Efficient and Scalable Cross-ISA Virtualization of Hardware Transactional Memory

Wenwen Wang
University of Georgia, USA

Pen-Chung Yew
University of Minnesota, USA

Antonia Zhai
University of Minnesota, USA

Stephen McCamant
University of Minnesota, USA

Abstract

System virtualization is a key enabling technology. However, existing virtualization techniques suffer from a significant limitation due to their limited cross-ISA support for emerging architecture-specific hardware extensions. To address this issue, we make the first attempt at hardware transactional memory (HTM), which has been supported by modern multi-core processors and used by more and more applications to simplify concurrent programming. In particular, we propose an efficient and scalable mechanism to support cross-ISA virtualization of HTMs. The mechanism emulates guest HTMs using host HTMs, and tries to preserve as much as possible the performance and the scalability of guest applications. Experimental results on STAMP benchmarks show that an average of 2.3X and 12.6X performance speedup can be achieved respectively for x86_64 and PowerPC64 guest applications on an x86_64 host machine. Moreover, it can attain similar scalability to the native execution of the applications.

CCS Concepts • **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**; **Dynamic compilers**; **Virtual machines**; *Multithreading*; • **Hardware** → Emerging languages and compilers.

Keywords System Virtualization; Cross-ISA; DBT; HTM

ACM Reference Format:
Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2020. Efficient and Scalable Cross-ISA Virtualization of Hardware Transactional Memory. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3368826.3377919

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377919

1 Introduction

System virtualization has played a critical role in many important applications [4, 24, 37, 46, 68, 69, 73]. However, due to the rapid evolution of processor hardware, existing virtualization techniques suffer a fundamental limitation from the lack of or limited support for cross-ISA virtualization, particularly, in emerging ISA-specific hardware extensions. For example, Intel x86 provides transactional synchronization extensions (TSX) [36] to support transactional memory, and IBM POWER8 provides similar support but with different implementations [41]. In another example, to support more secure execution environments, Intel x86 provides software guard extensions (SGX) [34], while ARM offers TrustZone [6]. And to a lesser extent, the SIMD instruction subsets have evolved across several different generations of the same ISA or different ISAs [5, 33, 35].

Cross-ISA virtualization of these hardware extensions is essential to build a transparent, efficient, and secure virtualization environment. This is crucial for workload migration, isolation and consolidation across machines with different ISAs, which are increasingly populating existing cloud environment [8, 9, 60]. For instance, Amazon launched elastic compute cloud (EC2) instances powered by ARM processors in 2018 [1]. Therefore, it is imperative to enhance existing virtualization techniques for such hardware extensions. In this paper, we focus on *hardware transactional memory* (HTM), which has become a permanent part of many modern architectures, e.g., Intel's Haswell, Skylake, and IBM's POWER8 and zEnterprise EC12 (zEC12). As more and more applications are taking advantage of HTM [25, 51, 53, 55, 70, 72, 76], virtualizing HTM across ISAs becomes necessary when such applications are migrated across machines with different HTM implementations.

Dynamic binary translation (DBT) is the cornerstone of cross-ISA virtualization [64]. In general, a DBT system can virtualize the execution environment of a *guest* machine on a physical *host* machine with a different ISA. By executing the host binary code translated from the guest binary code, the DBT system can emulate the functionality of the guest application on the host machine. Notable DBT systems include QEMU [11], Transmeta [17], IA-32 EL [7], and Dolphin [2]. In fact, many dynamic binary instrumentation (DBI) systems such as Intel Pin [44], DynamoRIO [13], and Valgrind [49],

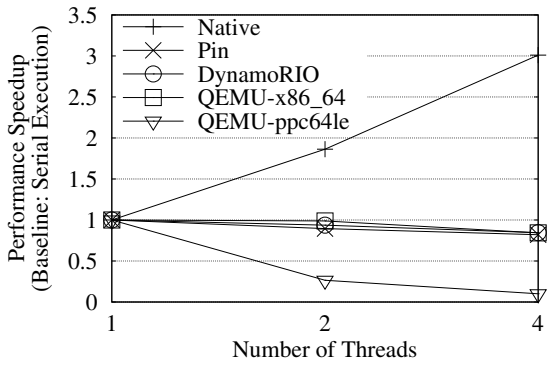

Figure 1. Poor scalability of existing popular DBT/DBI systems without HTM support.

also use techniques and infrastructures similar to DBTs even though their guest and host binaries are in the same ISA.

