



Eliminate the Overhead of Interrupt Checking in Full-System Dynamic Binary Translator

Gen Niu

Institute of Computing
Technology, Chinese Academy of
Sciences
Beijing, China University of
Chinese Academy of Sciences
Beijing, China
niu18z@ict.ac.cn

Fuxin Zhang

Institute of Computing
Technology, Chinese Academy of
Sciences
Beijing, China University of
Chinese Academy of Sciences
Beijing, China
fxzhang@ict.ac.cn

Xinyu Li

Institute of Computing
Technology, Chinese Academy of
Sciences
Beijing, China University of
Chinese Academy of Sciences
Beijing, China
lixinyu.xyz@gmail.com

ABSTRACT

Dynamic binary translation (DBT) is a ubiquitous technique for program emulation, instrumentation and debugging. Full-system dynamic binary translators, which can run operating systems, are required to emulate interrupt delivery. Existing full-system dynamic binary translators use a simple scheme to do so, by attaching to each translated code block a prologue that checks for pending interrupts. However, this approach is inefficient, as interrupts are delivered infrequently, relatively to the execution of translated blocks, and therefore most of the interrupt checks are unnecessary and wasteful.

In this paper, we present an alternative novel and efficient interrupt checking scheme. Our main insight is that emulating interrupt delivery can be done more efficiently by interrupting the execution of translated code blocks as it eliminates the need for repeated checks for pending interrupts. To deliver interrupts in such a fashion, we employ binary rewriting and handle the various situations in which the blocks might be interrupted. We implement our interrupt delivery scheme on our full-system dynamic binary translator LATX, which is based on QEMU. The experimental result shows that it provides a speedup of 2.x% on average for SPEC CPU2000, hdparm and CoreMark benchmarks. Although our scheme increases the interrupt latency by up to 30%, the overall performance is not affected negatively in any benchmark.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Virtual machines**; **Communications management**; *Dynamic compilers*.

KEYWORDS

Dynamic Binary Translation, Interrupt Handling, Full System Emulation, Virtualization

ACM Reference Format:

Gen Niu, Fuxin Zhang, and Xinyu Li. 2022. Eliminate the Overhead of Interrupt Checking in Full-System Dynamic Binary Translator. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22)*, June 13–15, 2022, Haifa, Israel. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3534056.3534939>

1 INTRODUCTION

Dynamic binary translation is widely used in program emulation [2, 6, 9, 10, 31], instrumentation [3, 7, 19, 23], debugging [8, 12, 21], and additional use cases [30]. There are two types of dynamic binary translators (DBT) based on the use-case. The first one is *user-mode DBT*, which can only execute user-space programs. The DBT and the program must be compiled for the same operating system. The second one is *full-system DBT*, which can run a complete operating system. With hardware emulation, the operating system can be arbitrary. In general, the term **guest** refers to the program executed by DBT. And the term **host** refers to the platform to run DBT.

In a full-system dynamic binary translator, emulating interrupt delivery is an important component [25]. Modern operating systems are usually interrupt-driven to manage hardware resources. The state-of-the-art full-system DBT, QEMU [2], contains many software implementations of many hardware devices, such as serial and network chips. The emulation of devices is separated from the emulation of CPU, which is running on a standalone thread named "vCPU thread".

The vCPU thread is responsible for translating and executing the guest binaries. When an interrupt is delivered to the



This work is licensed under a Creative Commons Attribution International 4.0 License.

SYSTOR '22, June 13–15, 2022, Haifa, Israel

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9380-5/22/06.

<https://doi.org/10.1145/3534056.3534939>

guest, the vCPU thread should pause code translation and execution, and instead handle the pending interrupt. The simplest way to do so is to add an interrupt checking prologue into the translated blocks. The translated blocks are generated through binary translation and they are used to emulate the guest instructions' functionality. During the execution of the translated blocks, the vCPU thread continues to check if there are interrupts pending.

In fact, the interrupt happens much more infrequently compared to the execution of translated blocks. In addition to checking the pending interrupts inside the translated blocks, the vCPU thread also has additional opportunities to serve the pending interrupts. The interrupt checking code inside each translated block is wasteful if there is no interrupt pending. According to the profiling result in Figure 3, a large number of pending interrupt checks are unnecessary.

In this paper, we propose a novel and efficient scheme to eliminate the overhead of unnecessary interrupt checks. The main idea is to inform the vCPU thread when an interrupt should be delivered, instead of checking the pending interrupts repeatedly. When the vCPU thread receives the info, it will stop executing translated blocks and go to handle the interrupts. We utilize the Linux/Unix signals and run-time binary rewriting technique to inform the vCPU thread.

Implementing this idea introduces two challenges. The first challenge is that we need to make sure the interrupt is handled within a reasonable time. The second challenge is performing binary rewriting and synchronization correctly in the presence of multiple vCPU threads. Both are described in detail in Section 4.

The main contributions of this paper are as follows:

- We design a novel and efficient scheme for interrupt checks in a full-system dynamic binary translator. With this new approach, we can eliminate the unnecessary overhead during the emulation of CPU.
- We implement this scheme in our full-system dynamic binary translator LATX, which is based on the state-of-art DBT, QEMU 6.0. LATX runs on LoongArch [16] host and is able to run x86 guests. Our scheme can be easily applied to other full-system DBT.
- We present a detailed evaluation of LATX interrupt checking scheme and its impact on interrupt latency and performance.

The rest of this paper is organized as follows. Section 2 tells the basic concepts in binary translation and Section 3 shows the motivation example. Section 4 describes the design and implementation. Section 5 performs the evaluation. Then Section 6 gives a brief description of related works. Finally, Section 7 concludes the above.

2 BACKGROUND

In this section, we give some background on dynamic binary translation and full-system dynamic binary translator (DBT).

2.1 Dynamic Binary Translation

Dynamic binary translation is a ubiquitous technique to emulate programs. The instruction set architecture (ISA) to be emulated is the **guest's** ISA and DBT follows the **host's** ISA. By translating the guest binaries into host binaries, DBT can be cross-ISA. For example, it can translate ARM binaries into x86 binaries, and run them on the x86 platform.

The translation process is organized as code blocks. A piece of continuous guest instructions forms a single-entry/single-exit block. This guest block is translated into one host binary block. In this paper, we use the term **basic block** to represent this piece of guest instructions and the term **translated block** to represent the host binary block. DBT executes as a loop of the lookup-translation phase and execution phase. During the lookup-translation phase, DBT searches the existing translated blocks to find the next block to execute. If that fails, DBT fetches the guest binaries and generates the translated block. During the execution phase, DBT simply executes the translated blocks.

A context switch is used to switch between these two phases. In QEMU, the context switch code is named *prologue* and *epilogue*. The *prologue* is used to switch from the lookup-translation phase to the execution phase and the *epilogue* is the opposite. The reason context switches are necessary is that the application binary interface (ABI) is the guest and the host is different. The binary code of DBT follows the host ABI. For efficiency, the translated blocks do not follow the host ABI. We use the term **Block Context** to represent the context in which DBT is executing translated blocks and the term **DBT Context** to represent the context in which DBT is executing its own binaries, which follow the host ABI.

