of all the pages of source binary code segment.
2. When an Execution page fault is raised.
   1. Copy the page to shadow page area.
   2. Fill the related table entry.
   3. Enable the EXEC permission of this page, and unset the READ and WRITE permissions.
3. When a software exception is raised
   1. If raised by SPIRE, then fill the related table entry. Else call the guest application’s handler.
4. When a Read/Write page fault is raised.
   1. If raised by SPIRE, redirect the access request to related shadow page. Else call the guest application’s handler.

### 3.5. Optimization

A profiling scheme is adopted by SPIRE to selectively optimize the indirect branch, and an invalidation routine will be called when SPIRE may do harm to the performance.

**Online Profiling.** As noted by many researchers, most of execution time costs on a small part of hot code, so do the indirect branches. Most of the dynamic executions of indirect branch happen on several hot branch sites, and some cold indirect branches only execute several times. As shown in Figure 3, on our benchmarks, about 40% indirect branch instructions execute less than 1000 times, and contribute less than 1% of the total executions.

SPIRE relies on the page fault and software exception to trigger the table filling routine, which may cost more than 1000 cycles. Thus using SPIRE to handle the cold indirect branches isn’t worthy. Furthermore, more shadow pages will take more extra memory.

To minimize the overhead of SPIRE, we explore an online profiling mechanism to identify hot instructions. When the execution count of an indirect branch exceeds the pre-defined threshold, it will be marked as hot instruction. Only hot instructions will be optimized by SPIRE, and cold indirect branches are still handled by other mechanism like hash lookup.

![Distribution of IBs execution count](image-url)
Invalidation Scheme. SPIRE can be seen as a cache of the mapping table between SPC and TPC. When the mapping is changed by some events, e.g. self-modifying code, code cache flush, SPIRE needs to update the related table entries to maintain the consistency. A two-level invalidation scheme is proposed to ensure the correctness and performance.

1) Invalidate table entries. When the mapping of SPC and TPC is modified, the related SPIRE table entries would be invalidated.

2) Invalidate indirect branch sites. When some source code is modified frequently, the exceptions caused by table entry refilling would result in non-trivial overhead. To avoid performance decreasing, indirect branches related to this target would fall back to hash lookup mechanism.

4. Implementation of SPIRE

As discussed in section 3, to overcome the obstacles, SPIRE prefers some features of guest and host architecture. First, the permission of host architecture’s memory page can be set to NO_READ | NO_WRITE | EXEC. So that the corrupted source code pages can be protected from being accessed. Second, the source binary instructions should be long enough, so that the trampoline can reside in the instruction slot.

The RISC architectures like ARM always can fulfill these requirements, but some architecture like x86 can’t. However, SPIRE can still be applied on these architectures. In this section, we will implement our SPIRE mechanism on an x86 to x86 binary translator to illustrate the versatility of SPIRE.

4.1. Keep Transparency of Corrupted Code Page

The READ|WRITE permission of x86’s page table is controlled by one bit, thus the READ and WRITE permission cannot be disabled simultaneously. As a result, the previous page protection scheme cannot prevent the corrupted code page from being accessed by self-reference or self-modifying code.

We modified the shadow page scheme to keep transparency of the corrupted source binary code page. After the source binary is loaded into memory, SPIRE allocates a continuous shadow code space with the same size of source binary code space. As shown in Figure 4, instead of reusing the source binary space, the SPIRE table is placed into the shadow page space, which has a fixed offset with the source binary space. So that the original code pages won’t be corrupted. With this mechanism, the table-entering routine should jump to the address at SPC + Offset, instead of the original SPC.

In this case, instead of the original 1-instruction table entering route, SPIRE needs to cost about 5 instructions to add the offset to SPC, but still much better than hash lookup. The following code sequence is an example of SPIRE table entering routine.

```
push %ecx;  // spill register
mov %ecx, *dest;
lea %ecx, $offset;
jmp %ecx;  // jump to (SSPC + OFFSET)
pop %ecx;  // executed later
```

The shadow code space is a direct-mapping space of source binary code space. It needn’t be initialized as a copy of source binary. Only the pages containing SPIRE table entries will be modified, and the unused pages can be left untouched. Due to the on-demand page allocating mechanism in operating system, only the page being accessed would be allocated a physical page frame. So the shadow page space won’t take too much physical memory.