Due to the lack of efficient virtualization support for HTMs, most existing DBT systems suffer from poor efficiency and scalability. Figure 1 shows the scalability of the application *genome* from the STAMP benchmark suite [47], using existing popular DBT/DBI systems[1]. Here, the host machine is a Quad-Core Intel Xeon E5-1620 v4, which supports Intel transactional synchronization extensions (TSX) [36]. We respectively emulate x86_64 and PowerPC64 guest binary code using QEMU-x86_64 and QEMU-ppc64le ("le" means the little-endian mode) running on the host machine. We also run Intel Pin and DynamoRIO on *genome*'s x86_64 binary without any instrumentation. As shown in the figure, existing DBT and DBI systems cannot scale the performance for a guest application with HTMs in it. In this paper, we mainly focus on DBT systems. Additional work might be required to integrate our mechanism into existing DBI systems, but the main approach is the same.

To this end, this paper presents an efficient and scalable cross-ISA virtualization mechanism for HTMs. It leverages host HTMs to emulate guest HTMs by translating guest hardware transactions (HTs) into host HTs. A software-based emulation scheme is employed during the translation stage to reduce the abort-causing interferences from the translator to the translated host HTs (see §3 for more details). Several optimizations are explored to further reduce the abort ratios of the host HTs. A prototype of such a mechanism has been implemented in an extensively-used DBT system QEMU [11]. Experimental results on benchmarks from STAMP show that an average of 2.3X and 12.6X performance speedup can be achieved for x86_64 and PowerPC64 guest binaries on an x86_64 host machine, compared to the original QEMU without HTM support. Moreover, it can also attain similar scalability to the original native execution on the host machine.

[1]We also evaluated Valgrind (version 3.13), but most of the STAMP benchmarks failed when the number of threads exceeds one.

In summary, this paper makes the following contributions:

- **An effective cross-ISA HTM virtualization mechanism.** We propose to emulate guest HTMs by leveraging host HTMs for improved performance and scalability. To the best of our knowledge, this is the first effort to virtualize HTMs across ISAs that leverages host HTMs.
- **A practical prototype implemented on a real DBT system.** We implement the proposed mechanism using QEMU, which is a widely-used DBT system, to demonstrate the feasibility and effectiveness of such a cross-ISA HTM virtualization mechanism.
- **Comprehensive experiments.** We conduct a number of experiments on benchmarks from STAMP, which show that the emulation with HTM support can achieve a significant performance speedup and similar scalability to the native execution on the host machine.

The rest of this paper is organized as follows. Section 2 describes the background of HTM and DBT. Section 3 identifies the challenges to virtualize HTM in cross-ISA DBT systems. Section 4 presents our proposed HTM translation mechanism. Section 5 explains the implementation details. Section 6 shows the experimental results. Section 7 discusses related work. And Section 8 concludes the paper.

2 Background: HTM and DBT

Transactional memory (TM) is a concurrency control mechanism that attempts to ease concurrent programming, in particular, for synchronizing accesses to shared variables among threads using transactions, as well as to take advantage of the increasing number of cores for a higher performance.

Based on different implementations, TM can be classified into software transactional memory (STM) and hardware transactional memory (HTM). Compared to STM, HTM suffers much less performance overhead and is provided to programmers through an extension to existing ISAs. For instance, the Intel TSX provides three machine instructions: `xbegin`, `xend`, and `xabort`, to begin, end, and abort hardware transactions (HTs), respectively.

After a transaction is started, the hardware keeps track of memory *loads* and *stores* executed in this transaction, and detect access conflicts using its cache coherence protocol [48]. The granularity of the conflict detection is a *cache line*. If the same memory location is accessed by two concurrent transactions, and at least one of the accesses is a *store*, the transactions are in conflict and one of them is aborted. The aborted transaction is then rolled back to the beginning of the transaction and the execution is restarted. Although the interfaces of HTMs vary among different ISAs, the semantics are very similar. Next, we describe the semantics shared by most HTMs available in commodity processors.

Best-Effort HTM. It is important to note that, on today's off-the-shelf processors that support HTMs, the hardware

```
1: ret = HTM_begin;
2: if (ret == HTM_BEGIN_SUCCESSFULLY) {
3:     if (htm_lock is not free)
4:         HTM_abort;
5:     /* transactional code */
6:     HTM_end;
7: } else { /* fallback path */
8:     acquire (htm_lock);
9:     /* non-transactional code */
10:    release (htm_lock);
11: }
```

Figure 2. A software-defined fallback path is required to guarantee forward progress of HTM.