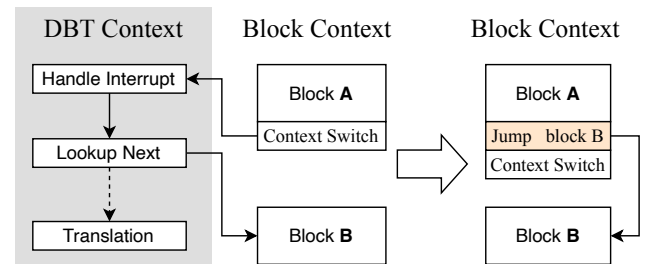


Figure 1: Block Chaining. In this example, *block A* ends with a direct jump targeting *block B*

Figure 1 shows a typical optimization named block chaining. A basic block usually ends with a branch, a call, a return, or other control flow instruction. After executing one block,

DBT needs to look up the next block. For an indirect jump, this is reasonable since the jump target is known only after its execution. But for direct jump, the lookup can be redundant since the jump target is fixed. In Figure 1, *block A* ends with a direct jump targeting *block B*. After executing *block A*, DBT will look up *block B*. If *block B* does not exist, it is generated through binary translation. Afterward, every time *block A* finishes executing, *block B* will be executed. To be more efficient, a direct jump instruction is inserted at the end of *block A*. After the execution of *block A*, the control flow is redirected to *block B* directly, without any context switch or lookup.

2.2 Full system DBT

A full-system DBT can run a complete operating system. To accomplish that, the hardware environment must be emulated. The state-of-the-art full-system DBT, QEMU, contains the software implementations of many hardware devices. Just like the real hardware, the emulated hardware also needs to be managed through interrupts.

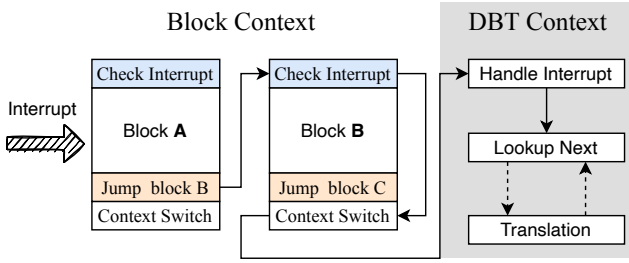


Figure 2: The Basic Interrupt Checking Scheme

Interrupt handling is processed in DBT Context as shown in Figure 1. Handling interrupt is a complex job and might change the control flow to the interrupt handler. Figure 2 shows the basic interrupt checking mechanism in full-system DBT. A small piece of host's binary code to check for pending interrupts is inserted at the beginning of each translated block. Once there is a pending interrupt, DBT will jump to the context switch code and switch to DBT Context. This interrupt checking mechanism works correctly with block chaining. It maintains precise interrupts for the guest because the interrupts are handled at the block boundary.

Modern full-system DBTs, such as QEMU, usually utilize multiple threads to help the emulation of hardware. The vCPU thread is responsible for managing the translated blocks. It executes the blocks, searches the blocks, and generates the blocks through binary translation. It is also responsible for handling the interrupts, which might change the control flow. The other thread, whose name in QEMU is "IO Thread", is responsible for the emulation of hardware devices.

When the emulated hardware needs to send an interrupt to the CPU, the IO thread will set an interrupt pending flag. Once the vCPU thread's interrupt checking code sees the flag, it will emulate the interrupt injection on the vCPU.

3 MOTIVATION

Existing full-system DBT utilizes the interrupt checking mechanism shown in figure 2. This is a simple and correct mechanism, which is easy to implement and not easy to get wrong. But it is inefficient because it makes the vCPU thread keep checking interrupts repeatedly. Compared to the execution of translated blocks, interrupts are delivered much more infrequently.

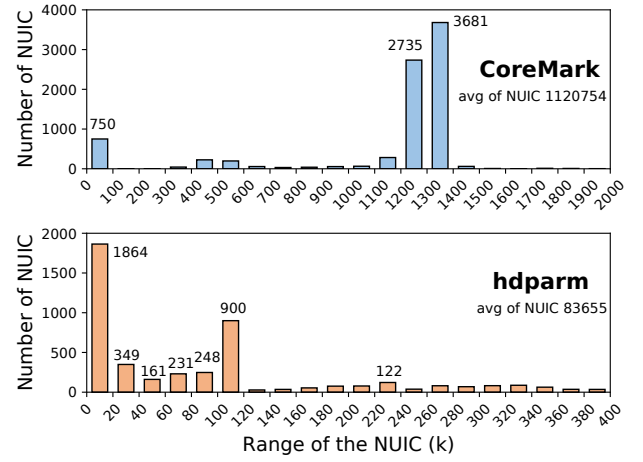


Figure 3: Number of Unnecessary Interrupt Checks

Figure 3 shows the profiling result of interrupt checks. When an interrupt check reports that there are interrupts pending, it is regarded as an **effective** interrupt check. Otherwise, it is regarded as an **unnecessary** interrupt check. We record the number of unnecessary interrupt checks between each two effective interrupt checks. We denote this number by NUIC. The NUIC is collected with a global counter that is increased by 1 at each unnecessary interrupt check. It is recorded and then cleared to zero at an effective interrupt check. The larger NUIC is, the more time is wasted on unnecessary interrupt checks.

For CoreMark [11], which is a small CPU-bound benchmark, the average NUIC is 1,120,754. The total number of effective interrupt checks in CoreMark is 8252. It is obvious that a lot of time is wasted on interrupt checks. For hdparm [18], which is a small IO-bound benchmark, the average NUIC is 83,655. This is much less than that in CoreMark, but it is still a large number. The total number of effective interrupt checks in hdparm is 4603.

We propose a novel scheme to remove the proactive interrupt checks inside the translated blocks. In the next section, we will give the details of our design.

4 DESIGN AND IMPLEMENTATION

The main idea is to inform the vCPU thread when an interrupt should be delivered to avoid repeated interrupt checks. We utilize the Linux/Unix signals as the communication mechanism. When the vCPU thread receives the info, it will stop executing translated blocks and switch to DBT Context to handle interrupts. As a result, interrupt checking code can be removed.

From another perspective, this is very similar to the behavior of real hardware. While the CPU is executing instructions, it does not check the pending interrupts proactively. Instead, the CPU automatically transfers to the interrupt handler when it receives an interrupt.

4.1 Unlink and Relink

The key operation to implement our scheme is called unlink. Unlink is operated on one specific translated block to make sure that the vCPU thread will switch to DBT Context after executing this block. Every time the vCPU thread switches to DBT Context, pending interrupts are checked and handled.

Due to the block chaining optimization, one translated block can jump to another directly through a jump instruction. Initially, this jump instruction's target is its next instruction. And it is followed by the context switch code, which redirects the control flow back to DBT Context. After block chaining, the jump instruction's target will be another block's entry address.