Some applications like virtual machine may generate/load binary code at runtime, SPIRE may not always be able allocate a shadow space with the fixed offset to the new generated code. When SPIRE finds that it cannot maintain a shadow code space which can map all the source binary code, it will invalidate all the optimized indirect branches, and totally fall back to hash lookup mechanism for safety.

4.2. Avoid Overlapping of Table Entries

As x86 is a variable-length ISA, its instruction length varies from 1-byte to 15-bytes. Meanwhile, the trampoline of SPIRE needs a 5-bytes long direct jump instruction on x86 host architecture. Therefore, the source instruction slot may be shorter than SPIRE table entry. If the table entry exceeds the instructions slot, it may be overlapped with another table entry. This situation happens when two branch targets are very close, and it will make the program failed.

To solve this problem, we propose an entry chain scheme. When the target source instruction slot is shorter than table entry, the “jmp TPC” trampoline cannot be filled directly. In this case, SPIRE fills the slot with a 2-bytes short jump, which can jumps to a

---

**Figure 4.** Workflow of SPIRE on x86 to x86 DBT

**Figure 5.** Structure of entry chain
following instruction slot long enough. If a long slot cannot be found inside the range of the short jump (0–127 bytes), the furthest appropriate slot is selected as an intermediate slot, which also be filled with a short jump. Then we can continue to search long slot by taking this new instruction as start point. This process can be recurred until the long slot is found or the number of intermediate slots exceeds the threshold. Figure 5 demonstrates the structure of entry chain.

To prevent the intermediate slot from conflicting with another table entry, we add a software exception instruction (INT3 on x86) in the front of the slot. So the intermediate instruction slot should be at least 3-bytes long. The ending slot should be no less than 6-bytes. If the intermediate slot is just a target of another indirect branch, when the control-flow jumps to this slot, the INT3 instruction will be executed, and a software exception will be raised. The exception handler will try to adjust the conflict entry chain to free this slot.

If it is failed to find an entry chain, the related indirect branch will be handled by hash mechanism. The entry chain scheme costs extra instructions, but these direct jump instructions will be handled by hash lookup mechanism. The entry chain.

4.3. Instruction Boundary Issue

On x86 platform, the control flow may jump to the middle of an instruction. To our knowledge, this technique is only used to access a prefix, such as LOCK. The following code sequence is excerpted from the benchmark 254 gap, showing an example that a branch instruction jumps across a LOCK prefix.

| 0x80a9807:  | je 0x80a980a |
| 0x80a9809:  | lock cmpxchg %ecx, 0x817a160 |

Our implementation has considered this across-prefix case. The source binary instruction at branch target address is checked for prefix, an instruction with prefix wouldn’t be selected as SPIRE table entry.

Many long x86 instructions can contain a shorter valid instruction, e.g. “mov %ch, 0” resides in “movl %esi, 0x08030000(%ebp)”[26]. Theoretically, a branch instruction can jump to the middle of an arbitrary instruction and continue executing correctly. Although this case may be impractical in real application, we also propose a heuristic scheme to deal with it. For example, when a 5-bytes long instruction at 0x8001000 will be selected as a table entry, the byte streams starting from 0x8001001 0x8001002, 0x8001003, 0x8001004 are decoded to instruction sequences. If all of these “middle instruction sequence” is illegal or much different from the original instruction sequence starting from 0x8001000, this table entry is considered as safe. Because we believe overlapping two different instruction sequences into one byte stream is unrealistic. This scheme isn’t perfect, but can reduce the possibility that jumping to middle of instruction makes SPIRE failed. Actually, we checked lots of applications, including productive software like office and web browser, and never found such case.

4.4. Implementation on Other Platforms

In this section, we will discuss how to implement SPIRE on other popular platforms.

As we mentioned before, there are two architecture related features impacting the implementation of SPIRE: 1) Page permission control is needed to prevent corrupt code pages from being accessed by other code. 2) Source binary instruction should be long enough to avoid entry overlapping. The former issue can be solved by shadow code space scheme described in section 4.1, so the instruction length becomes the main concern.