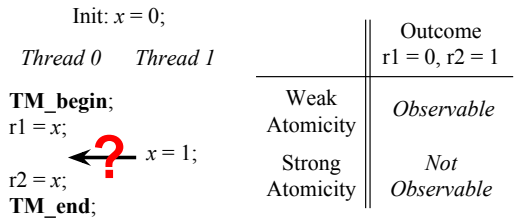

Figure 3. An example for weak and strong atomicity.

provides no guarantee that a transaction will eventually commit, i.e., hardware only supports a *best-effort* HTM. Programmers thus must *always* provide an alternative code sequence in the fallback path to guarantee forward progress. This is the main reason why existing DBT/DBI systems such as QEMU and Intel Pin can still emulate guest applications with HTMs even without using host HTM support.

Figure 2 shows an example of how to program correctly with an HTM. In the example, **HTM_begin** delimits the start of an HT (line 1). If the transaction is started successfully, it begins to execute the code in the transaction (line 2). During the execution, the lock *htm_lock*, which is used in the fallback path to protect its non-transactional code (line 8–10), is examined to see if it has been acquired (line 3). If yes, the transaction is aborted (line 4). Otherwise, the normal transactional code is executed (line 5). Note that it is necessary to track *htm_lock* in the transaction to synchronize its execution with the non-transactional code. Otherwise, inconsistent data could be accessed by the transaction. If the transaction is completed successfully, the execution results of the transaction are committed via **HTM_end** (line 6). During the execution of the transactional code (line 5), the transaction will abort if an access conflict is detected. When it aborts, the execution is rolled back to the beginning of the transaction (line 2). The condition in the *if statement* (line 2) will fail, and the execution will enter the fallback path (line 8), which is essentially a critical section protected by *htm_lock*. In practice, as mentioned in previous work [19, 70], *htm_lock* is typically a *coarse-grained* lock to simplify the multi-thread programming. This is one of the main reasons why existing DBT/DBI systems cannot achieve scalable performance with an increased number of threads.

It is worth noting that, in the above example, there is only one try to the transaction. That is, the transactional code (line 5) is executed at most once, no matter whether the transaction is committed successfully or aborted. In practice, it is possible to retry the transaction multiple times to improve its chance of a successful commit. To this end, a branch statement can be added on the fallback path (before line 8) to redirect the execution flow to restart the transaction (line 1).

Strong Atomicity. A TM implementation can provide two distinct atomicity semantics, i.e., *weak atomicity* and *strong atomicity* [45]. In weak atomicity semantics, transactions are atomic *only with respect to other transactions*, but not with the non-transactional code. That is, the execution of non-transactional code may interleave (and access the shared data) with a transaction. In contrast, strong atomicity semantics enforce atomicity with respect to both other transactions and non-transactional code. The example in Figure 3 illustrates the difference between them. As shown in the example, the execution of the transaction, which is surrounded by **TM_begin** and **TM_end**, can be interleaved with the non-transactional code in weak atomicity, and thus the outcome "r1 = 0, r2 = 1" is legal and observable. But, this outcome is not observable under a strong atomicity semantic because the execution of the transaction is atomic and cannot be interleaved with non-transactional code.

Although the strong atomicity is more intuitive than the weak atomicity from a programmer's perspective, most STM implementations in practice do not provide a strong atomicity semantic due to the significant overhead required to monitor memory accesses in non-transactional code. In contrast, HTM can leverage the cache coherence protocol to track memory accesses inside and outside of a transaction. As a result, HTM almost by default offers strong atomicity semantics, which makes HTM easier to program in addition to the potential benefit of higher performance.

Granularity of Conflict Detection. As existing HTMs leverage cache coherence protocols to detect access conflicts, the granularity of the conflict detection is by default a *cache line*. It is thus possible that two memory accesses to the same cache line can be detected as conflict even though they are not accessing the same memory location. This is also called *false sharing* or *false conflict* in previous work [42, 61]. Table 1 shows the granularity of conflict detection on different architectures that support HTM [48]. As shown in the table, different architectures employ different conflict detection granularities. For instance, the conflict detection granularity for Intel Haswell processors is 64 Bytes.

Table 1. HTM features of different architectures.

	Haswell	POWER8	zEC12
Conflict Granularity	64 B	128 B	256 B
Load Capacity	4 MB	8 KB	1 MB
Store Capacity	22 KB	8 KB	8 KB
Abort Reasons	6	11	14

Transaction Capacity. The transaction capacity is the maximum amount of memory data that can be accessed in a transaction without triggering an abort. For HTM, it is limited to the amount of the hardware resources available to keep track of memory accesses for conflict detection, and to buffer transactional *stores*. Table 1 also shows the transaction capacities of different architectures. In general, the *load capacity* is larger than the *store capacity* because *stored* data needs to be buffered to allow the recovery of the machine state in case the transaction is aborted.