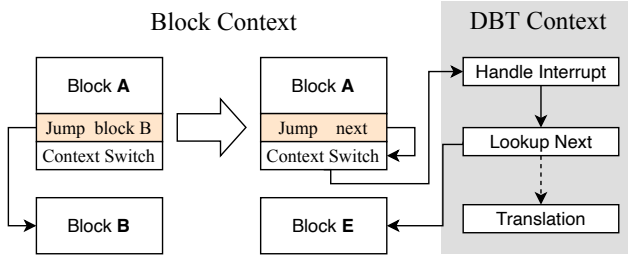


Figure 4: The Unlink Operation

Figure 4 shows the unlink operation. Unlink utilizes runtime binary rewriting technique to modify this jump instruction's destination from another block's entry to the address of its next instruction, which is the beginning of the context switch code. In LoongArch, this is done by writing the new jump instruction with an atomic store instruction. In addition, we bound one vCPU thread to one physical CPU core, then we can flush the instruction icache to finish the

synchronization between store and instruction fetch. For blocks that are not chained, unlink simply does nothing.

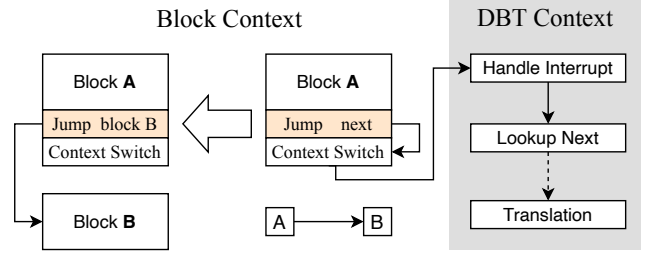


Figure 5: The Relink Operation

Figure 5 shows the relink operation. Unlink also records the original chaining data, such as block A jump to block B. Then relink can recover the block chaining according to the recorded data. Once the vCPU thread switches to DBT Context, relink will be done immediately. If no chaining data is recorded, relink will do nothing.

The main challenge here is to identify the correct translated block that needs to be unlinked. The vCPU thread is not executing translated blocks all the time. It does the translation job when it fails to look up the next block. For some complex instructions, it uses helper functions to finish the emulation.

The second challenge is the synchronization of unlink and relink when there are multiple vCPU threads. Modern full-system DBT usually utilizes the host multicore architecture to help the emulation of guest multicore architecture [6, 29]. When two vCPU threads are operating on the same translated block, the unlink and relink should be carefully processed.

4.2 Block Execution

The vCPU thread is an infinite loop of the lookup-translation phase and execution phase. Two important things are done between these two phases. One is handling exceptions and the other is handling interrupts. Every time the vCPU thread switches to DBT Context from Block Context, pending interrupts are checked and handled. During the execution phase, we need to unlink the translated block that is currently executing. After executing this block, the vCPU thread naturally switches to DBT Context. During the lookup-translation phase, we simply do nothing.

We utilize the signal mechanism to inform the vCPU thread when an interrupt should be delivered to the guest. The signal handler of the vCPU thread does the unlink operation. After that, the vCPU thread will go back to execute the unlinked block and finally switch to DBT Context. The key information that is needed is the program counter (PC) indicating the place where the vCPU thread got interrupted

by a signal. This PC is provided by the signal, so we call it *SGPC* (Signal PC) for convenience. Then we need to find out which block is currently running.

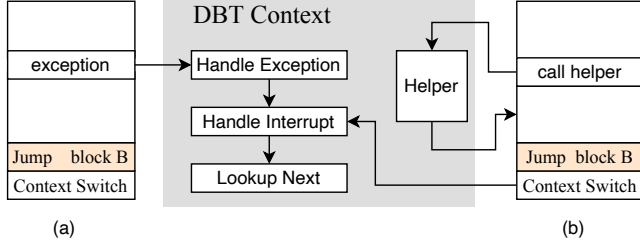


Figure 6: Block Execution

In the first situation, which is also the simplest one, a translated block contains no jump, branch, or call. If this kind of block is currently executing when the signal is delivered, the *SGPC* must be located inside its address range. We can find it by searching all the translated blocks to see if there is one block's address range that covers the *SGPC*.

In the second situation, which is described in Figure 6a, the execution of one block may cause exceptions. To emulate the behavior of guest exception, the vCPU thread switches to DBT Context before it completes the execution of the entire translated block. If this kind of block is currently executing when the signal is delivered, we can find it when the *SGPC* is before the exception. Otherwise, it is the same as the third situation.

Figure 6b describes the third situation, in which the signal is delivered while the vCPU executes the helper function. For some instructions, the DBT does not generate translated code to directly emulate them. Instead, it generates instructions to call a helper function, which finishes the emulation. The emulation of guest exception is also done by helper functions. If the *SGPC* is located inside one helper function, we can not find the currently executing block. In addition, helper functions usually call many other functions to finish the emulation work. That means we can not use the return address directly.

Fortunately, helper functions run in DBT Context. Therefore context switch must be done before its execution. We modify the context switch code to save the currently executing block into memory. We call this block *HPBlock*, which stands for "helper block". If the signal handler finds the *HPBlock* is not empty, it will unlink the *HPBlock* directly.

Finally, we need to distinguish the helper functions from other DBT Context. It is easy to do so since helper functions are only called from Block Context. We use one flag named *BlockExec* to indicate whether the vCPU thread is executing translated blocks. This flag is set before the execution phase and cleared after switching to DBT Context.

4.3 Interrupt Signal Handler

The signal handler of the vCPU thread is responsible for unlinking the translated block. Algorithm 1 describes the action that the interrupt signal handler takes. The *CCHi* stands for the highest address of code cache and the *CCLo* stands for the lowest address of code cache. If the *SGPC* is located inside the code cache, the vCPU thread must be executing translated blocks. Otherwise, the vCPU thread might be executing inside a helper function, or in other DBT Context.

Algorithm 1: Interrupt Signal Handler

Input: *SGPC*, *BlockExec*, *CCLo*, *CCHi*, *HPBlock*

```

block = null;
if BlockExec is 1 then          /* Block Context */
    if CCLo <= SGPC <= CCHi then
        block = search_block(SGPC);
    else
        block = HPBlock;
else                             /* DBT Context */
    do nothing;
if block != null then
    Unlink(block);

```

Overall, the signal handler is pretty simple. The most complex operation is searching the block with *SGPC*. QEMU uses the data structure *GTree* [28] to help search. According to its documents, *GTree* is an opaque data structure representing a balanced binary tree, which can perform lookups efficiently.