We classify the popular platform into two categories: variable-length ISA platform, represented by x86, and the fixed-length ISA platform, represented by ARM and MIPS. We will take x86 and ARM as examples to illustrate how to implement SPIRE on different DBT systems.

**ARM to ARM DBT.** Both guest and host instructions are fixed length, so the table overlapping issue wouldn’t exist. However, there is another issue in such RISC platforms: the offset of direct jump instruction is limited. In ARM platform, the jump offset is a 26-bit signed integer, thus the jump range is limited to 32MB. This range limit may be enough for embedded applications, but not enough for desktop or server applications. If the source binary code size is larger than 32MB, the trampoline cannot reach the code cache.

We modify our SPIRE mechanism to solve this problem. Figure 6 shows the structure of this variant. Instead of storing a redirecting trampoline to TPC, we directly put the TPC address in the SPIRE entry. Thus the workflow of SPIRE becomes: 1) Load the TPC from the memory address SPC + OFF. 2) Jump to this TPC. With this variant, we cannot leverage the software trap instruction to identify the unfilled entry. Instead, we fill the corrupted memory page with a specific handler address. When an unfilled table entry is loaded, the control flow will jump to the predefined handler. The source binary code space is omitted in the figure. To keep the transparency to self-reference code, the SPIRE table should also reside in the shadow space.

**ARM to x86 DBT.** On ARM to x86 DBT systems, the source instruction slot is always 4-bytes. A trampoline of host x86 platform needs 5-bytes, which is too long. To solve this problem, it can adopt the same mechanism with ARM to ARM DBT.

**x86 to ARM DBT.** The source instruction slot is variable length which may be shorter than SPIRE table entry, and ARM doesn’t own a short jump instruction to build entry chain as described in section 4.2. Storing the TPC in the entry cannot deal with the entry overlapping problem. So far, we haven’t found an effective way to implement SPIRE mechanism on such a DBT system.
5. Evaluation

Normalized execution time is the ratio of the execution time running on DBT system and on native machine, which is widely used to measure the performance of DBT system. We use “overhead reduction” to measure the performance improvement of our mechanism, the definition is as follows:

\[
\text{Overhead Reduction} = \frac{\text{Overhead}_{\text{before opt}} - \text{Overhead}_{\text{after opt}}}{\text{Overhead}_{\text{before opt}}} \times 100\%
\]

Due to the “Ret” instructions is highly predictable, there are a variety of efficient schemes to handle ret, such as “shadow stack” or “function cloning”[15]. Thus handling other indirect branches is a key factor to performance. Hiser et al. and Kim et al.’s evaluations [8, 13] show that indirect branch (excluding ret) handling mechanisms impact the performance significantly. The “Return Cache” mechanism of HDTrans also can handle “Ret” instructions efficiently, so our evaluation doesn’t apply SPIRE on “Ret” Instructions.

5.1. Experimental Setup

To get a convicting result, all of our experiments are running on the real machine. The experimental platform is shown in Table 2.

We implement our SPIRE technique in a high performance DBT system HDTrans[12]. HDTrans is a light-weight x86 to x86 binary translation and instrumentation system with a table-driven translator. It uses an very efficient hash lookup routine called SIEVE[27] to handle indirect branch, thus we selected the original HDTrans as the performance baseline.

The benchmarks are selected from SPEC CPU2000, SPEC CPU2006[28] and an individual C++ benchmark. All these benchmarks are indirect branch intensive, and widely used to measure the effect of indirect branch optimizing [29]. The details of benchmarks are shown in Table 1. We also evaluate all the SPEC CPU2000 benchmarks to show the side effect of SPIRE.

5.2. Performance Improvement

Execution Time Improvement. Figure 7 shows the performance comparison of several well-known DBT systems. Pin is a widely used dynamic binary instrumentation platform. It supplies a rich set of interface, which can be used to develop various pin-tools, such as memory checker, cache simulator. DynamoRIO is a dynamic optimization platform on x86 platform[10].

HDTrans-0.4 refers to the original HDTrans system, which use hash lookup to deal with indirect branch. HDTrans-SPIRE refers to the HDTrans system with our SPIRE mechanism. The original HDTrans doesn’t have a software prediction mechanism, for comparison, we implement software prediction technique in

![Figure 7: Performance comparison of DBT systems.](image-url)
HDTrans, called HDTrans-SP. The length of its prediction chain is 3, which can achieve the best performance among different configurations. The GeoMean in the figure refers to the geometric mean of benchmarks. The error bars indicate the 95% confidence intervals.