Abort-Reason Code. To facilitate the debugging of an HTM program, hardware typically provides an abort-reason code to tell the reason why a transaction is aborted, such as a memory access conflict or a capacity overflow. It is a helpful hint that allows a program to determine whether to retry an aborted transaction or not. Table 1 shows the counts of abort reasons supported by different architectures. For example, Intel TSX supports 6 abort reasons, which are provided to the software through the `EAX` register.

Some Key Features in DBT. A DBT system dynamically translates binary code in its *guest* ISA and produces another binary code in its *host* ISA. The translated host binary code is stored in a software-managed *code cache* and can be reused in the same execution to mitigate the translation overhead.

In general, DBT systems translate a guest binary at the granularity of a *basic block* (or *block* for short), which is a sequence of guest instructions with only one entry and one exit [57]. After a block is translated, a user-level *context switch* will transfer the control from the translator to the generated code in the code cache. But, if the next block is not in the code cache, i.e., not translated yet, it will transfer the control back to the translator (via another user-level context switch) to translate the next block. Given the heavy performance overhead in such repeated context switches, each of which typically requires saving and restoring all host registers, the translated basic blocks are often chained together (a DBT optimization called *block chaining*) to allow the execution to stay within the code cache and reduce the number of context switches [29, 63]. Figure 4 gives a comparison between a native execution and a DBT emulation.

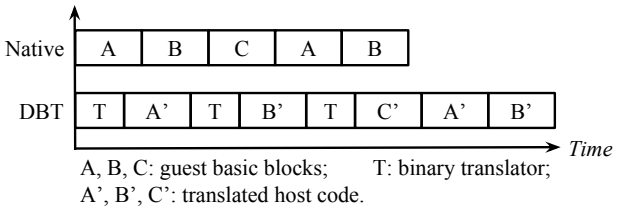

Figure 4. Native execution vs. DBT emulation.

3 Issues and Challenges

Inspired by the observation that different implementations of HTM in different architectures have very similar semantics, we leverage the host HTM to virtualize the guest HTM in a DBT system. To this end, we translate a guest HT into a corresponding host HT and emulate the guest HT by executing the host HT. Although the idea is rather intuitive, there are still several significant challenges to put it into practice.

Interference from DBT. A simple and intuitive HTM translation mechanism that requires no change to existing DBT framework is simply to translate guest transaction begin/end/abort instructions into corresponding host transaction begin/end/abort instructions. In fact, we did implement such a mechanism in an existing DBT system QEMU [11]. However, the experimental results show that almost *all* host HTs are aborted due to various reasons. After some investigation, we found most of the HTs are aborted during the binary translation phase. That is, the binary translator in the DBT system severely *interferes* with the host HTs.

There are typically three major reasons why the translator can abort a host HT. First, most of existing translators are not designed to work with HTs. They are not aware of the operations that are disallowed in an HT. For example, a translator can freely invoke system calls to allocate/deallocate memory during the translation process, which can cause an HT to abort persistently. Second, the host transaction capacity can be substantially exceeded due to undisciplined data accesses in existing translators. Third, to support multi-threaded guest applications, some DBT systems such as QEMU use multi-threaded translators to support concurrent translation [15]. This can potentially increase HT aborts, especially when two translator threads are active in two concurrent transactions.

If a host HT is aborted due to the interference from the translator, it does not help to simply retry the HT. This is because all of the blocks translated in an aborted transaction are discarded and need to be re-translated, which will only trigger more aborts as the cause of the abort remains the same. It is obvious that existing binary translators need to be refactored to support HTM. A naïve solution is to translate *all* basic blocks, instead of one basic block at a time, in a guest HT before emulating them on the host machine. However, this is impractical due to two reasons. First, translating all basic blocks in a guest HT at a time is non-trivial and often impossible. It faces the same challenges as in a *static* binary translation such as the code discovery problem for ISAs that have variable-length instructions (e.g., x86), and indirect branches with unknown branch targets [14, 40]. Second, it

could introduce a significant translation overhead, especially when the guest transaction has a very large code region and only a small portion of the code is executed at runtime. Such translation overhead is unacceptable to a DBT system.

Therefore, to overcome the above challenges, we propose a new translation mechanism. In particular, the mechanism employs a *software-based emulation* when a host HT is aborted due to interference from the binary translator. The software-based emulation is used to *dynamically* translate basic blocks and also emulate the semantics of a guest HT when it is not fully translated yet. Since all of its blocks only need to be translated once, a partially translated guest HT can be executed until the next untranslated basic block is encountered. This way, we can reduce the interference from the translator and cut down the aborts substantially.