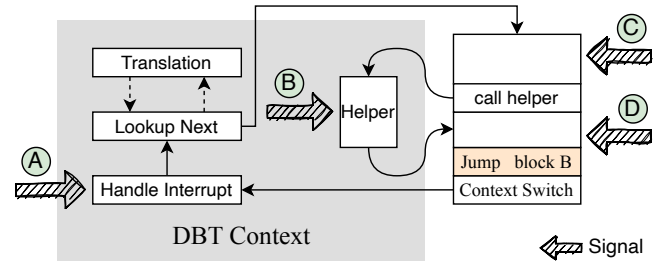


Figure 7: Interrupt Situation

Figure 7 shows four different situations in which the vCPU thread is interrupted by a signal. The signal handler takes different actions in different situations. For *Signal-A*, the *SGPC* is located outside the code cache, and the *BlockExec* flag is 0. The signal handler simply does nothing. For *Signal-B*, the *SGPC* is located outside the code cache, but the *BlockExec* flag is 1. This means the vCPU thread is executing a helper function so the *HPBlock* should be unlinked. For *Signal-C*

and *Signal-D*, the *SGPC* is located inside the code cache. The signal handler searches the blocks to find the block that is currently running and unlink it.

4.4 Recheck after Handling

For situation A shown in Figure 7, if an interrupt is delivered after the interrupt handling and before the execution of translated blocks, the signal handler does nothing. But the vCPU thread will switch into Block Context soon after, and it might take a long time to switch back to DBT Context.

To deal with this problem, we add the interrupt checking code in the context switch code. This small piece of context switch code is different from those used for helper functions. It is only used before the vCPU thread starts executing the translated blocks. The flag *BlockExec* is also set inside it. With this modification, the signal handler can safely ignore this situation. And this kind of context switch is not frequently executed with block chaining.

4.5 Multiple Threads

Modern full-system DBT usually utilizes the host multicore architecture to help the emulation of guest multicore architecture [6, 7, 29]. This is done by using multiple threads offered by the host operating system. Each thread is responsible for emulating one core of the guest. So there can be more than one vCPU thread. To maintain efficiency, the translated blocks are shared by multiple threads.

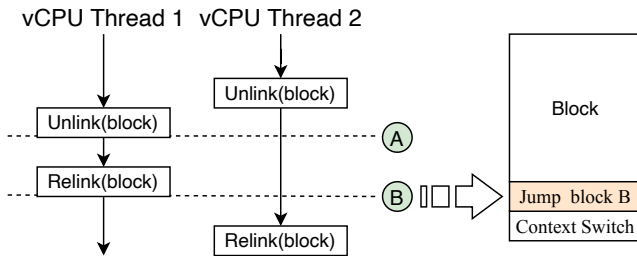


Figure 8: Unlink and Relink with two threads

The unlink and relink operations must be protected by locks, in case two or more vCPU threads are modifying the same block at the same time. But this is not enough. Consider the situation shown in Figure 8. At the *time A*, thread 1 and thread 2 both finish the unlink operation. Then thread 1 continues to execute the unlinked block and switch back to the DBT Context. At the *time B*, before thread 2 starts executing this block, thread 1 finishes the relink operation. Then thread 2 continues to execute this block, which is, however, not unlinked.

We establish a rule to address this problem. The key is to make sure that only the last relink operation works. In

the unlink operation, each thread saves its thread ID into the metadata of this block. In the relink operation, each thread clears its thread ID. When thread 1 is doing the relink operation, it can see the ID of thread 2 and know that thread 2 has not relinked it yet. Then thread 1 simply clears its thread ID and finish the relink operation. At the *time B* in Figure 8, this block is still unlinked. After thread 2 executed this block, it switches to DBT Context. When thread 2 is doing the relink operation, it can only see its own thread ID. Then it can safely relink this block and clear the thread ID. With the lock protecting the unlink and relink operation, setting or clearing the thread ID is atomic.

4.6 Other Architectures

Our scheme is not limited to LoongArch. We reuse many functions provided by QEMU, which is a retargetable DBT. The unlink and relink operation utilize the function designed for block chaining. Two things are done in this function. One is modifying the jump instruction atomically and the other is flushing the instruction cache for certain architectures. In some architectures, the run-time binary rewriting is more complex. For example, x86 requires INT3 to be set and more complicated synchronization [14].

In some platforms, the memory can not be writable and executable at the same time. We can map two different virtual memory spaces to the same physical memory space. The two virtual memory spaces have different privileges. The **RW** space is readable and writable but not executable. The **RE** space is readable and executable but not writable. The **RW** is used to store the translated blocks and enable run-time binary rewriting. The **RE** is used to execute translated blocks. QEMU utilizes the memfd and mmap to implement this strategy.

The other difference is the signal mechanism. Different architectures usually have a different definition of the data structure used by the signal handler.

5 EVALUATION

In this section, we evaluate the performance of LATX with and without our novel scheme. Since we modify the interrupt checking mechanism, we also evaluate the extra interrupt latency introduced by our scheme. In addition, we provide some profiling results of interrupt handling. Finally, we test our scheme with multiple-thread mode enabled.

5.1 Evaluation Setup

We evaluate our full-system dynamic binary translator LATX, which is based on the state-of-the-art DBT, QEMU 6.0. The host platform is loongson 3A5000 [17] processor using LoongArch [16] instruction set architecture. It is a 64-bit 12 nm

processor with 4 cores running on 2.5 GHz. The host operating system is Linux 4.19. The guest platform is i386 ubuntu server 16.04 with Linux kernel 4.4.

The benchmark we choose is SPEC CPU2000 [24] statically compiled with O3 optimization by GCC 4.8. We evaluate all its CINT benchmarks with reference input. We also run two additional microbenchmarks. One is a CPU-bound benchmark, CoreMark [11]. The other is an IO-bound benchmark, hdparm [18]. The baseline is LATX without our interrupt checking scheme. LATX is a full-system DBT based on QEMU 6.0 and is able to run x86 guests.

5.2 Performance

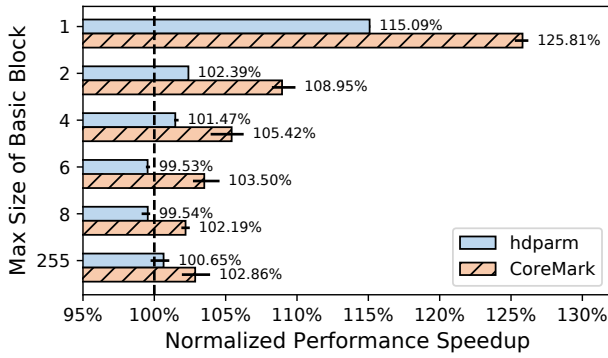


Figure 9: CoreMark and hdparm

We first evaluate the overall performance by running two benchmarks. CoreMark is run 5 times, hdparm is run 20 times, and the average score is presented. Figure 9 shows the results. We limit the maximum number of guest instructions of each basic block to 1, 2, 4, 6 and 8 to have a better understanding. The default value is 255.

When the maximum block size is limited to 1, the speedup is 25.8% for CoreMark and 15.0% for hdparm. This is reasonable because we remove the interrupt checking code in all blocks. The more blocks are executed, the more overheads are caused by the interrupt checking code. With the increase in block size, the number of blocks is decreased, and the number of the removed interrupt checking code is decreased too. Finally, the speedup is 2.8% for CoreMark when the maximum block size is 255.