The performance results show that HDTrans-SPIRE achieves the best performance. Compared with original HDTrans using hash lookup mechanism (HDTrans-0.4), SPIRE achieves a significant performance improvement, speedups the average normalized execution time from 1.58 to 1.37, reduces the performance overhead by 36.2% on average, up to 51.4% on perlbench benchmark. The average execution time is improved by 15.4%.

Compared with the software prediction technique (HDTrans-SP), the SPIRE performs better at most benchmarks, and gets a 17.7% performance overhead reduction on average. Software prediction mechanism runs faster on “richards” benchmark, because the indirect branches in “richards” are highly predictable, which can achieve a 92.9% hit-rate in our evaluation.

DynamoRIO is designed as a dynamic optimization system with various optimizing techniques like trace generation. It runs much faster than original HDTrans. But HDTrans with SPIRE mechanism performs better than DynamoRIO, which proves the effectiveness of SPIRE. Pin mainly focuses on dynamic instrumentation for program analysis, so it runs relatively slower than others.

Dynamic Instruction Count Improvement. Because of the translation overhead and control transfer instruction handling overhead, DBT systems always suffer more dynamic instruction count than native execution. Our SPIRE mechanism can reduce the instruction count for handling indirect branch, which further improves the total dynamic instruction count. For example, a hash lookup takes more than 11 instructions on our implementation, while SPIRE only needs 6 instructions.

Figure 8 shows the dynamic instruction count normalized to native execution. Compared with hash lookup, SPIRE can reduce the total instruction count by 4.9%. The software prediction mechanism executes a bit more instructions than hash lookup, but achieves a better performance. Probably because the compare and conditional branch instructions used in software prediction take less CPU cycles than the instruction sequence of hash lookup. More micro-architecture issues will be discussed in the later section.

Entry Chain Scheme. The case that two branch targets are quite close is very rare, which mostly exists in the indirect branches related with “Switch-Case” statement. According to our evaluation, the rate of closed-targets is less than 0.01% in dynamic (with execution count weighted).

DynamoRIO is designed as a dynamic instruction count comparison.

Figure 9. Ratio and length of entry chain.

However, it’s hard to detect this case before it happens. For safety, SPIRE entry chain scheme is used to avoid the possibility of table entries overlapping. Figure 9 shows the ratio of the chained table entries and the average length of chains. According to our evaluation, 87.6% of table entries must be chaine. The average length of chains is 1.99. Fortunately, predictable direct jump instructions wouldn’t introduce too much overhead.

Performance on Other Benchmarks. We also evaluate SPIRE mechanism on all other benchmarks of SPEC CPU 2000 INT. All these benchmarks are not indirect branch intensive. For example, the dynamic rate of indirect branch of 164.gzip is 0.04%, 175.vpr is 0.03%. With online profiling scheme, SPIRE mechanism would rarely be triggered on these benchmarks. The results presented in Figure 10 illustrate that SPIRE mechanism has no performance side effect on benchmarks that are not indirect branch intensive. The results also show that indirect branch un-intensive benchmarks suffer less overhead than indirect branch intensive benchmarks, which proves the importance of indirect branch handling.

Figure 10. Performance on other benchmarks.

5.3. Memory Overhead

The shadow page mechanism introduces extra memory to store corrupted code pages or SPIRE table entries. On our x86 implementation, the shadow page space need a continuous virtual address space as large as source binary code space, but only the page
containing SPIRE table entries will take the physical memory. The online profiling mechanism further reduces the number of shadow pages. As shown in Figure 11, SPIRE consumes 5.6% more physical memory than before on average, which is acceptable.

![Figure 11. Memory overhead of SPIRE.](image)

5.4. Impact on Micro-architecture features

The performance of DBT system is closely related to pipeline events, such as cache miss and branch prediction miss [30], especially on a super-scalar architecture like x86. In this section, several important micro-architecture features are evaluated to show the effectiveness of SPIRE.