To preserve strong atomicity semantics in our software-based emulation, we dynamically instrument memory accesses in the guest HT to detect access conflicts from other concurrent transactions that are also in the software-based emulation. Moreover, we leverage the page protection mechanism to detect conflicts from other non-transactional code. See §4.2 for more details.

Block Translation. One new challenge that can arise from the above-mentioned software-based emulation is that the very mechanisms can now interfere with the execution of the host HT after the guest HT is *fully* translated. The instrumented code becomes unnecessary for the translated host HT since such conflicts can be detected automatically by the host HTM hardware. Even worse, the data used to track memory accesses by the instrumented code can exceed transaction capacity of the host HTM. A naïve solution is to check the DBT status each time the instrumented memory accesses are executed, and invoke the instrumented code only if the status is in the software-based emulation. However, this can introduce a significant overhead because such a check may have to be done at every memory access.

In contrast, we propose to generate *two versions* of host code for each block with memory accesses in a guest HT. One is instrumented for software-based emulation and the other is un-instrumented for hardware transactional execution after it is fully translated. Note, the un-instrumented host code can also be used for non-transactional execution, as a block can be executed both inside and outside of transactions, e.g., blocks in a library. We then enhance the existing block chaining mechanism to chain host code in the same version together for improved performance. See §4.1 for more details.

Indirect Branches. Guest indirect-branch instructions can lead to frequent context switching between the execution in the code cache and the execution in the binary translator, because the guest branch targets can only be resolved until the indirect branch instructions are dynamically emulated. Such context switching can cause a large number of host HT aborts that need to be reduced.

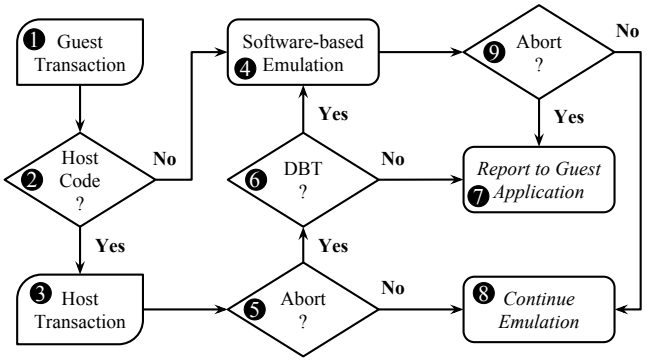

Figure 5. Overview of our HTM virtualization mechanism.

To this end, optimizations such as shadow stacks [32, 38, 39] and indirect branch tables [30, 31] are developed to mitigate the overhead. Unfortunately, these optimizations are typically not designed for HTs, and thus are unaware of the constraints posed by the host HTM. Therefore, additional work is required to adapt these optimizations to transactional execution. More details can be found in §4.3.

Retry Strategy. So far, we only discuss how to handle a host HT abort due to interferences from the translator. However, if a host HT is aborted due to a conflict access to the guest data, it is unclear what should be done next. One option is to retry the host HT several times until it can no longer make further progress, and report the abort to the guest application for further action. The problem with this option is that it can result in an *unfaithful* emulation, as the corresponding guest HT can also be aborted due to the same conflict access running natively on a guest machine. Moreover, the additional retries could introduce additional performance overhead if the host HT is finally aborted. Instead, we use a straightforward but efficient retry strategy. Each time when an HT abort results from a conflict access to the guest data, we immediately report the abort to the guest application and let the guest application decide whether it is necessary to retry the transaction or not. The host abort-reason code is translated to guest abort-reason code (i.e., a *reverse* translation) to provide the abort information for the guest application. More details can be found in §4.4.

4 HTM Virtualization

Figure 5 shows an overview of our mechanism. Before emulating a guest HT, we first check whether this guest HT (❶) has been translated into a host HT or not (❷). If yes, the host transaction is executed (❸). Otherwise, the software-based emulation is invoked to translate and emulate the guest transaction (❹). If the host transaction is aborted (❺), we need to check the abort reason to see if the abort is caused by the interference from the binary translator or not (❻). If yes, the software-based emulation is invoked to retry the transaction.

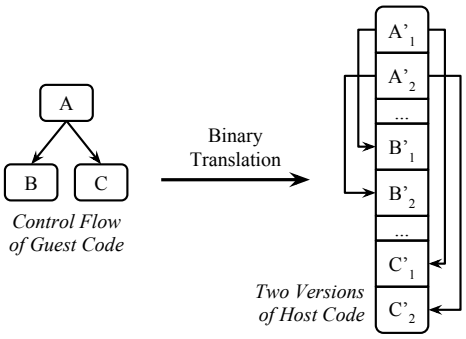

Figure 6. Block chaining for two versions of host code.