For the IO-bound benchmark hdparm, there is nearly no speedup. This is because the interrupt latency is increased. And the result of hdparm is not stable enough. More details are described in section 5.3.

Figure 10 shows the performance improvement of the SPEC CPU2000 CINT benchmark. Each program is executed 3 times and the geometric mean is presented. The result is approximately the same as CoreMark. On average, when

the maximum block size is 1, the performance improvement is 31%. The overall performance improvement is 2.2% with the default block size. No benchmark shows performance slowdown.

Note that the 181 mcf has the lowest improvement among all benchmarks. Even when the maximum block size is limited to 1, the improvement is only 7%. This is because the behavior of 181 mcf is different from others. According to the profiling results of SPEC CPU2000 in the work [15], the cache miss ratio of 181 mcf is very high. This tremendously limits its potential to improve performance. The profiling results in Figure 15 also prove this.

According to the evaluation result, our scheme works better when the maximum block size is small. This is usually used when the full-system DBT is running in single-step mode.

5.3 Interrupt Latency

The interrupt latency is increased in our scheme since the interrupts are delivered through an additional communication mechanism. Consider the situation that an interrupt is delivered when the vCPU thread is executing inside one translated *block A*. In the original basic scheme, one flag is set by another thread and then the vCPU thread checks it at the beginning of the next translated block. In our scheme, a signal is sent to the vCPU thread. Then the vCPU thread needs to find the current executing translated block and unlink it. In both schemes, the translated *block A* runs to completion before the interrupt handler is invoked.

We now measure the extra interrupt latency caused by our scheme. We pick three key time points related to the interrupt latency. **T-send** is the time when an interrupt is sent to the vCPU thread. Usually, the interrupt is sent from another thread, but sometimes the vCPU thread could send an interrupt to itself. **T-exec** is the time when DBT starts to handle the pending interrupts. Emulation of interrupt delivery is a complex work, in which the DBT needs to check the type of the interrupt and find out if the current vCPU is ready to serve this kind of interrupt. Then DBT needs to emulate the behavior of the real guest CPU, such as walking the Interrupt Description Table (IDT) for x86 guests and switching the context to prepare for handling the interrupt. Finally, **T-done** is the time when DBT finishes handling the interrupt and gets ready to execute the interrupt handler.

We measure two types of interrupt latency. **Send-Exec** is the time starting from **T-send** to **T-exec**. And **Send-Done** is the time starting from **T-send** to **T-done**. In general, the first one represents the interrupt latency seen by DBT and the second one represents the interrupt latency seen by the guest operating system.

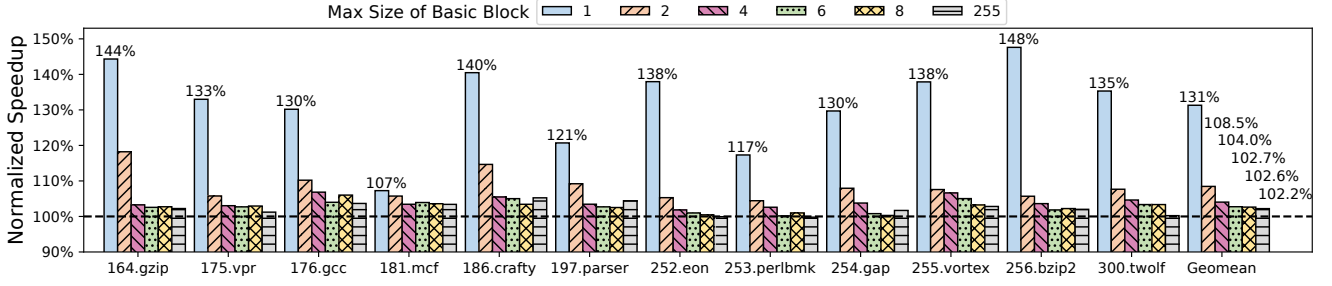


Figure 10: SPEC CPU2000 CINT Performance Speedup

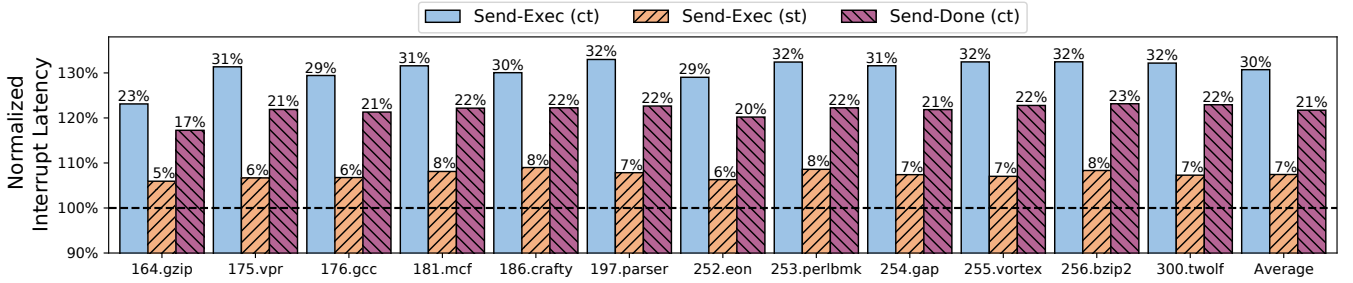


Figure 11: SPEC CPU2000 CINT Interrupt Latency The maximum size of basic block is 255.

Figure 11 illustrates the profiling results of interrupt latency. We still use the ratio of interrupt latency. For Send-Exec we distinguish between two different situations. The **cross-thread (ct)** situation means the signal is sent from another thread to the vCPU thread. The **same-thread (st)** situation means the signal is sent from the vCPU thread itself. For the latter situation, the signal is redundant. In our implementation, no signal is sent in the same-thread situation.

In general, the **Send-Exec (st)** shows the extra latency caused by the signal handler. And the **Send-Exec (ct)** shows all the extra latency, which includes the signal delivery overhead and the execution time of the signal handler. According to the result, the overall overhead is about 30%, and the signal handler is responsible for about 7%. The **Send-Done (ct)** shows the extra latency seen by the guest is about 21% overall.

Table 1: Interrupt Latency (μs)

Latency	baseline	our scheme	increase
Send-Exec(ct)	19.51	25.46	30%
Send-Exec(st)	3.69	3.96	7%
Send-Done(ct)	29.88	36.35	21%

The interrupt latency does increase, but the extra overhead is only 30% over the baseline. Table 1 is the value of interrupt latency measured in microseconds. According to the result in Figure 9 the increased interrupt latency does not impact the overall performance. And considering the whole process of interrupt handling, the interrupt handler gets rid of the burden of checking pending interrupts, which improves the overall performance. In other words, our scheme speeds up the common cases but introduces overheads in the infrequent events in which an interrupt is actually delivered to the guest. So that the overall performance is improved.