Because DynamoRIO and Pin doesn’t own the same code base with HDTrans, the behavior between them and HDTrans may be quite different. So we didn’t include DynamoRIO and Pin into micro-architecture evaluation.

**Branch Prediction Miss.** SPIRE routine contains one indirect jump and one or several direct jump instructions. The target distribution of the indirect branch is the same as it in source program. A typical hash lookup routine contains at least one compare-jump pair and always ends with one indirect branch. If the hash lookup routine is shared by all branches, the ending indirect branch will be hard to predict. With software prediction mechanism, when prediction hits, only one or several compare-jump pairs are needed, but when predictions fails, a full hash lookup would be called.

Figure 12 describes the result of branch prediction miss, including indirect branch and conditional branch. SPIRE achieves the best result.

![Figure 12. Branch prediction misses results.](image)

**I-Cache & I-TLB Miss.** SPIRE mechanism jumps from code cache to source binary code space, and jumps back to code cache, which may pollute the I-Cache and I-TLB. Because branch instructions has strong locality, most of indirect branch executions occur on hot targets of hot branches, so that the extra pressure for I-Cache and I-TLB won’t be too much, even in large code footprint applications. Meanwhile, SPIRE can reduce the code size of IB handling routines, which facilitates I-Cache and I-TLB hitting.

Figure 13 shows the results of L1 I-Cache miss. Software prediction has the lowest miss count, because the prediction routine is inlined in the code cache. SPIRE performs better than hash lookup mechanism. The I-TLB miss rate is too low to compare (less than 0.01 misses / 1k insns).

![Figure 13. L1I-Cache misses results.](image)

**D-Cache & D-TLB Miss.** SPIRE is an instruction-driven mechanism, and doesn’t contain memory access instructions. The original HDTrans also uses an instruction-driven hash lookup routine. Software prediction routine embedded the data into instructions too. All of them won’t impact the D-Cache and D-TLB significantly. As presented in Figure 14 and Figure 15, all three mechanisms have similar D-Cache and D-TLB misses results. It should be noted that the total instruction count of SPIRE is fewer than others, so the number of misses per instruction is relatively higher.

![Figure 14. L1D-Cache misses results.](image)
The experimental results of Hiser et al. indicated that the performance of IB handling mechanism is highly dependent on the implementation of underlying architecture[8], because different micro-architectures perform different on a same code sequence.

The DBT overhead are highest on ATOM processor. That’s probably because DBT introduces extra control transfer instructions, and increases footprint of the application, which may result in more cache misses or branch prediction misses. ATOM has a relatively simple micro-architecture, which has a low tolerance to these pipeline stall events.

To illustrate the effectiveness of our technique, we evaluate SPIRE on several different CPUs, including AMD Opteron processor for server, and Intel ATOM processor for netbook. The results shown by Figure 16 illustrate that SPIRE mechanism performs well on all of the experimental platforms.

The DBT overhead are highest on ATOM processor. That’s probably because DBT introduces extra control transfer instructions, and increases footprint of the application, which may result in more cache misses or branch prediction misses. ATOM has a relatively simple micro-architecture, which has a low tolerance to these pipeline stall events.

6. Conclusion

In this paper, we propose a novel mechanism called SPIRE, which can significantly reduce the indirect branch handling overhead.

SPIRE mechanism reuses the virtual address space of source binary code to build a dispatching structure for indirect branches. With SPIRE, the indirect branch can jump to the originally SPC address without address translation. The trampoline residing in the table entry will redirect the control flow to related TPC. Only 2–6 instructions are needed to handle an indirect branch execution.

We implemented our SPIRE mechanism on x86 platform. The experiments prove that compared with the previous techniques, SPIRE can significantly improve the performance, while introducing an acceptable memory overhead. The evaluation of micro-architecture features and the evaluation on different CPUs also illustrate the effectiveness of SPIRE. Furthermore, we discuss the implementing issues of SPIRE on different architectures.

SPIRE doesn’t need special hardware or instructions, so it can be widely used in DBT systems. It also can cooperate with other optimization mechanism. For example, it can be combined with software prediction. With this adaptive mechanism, the highly predictable IBs will be handled by software prediction, and the hard-to-predict IBs will be handled by SPIRE. We believe that the idea of SPIRE can also be applied on other occasions that need address translation.

References