Otherwise, the abort is caused by the original guest application, we then report the abort reason to the guest application (❼). If the host HT is completed successfully, the emulation continues without interruption (❽). If an access conflict is detected during the software-based emulation (❾), the emulation will be aborted and the abort reason is reported to the guest application. Otherwise, the emulation continues.

4.1 Binary Code Translation

There are two main purposes for the software-based emulation: 1) translating basic blocks in a guest HT to avoid the interference from the translator to the execution of the translated host HT; 2) emulating the guest transaction during the binary translation process. We focus on the first in this subsection, and the second in next subsection.

For each basic block in a guest HT that contains memory accesses, two versions of the host binary code are generated for this block. In one version, each memory access is instrumented for the conflict detection in the software-based emulation. In the other version, there is no instrumentation and is used for non-transactional and hardware transactional execution. To accommodate two versions of the host code, we enhance the block chaining scheme in existing DBT systems to chain host code with the same version together, as shown in Figure 6. Each time when we need to chain host code of two guest blocks together, we respectively chain the host code with the same version of these two basic blocks together. Note that block chaining only happens in the software-based emulation or non-transactional execution because there will be no further translation/optimization in a host HT after all of its blocks are translated and chained together.

The correct emulation of *atomic instructions* is a key challenge for DBT systems [15, 71], especially when the guest and the host ISAs have different atomic instructions. However, when an atomic instruction is executed in an HT with strong atomicity, it can be handled as a couple of normal memory access instructions because the HT has already guaranteed its atomic semantics. We can thus emulate a guest atomic instruction as a couple of normal memory access instructions during the software-based emulation. This can substantially simplify the emulation as there is no need to execute host atomic instructions during the emulation.

4.2 Software Emulation of Strong Atomicity

To correctly emulate the semantics of guest HTs, our software-based emulation needs to provide strong atomicity. We dynamically instrument memory accesses in guest HTs to detect conflicts between concurrent transactions. For potential conflicts between transactional code and non-transactional code, we try to detect them by leveraging page protection mechanism available in modern hardware. Note that there is no need to detect conflicts between software-based emulation and host HTs because hardware can automatically detect such conflicts and abort HTs when detected.

For each memory access in a guest HT, we dynamically instrument it to check if it conflicts with any memory access in concurrent transactions. To emulate such conflict detection on the guest hardware, the granularity of the conflict detection is set to the size of a guest cache line, e.g., 64 Bytes for Intel Haswell guest architecture. For the guest memory region that corresponds to a guest cache line, a *lock* is assigned to synchronize concurrent transactional accesses to this region. Each time such a guest memory region is accessed in a transaction, we perform the conflict detection (i.e., it is an *eager* conflict detection policy) by checking if there is any access to this region from other threads. If yes, we further compare the types (i.e., *load* or *store*) of these accesses because concurrent transactional *loads* from different threads are allowed. If no access conflict is detected, the access is granted and recorded for future conflict detection. Otherwise, we abort the software-based emulation of this transaction and report the abort reason to the guest application.

To detect potential access conflicts between transactional code and non-transactional code, our software-based emulation leverages page protection mechanism, which is available in modern commercial hardware. Specifically, each guest memory page is set to *"read-only"* when it is read in a transaction at the first time. A page access violation (through a page fault) will be triggered when a memory access in the non-transactional code tries to write this page. Such writes are delayed until the conflicted transactions are completed.

Besides, for each guest memory region accessed in a transaction that corresponds to a guest cache line, a buffer is allocated to keep the results of stores in the transaction to avoid modifying the original memory locations directly. These buffered store results will be committed to their original memory locations if the transaction completes successfully, or discarded if the transaction is aborted. To guarantee the consistency in the commit process, all original (virtual) pages modified in the transaction are set to *"non-readable"* before the commit. We employ a rather commonly-used *parallel mapping* technique to remap a new *"writable"* (virtual) page for each modified (virtual) page to the *same* physical page [18, 27, 50]. This allows two *virtual* pages with

different protection settings, one with "non-readable" and the other with "writable," to be mapped to the same *physical* page. We can then commit the buffered store results through the new "writable" (virtual) page while the original "non-readable" (virtual) page can protect it from accesses in non-transactional code. This way, we can avoid access conflicts between transactional code and non-transactional code during the commit of buffered store results.

4.3 Handling Indirect Branches

To enhance the performance of supporting guest indirect branches, several optimizations have been adopted in DBT systems, e.g., using a shadow stack [32, 38, 39] and an indirect branch table [30, 31]. However, additional work is required to adapt them to the transactional execution.