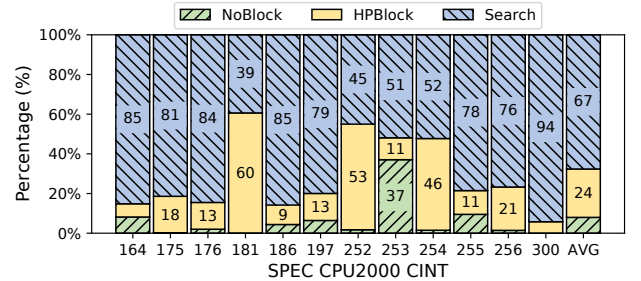


Figure 12: Signal Handler Profile. NoBlock: do nothing. HPBlock: simply unlink the HPBlock. Search: search to find the translated block that is currently executing.

If the signal handler does not need to search the blocks, it will be very simple and efficient. We profile the cases that are met by the signal handler. Figure 12 shows the profiling result of different cases. On average, there are 67% cases in which we need to search the currently executing block. And in 24% cases, the signal handler only needs to unlink the *HPBlock*. In the remaining 7% cases the signal handler simply does nothing.

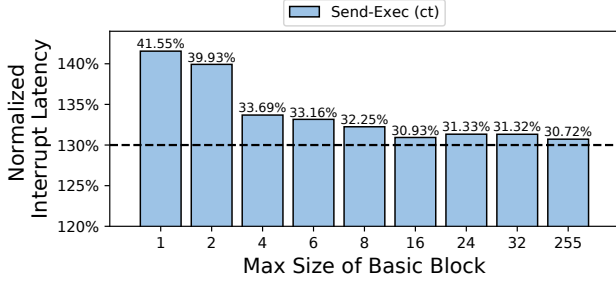


Figure 13: SPEC CPU2000 CINT Interrupt Latency Evaluate the affect of the maximum size of basic block. The value is the average interrupt latency normalized to baseline.

If the signal handler does need to search the blocks, the overhead will depend on the number of existing blocks. As the maximum block size decreases, the number of translated blocks increases, which enlarges the overhead of the signal handler. Figure 13 shows the result of the interrupt latency with a different maximum size of the basic block. When the maximum block size is limited to 1, the interrupt latency is increased by 42% compared to the baseline. Note that when the maximum block size is larger than 8, the extra overhead is always around 30%. The reason is that increasing the **maximum** block size does not increase the actual size of every block. Many basic blocks are naturally small because one basic block should end up with a control transfer instruction including jump, branch and call. Those control transfer instructions are pretty common in many programs. Moreover, in a full-system DBT, many privileged instructions should be considered as the end of a basic block.

Note that the signal delivery mechanism introduces considerably greater overhead than the signal handler. Recently, a new hardware extension, named **User-Space Interrupts** [5, 20] has been introduced. It enables user-space processes to send interrupts directly to each other. The testing result in work [20] shows that the user-space interrupt is 17 times faster than the signal mechanism. If we use the user-space interrupts as the communication mechanism in our scheme, the interrupt latency might be increased by only about 10%, which is completely negligible. But currently, the support is limited to certain architectures.

5.4 Sensitivity Test

The extra interrupt latency introduced by our scheme mainly comes from two sources. One is the delay caused by the signal mechanism and the other is the overhead caused by the signal handler. Since we can not control the first one, we should keep the signal handler efficient. In the worst case of our signal handler, searching the translated blocks, which is time-consuming, should be performed. But in all other situations, the signal handler is pretty simple.

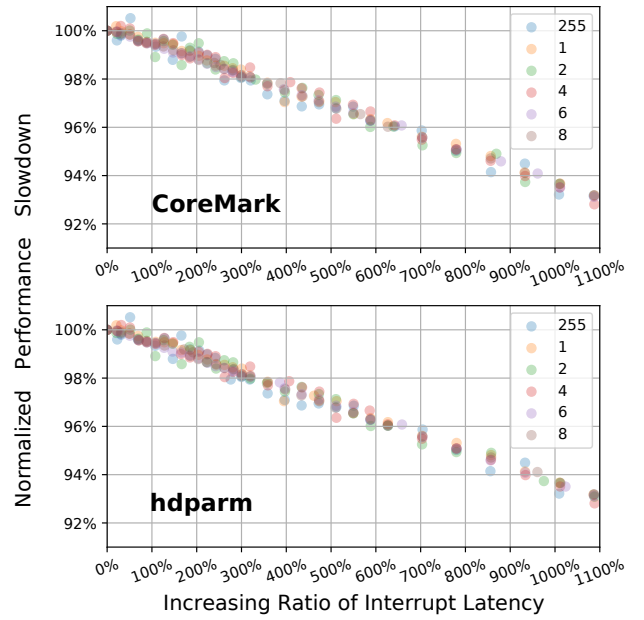


Figure 14: Signal Handler Overhead Test. The horizontal axis is the increasing ratio of the interrupt latency. For example, $x=100\%$ means the interrupt latency is doubled. The result is normalized to LATX with our unmodified scheme.

We manually increase the overhead of the signal handler to test its influence on the overall performance. The result is shown in Figure 14. Here we use LATX with our scheme as the baseline. Its interrupt latency Send-Exec(ct) is $25\ \mu s$ on average, which is shown in Table 1. With the interrupt latency increasing, the performance should be decreasing. In both CoreMark and hdparm, when the latency is increased by 100% (aka $50\ \mu s$), the performance is still more than 99%. Even if we increase the latency to $250\ \mu s$, which is 10 times longer than the baseline, the performance is still more than 94%. Note that our scheme only introduces a 30% overhead compared to the original scheme, whose average interrupt latency Send-Exec(ct) is $19\ \mu s$ shown in Table 1.

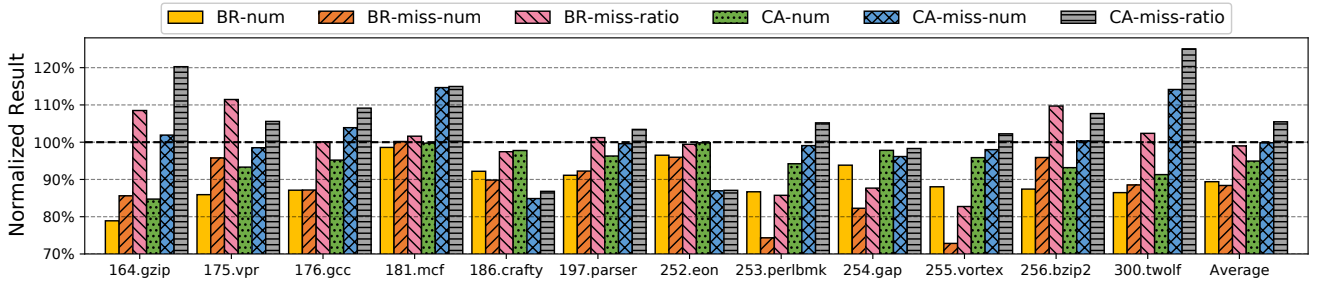


Figure 15: SPEC CPU2000 CINT Branch(BR) and Cache(CA) Profile Result

5.5 Speedup Analysis

Here we are trying to find out where the performance improvement comes from. A simple fact is that the number of dynamically executed host instructions is decreased. Our scheme removes the interrupt checking code from all translated blocks. This can explain the apparent performance speedup when the maximum block size is 1.

Figure 15 is the profiling result for branch and cache collected by the Linux perf [22] tool. On average the number of branch instructions is reduced by 11% and the number of cache access is reduced by 5%. This is reasonable since the interrupt checking code contains a load instruction and a branch instruction. For the behavior of the cache, the number of cache misses is not reduced at all, so the overall cache miss ratio is increased. For the behavior of the branch, the number of branch miss is reduced by 12% and the overall branch miss ratio is reduced by 1%.

The profiling result in Figure 15 also explains the behavior of 181.mcf as shown in Figure 10. Both the branch number and the cache miss are not apparently reduced, so the speedup of 181.mcf is pretty small.

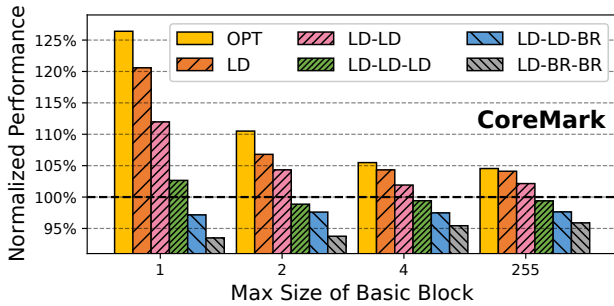


Figure 16: Modified Checking Codes Test

Another experiment is performed to evaluate the effect of the load and branch in the interrupt checking code. We manually modify the interrupt checking code to test its performance. For this matter, we ignore its function of checking

interrupt but focus on its overhead. The interrupt checking code of the baseline consists of one load instruction and one branch instruction. The **OPT** is our scheme that removes all the interrupt checking code. The **LD** contains only one load at the beginning of each translated block. Similarly, the **LD-LD** contains two loads and the **LD-LD-LD** contains three. The **LD-LD-BR** contains two loads and one branch and the **LD-BR-BR** contains one load and two branches.

Figure 16 shows the result of CoreMark with modified interrupt checking code. The **LD-LD-BR** and **LD-BR-BR** both contain more instructions than the baseline, so their performance gets decreased. Note that the **LD-BR-BR** is significantly lower, which means the branch instruction harms the performance more badly than the load instruction. The **LD-LD** simply replaces the branch instruction from the baseline, and its performance becomes better. On the other hand, the **LD** is very close to the **OPT** when the block size is 255.

A similar evaluation is performed on x86 to get further observation. We modify the standard QEMU 6.0 to run this experiment. The host platform is AMD Ryzen 7 Pro 4750G with ubuntu 20.04. The baseline is **LD1BR1**, which contains one load and one branch. We add more load instructions or branch instructions on the baseline to see how the performance gets decreased. The **LD2BR1**, **LD3BR1**, **LD4BR1** contains more load instructions and the **LD1BR2**, **LD1BR3**, **LD1BR4** contain more branch instructions.

Figure 17 shows the result of CoreMark with modified interrupt checking code on x86. In all the results, the **LD1BR2** is lower than the **LD1BR1** and the **LD1BR3** is lower than the **LD3BR1**. It is also true for the **LD1BR4** and the **LD4BR1**.

Overall, we can conclude from the above that the performance gained by our scheme is mostly from removing the branch instruction in the interrupt checking code, compared to removing the load instruction.

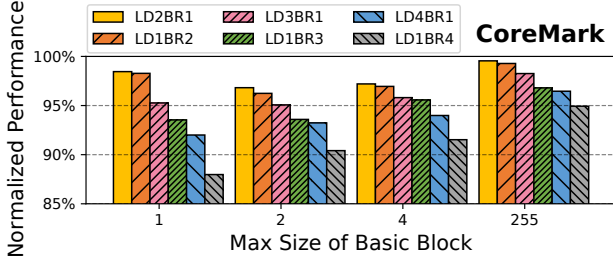


Figure 17: Modified Checking Codes Test on x86

5.6 Multiple Threads

The extra overhead under multiple thread environments mostly comes from conflicts. Conflict is defined as the situation where two or more vCPU threads are operating on the same translated block in their unlink and relink operation at the same time. The former experiments are performed with only one vCPU thread so no conflict occurs.

In fact, conflict is rare. Different vCPU threads are usually executing different translated blocks. Without any synchronization mechanism, different vCPU threads will probably not execute the same code at the same time.

We perform the experiment with SPEC CPU2000 running in multiple thread mode. The *test*, *train* and *reference* input dataset are used. In both two-thread mode and four-thread mode, no conflict is observed. Without conflicts, no more overhead will be introduced in multiple thread mode, which means our scheme is also efficient when a full-system DBT is running with multiple vCPU threads.

6 RELATED WORKS

In this section, we give the previous design of the interrupt checking mechanism in full-system dynamic binary translator. We also discuss that in the hardware virtualization targeting same-ISA virtualization.

Many full-system DBTs utilize the interrupt checking mechanism shown in Figure 2. We treat this mechanism as the basic scheme. Embra [31] says it detects the external interrupts on its own. QEMU [2] follows this basic scheme since its version 1.5. Captive [26, 27] runs a full-system DBT in guest mode using hardware virtualization. Its external interrupts are propagated as real interrupts into the guest system. But the interrupt handler simply causes a flag to be set to indicate the pending interrupts, which is the same as the basic scheme.

Some full-system DBT runs on bare metal directly. DAISY [10] emulates PowerPC and it responds to the external interrupt directly on real hardware. Its interrupt handler is an incrementally compiled version of the standard PowerPC

interrupt handler. The Transmeta Code Morphing [9] contains a roll-back mechanism to deal with external interrupts. When an interrupt is delivered, the Crusoe processor will roll back and switch to handle the interrupt automatically. They all don't need to check the pending interrupts inside the translated blocks.

Other schemes are trying to reduce the overhead of interrupt checking. Spink et al. [25] propose an algorithm to decrease the static and dynamic interrupt checks in a region-based DBT. But it only reduces the interrupt checks inside one region, which is generated from multiple blocks. The vCPU thread still needs to check interrupts on its own. QEMU [2] utilizes a similar zero-overhead interrupt handling mechanism before its version 1.5. But it suffers from serious race conditions [25] that all the blocks in the same chain of translated blocks are required to be modified.

Hardware virtualization has been developed for years to virtualize a complete operating system. Handling interrupts is also an important thing for virtual machines. In QEMU KVM, the IO thread sends the interrupt to the KVM vCPU thread through the inter-processor interrupt (IPI). But the IPI can cause VM exit, which is not efficient. Many architectures support posted interrupt [1, 4, 13] to directly send an interrupt into the guest system. Unlike the interrupts in DBT, the CPU in guest mode can handle the interrupt directly. Then the guest system automatically transfers to the corresponding interrupt handler.