Shadow Stack. A shadow stack is mainly used to support guest `return` instructions. The basic idea is to push the *guest return address* and its corresponding *host code address* to the shadow stack at a guest `call` instruction. When a `return` is encountered, the return address on the shadow stack is popped, and compared with the return address on the guest stack because it may be altered during the execution. If they match, the execution can be transferred to the host address saved on the shadow stack. To this end, additional host code is generated to maintain the pointer of the shadow stack. Unfortunately, this could potentially introduce additional pressure on a transaction's *store capacity* and result in unexpected transaction aborts. To handle this issue, we allocate the shadow stack with a *fixed offset* from the guest stack [16] and access it using the guest stack pointer with the fixed offset. Thus, the shadow stack pointer can be eliminated.

Indirect Branch Table. Each entry of the table is a pair of guest address and its corresponding host address. Initially, the table is empty. An entry is inserted when an indirect branch target is missed from the table. Each time an entry is hit, the table is updated to reduce the next lookup time. Obviously, this design may not be appropriate for transactional execution due to its frequent updates. Instead, we use a simple but effective approach that updates the table only from the *outside* of host HTs, i.e., after successful host HTs. This way, we can attain its benefit without introducing additional pressure on the store capacities of host HTs.

Note we only place the address of the un-instrumented version of the host binary code to the shadow stack and the indirect branch table to speedup the transactional execution. That is, such indirect branch optimizations are bypassed in our software-based emulation.

4.4 Retry Strategy and Abort Translation

Figure 7 shows how to emulate a guest transaction *begin* instruction using the host transaction *begin* instruction. Specifically, when a host transaction is aborted (line 4) and the abort reason is not due to the binary translator (line 7), we report the abort to the guest application and let it decide whether to retry the transaction or not (line 9). This simple retry strategy enables us to achieve not only a faithful emulation of a guest HTM from the perspective of an HT abort but also an efficient emulation, as will be shown in §6.1.

```
/* Emulation of Guest_HTM_begin */
1: ret = Host_HTM_begin;
2: if (ret == HTM_BEGIN_SUCCESSFULLY) {
3:     set_emulated_guest_register (success);
4: } else {  /* host transaction is aborted */
5:     if (the abort is due to the translator) {
6:         software_based_emulation ( );
7:     } else {
8:         translate_abort_reason_code (ret);
9:         set_emulated_guest_register (abort);
10:    }
11: }
```

Figure 7. Emulation of a guest HTM *begin* instruction, simplified for demonstration.

To facilitate the guest application to make a retry decision, we provide the *abort reason* of the host HT (in a coded number) to the guest application. Due to different kinds of abort reasons supported by different architectures, as shown in Table 1, we need to translate the host abort-reason code into the corresponding guest abort-reason code (line 8 in Figure 7). For instance, when an HT is aborted, the Intel Haswell processor tells the software if the transaction is likely to succeed on a retry. In contrast, the IBM POWER8 architecture informs the software that if the transaction abort is persistent, i.e., the abort is likely to recur on each execution. Obviously, these two abort reasons can be translated into each other albeit with an opposite semantic. This way, most of the key abort reasons can be translated between different architectures. However, there are still some abort reasons that are only supported by specific architectures. For example, IBM POWER8 provides an abort reason that tells if a transaction abort is due to an instruction fetched from a memory location that was written previously in a transaction. Though such abort reasons are rarely used in practice, we translate them into abort reasons that have relatively similar semantics.

5 Implementation

We have implemented the proposed HTM virtualization mechanism on QEMU [11]. The implementation takes x86-64 and PowerPC64 (with the little-endian mode) as the guest ISAs and x86-64 as the host ISA. The guest transaction instructions are translated into *helper function* calls, which are regularly used in QEMU to translate complicated guest instructions, e.g., floating-point instructions, to invoke corresponding host transaction instructions.