Recently, a new hardware extension, named **User-Space Interrupts** [5, 20] has been introduced in some architectures. It allows a user-space process to directly send an interrupt to another user-space process. This new feature can be used in full-system DBT as the interrupt deliver mechanism.

7 CONCLUSION

In this paper we propose a novel and efficient interrupt checking scheme for a full-system dynamic binary translator. Our scheme removes the interrupt checking code from all translated blocks, which results in an overall performance speedup. The interrupts are sent through the Linux/Unix signal mechanism. The interrupt latency is slightly increased, but it does not harm the overall performance. Other communication mechanisms such as user-space interrupts can be used to further reduce the interrupt latency. Through additional experiments, we find that the performance gained by our scheme mostly comes from the elimination of branch instruction from the original interrupt checking code.

8 ACKNOWLEDGE

We appreciate the anonymous reviewers and our shepherd Nadav Amit for their insightful comments and feedback on the paper.

REFERENCES

- [1] ARM Holdings 2013. *ARM Generic Interrupt Controller Architecture version 2.0 - Architecture Specification*. ARM Holdings. <https://developer.arm.com/documentation/ihl0048/latest>.
- [2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10–15, 2005*. USENIX, Anaheim, CA, USA, 41–46. <http://www.usenix.org/events/usenix05/tech/freenix/bellard.html>
- [3] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) (CGO '03). IEEE Computer Society, USA, 265–275.
- [4] Edouard Bugnion, Jason Nieh, and Dan Tsafir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.
- [5] Jonathan Corbet. 2021. User-space interrupts. <https://lwn.net/Articles/871113/> [Online] Accessed on 2022-02-23.
- [6] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) (CGO '17). IEEE Press, 210–220.
- [7] Emilio G. Cota and Luca P. Carloni. 2019. Cross-ISA Machine Instrumentation Using Fast and Scalable Dynamic Binary Translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) (VEE 2019). Association for Computing Machinery, New York, NY, USA, 74–87. <https://doi.org/10.1145/3313808.3313811>
- [8] Marcos Cunha, Nicolas Fournel, and Frédéric Pétrot. 2015. Collecting Traces in Dynamic Binary Translation Based Virtual Prototyping Platforms. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools* (Amsterdam, Holland) (RAPIDO '15). Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. <https://doi.org/10.1145/2693433.2693437>
- [9] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiher, and Jim Mattson. 2003. The Transmeta Code Morphing[®] Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) (CGO '03). IEEE Computer Society, USA, 15–24.
- [10] Kemal Ebcioglu and Erik R. Altman. 1997. DAISY: Dynamic Compilation for 100Compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) (ISCA '97). Association for Computing Machinery, New York, NY, USA, 26–37. <https://doi.org/10.1145/264107.264126>
- [11] EEMBC. 2009. CoreMark - An EEMBC Benchmark. <https://www.eembc.org/coremark/index.php>
- [12] Jon Eyolfson and Patrick Lam. 2013. Detecting Unread Memory Using Dynamic Binary Translation. In *Runtime Verification*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–63.
- [13] Intel Corporation 2021. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 Chapter 29.6 Posted-Interrupt Processing*. Intel Corporation. <https://cdrdv2.intel.com/v1/dl/getContent/671447>.
- [14] Intel Corporation 2021. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 Chapter 8.1.3 Handling Self- and Cross-Modifying Code*. Intel Corporation. <https://cdrdv2.intel.com/v1/dl/getContent/671447>.
- [15] Shengmei Li, Buqi Cheng, Xingyu Gao, Lin Qiao, and Zhizhong Tang. 2009. Performance Characterization of SPEC CPU2006 Benchmarks on Intel and AMD Platform. In *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 02 (ETCS '09)*. IEEE Computer Society, USA, 116–121. <https://doi.org/10.1109/ETCS.2009.288>
- [16] Loongson Technology 2021. *LoongArch Documentation*. Loongson Technology. <https://loongson.github.io/LoongArch-Documentation/>.
- [17] Loongson Technology 2021. *Loongson 3A5000/3B5000 Product Manual*. Loongson Technology. <https://www.loongson.cn/productShow/32>.
- [18] Mark Lord. 2012. `hdparm - get/set SATA/IDE device parameters`. <https://linux.die.net/man/8/hdparm>
- [19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (PLDI '05). Association for Computing Machinery, New York, NY, USA, 190–200. <https://doi.org/10.1145/1065010.1065034>
- [20] Sohil Mehta. 2021. User Interrupts – A faster way to signal. <https://lpc.events/event/11/contributions/985/> LPC 2021.
- [21] Bojan Mihajlović, Željko Žilić, and Warren J. Gross. 2014. Dynamically Instrumenting the QEMU Emulator for Linux Process Trace Generation with the GDB Debugger. *ACM Trans. Embed. Comput. Syst.* 13, 5s, Article 167 (dec 2014), 18 pages. <https://doi.org/10.1145/2678022>
- [22] Ingo Molnar. 2009. Performance Counters for Linux. <https://lwn.net/Articles/337493/> [Online] Accessed on 2022-02-23.
- [23] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. <https://doi.org/10.1145/1273442.1250746>
- [24] SPEC. 2000. SPEC CPU2000 benchmark suite. <http://www.spec.org/cpu2000/>
- [25] Tom Spink, Harry Wagstaff, and Björn Franke. 2016. Efficient Asynchronous Interrupt Handling in a Full-System Instruction Set Simulator. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems* (Santa Barbara, CA, USA) (LCTES 2016). Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/2907950.2907953>
- [26] Tom Spink, Harry Wagstaff, and Björn Franke. 2016. Hardware-Accelerated Cross-Architecture Full-System Virtualization. *ACM Trans. Archit. Code Optim.* 13, 4, Article 36 (oct 2016), 25 pages. <https://doi.org/10.1145/2996798>
- [27] Tom Spink, Harry Wagstaff, and Björn Franke. 2020. A Retargetable System-Level DBT Hypervisor. *ACM Trans. Comput. Syst.* 36, 4, Article 14 (may 2020), 24 pages. <https://doi.org/10.1145/3386161>
- [28] GTK Development Team. 2021. Struct Tree. <https://docs.gtk.org/glib/struct.Tree.html> [Online] Accessed on 2022-02-23.
- [29] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: A Scalable and Portable Parallel Full-System Emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) (PPoPP '11). Association for Computing Machinery, New York, NY, USA, 213–222. <https://doi.org/10.1145/1941553.1941583>
- [30] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique – A Survey on Binary Rewriting. *ACM Comput. Surv.* 52, 3, Article 49 (jun 2019), 37 pages. <https://doi.org/10.1145/3316415>
- [31] Emmett Witchel and Mendel Rosenblum. 1996. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Philadelphia, Pennsylvania, USA) (SIGMETRICS '96). Association for Computing Machinery, New York, NY, USA, 68–79. <https://doi.org/10.1145/233013.233025>