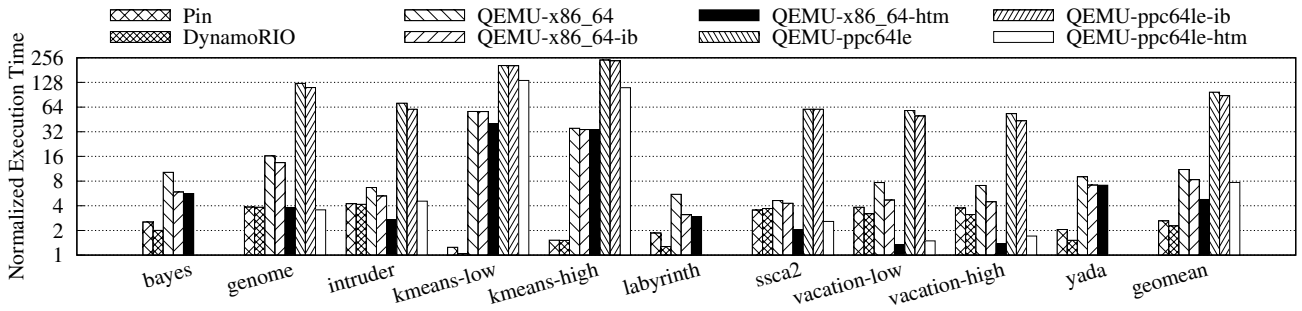

Figure 8. Normalized execution time with the native execution on the host machine as the baseline.

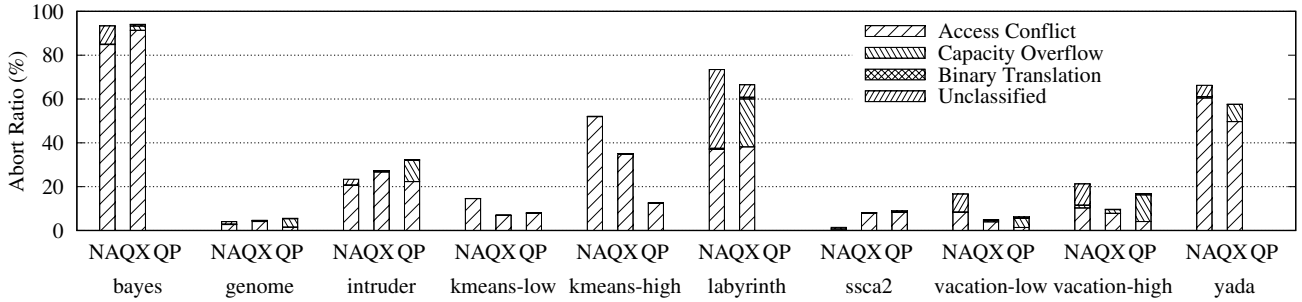

Figure 9. Abort ratios and reasons of host hardware transactions in native execution and on QEMU with proposed HTM virtualization mechanism. NA: native execution, QX: QEMU-x86_64-htm, QP: QEMU-ppc64le-htm.

In order to simplify our implementation, an explicit host transaction abort instruction with a specific abort code is issued when a basic block is required to be translated during the execution of host HTs. The abort instruction is used to indicate that this transaction abort is caused by the request of the translation. The instrumentation of the transactional memory accesses for the conflict detection in the software-based emulation is implemented at the TCG Op level, which is the intermediate representation used by QEMU for retargetable binary translation. Because there is no support for indirect branches in the original QEMU, we added the shadow stack and indirect branch table optimizations in our prototype. For each basic block ending with a guest function call instruction, we translate the block at the return address (which is known at the translation time) immediately after the translation of the current block to obtain the address of the translated host code. In addition, the indirect branch table is implemented as a hash table using the lowest ten bits of the guest address as the hash key.

6 Experimental Results

To evaluate our HTM virtualization mechanism, we compare the implemented prototype with the original QEMU. The experimental results of existing popular DBI systems, including Intel Pin [44] and DynamoRIO [13], are also included for reference. The system configurations we used in the evaluation are listed as follows:

- **Native:** native execution of the host binaries compiled from the source code on the host machine.
- **Pin:** execution of the native host binaries using Intel Pin without any instrumentation.
- **DynamoRIO:** execution of the native host binaries using DynamoRIO without any instrumentation.
- **QEMU-x86_64/ppc64le:** emulation of the guest binaries using QEMU on the host machine.
- **QEMU-x86_64/ppc64le-ib:** emulation of the guest binaries using QEMU with the indirect branch optimizations on the host machine.
- **QEMU-x86_64/ppc64le-htm:** emulation of the guest binaries using QEMU with the indirect branch optimizations and our HTM virtualization on the host machine.

Our evaluation uses the STAMP benchmark suite [47], which has been used widely to evaluate various TM implementations. The version of STAMP is an updated version of 0.9.10 [48], which has fixed nonessential transaction aborts in the original version for four benchmarks: *genome*, *intruder*, *kmeans*, and *vacation*. Each configuration mentioned above is evaluated with all benchmarks in the STAMP benchmark suite, except three benchmarks: *bayes*, *labyrinth*, and *yada* for QEMU-ppc64le(-ib/htm) due to random errors in the original QEMU when we run those three benchmarks with multiple threads on the host machine. In addition, we use the default parameters as specified in the README files of STAMP. For benchmarks *kmeans* and *vacation*, there are

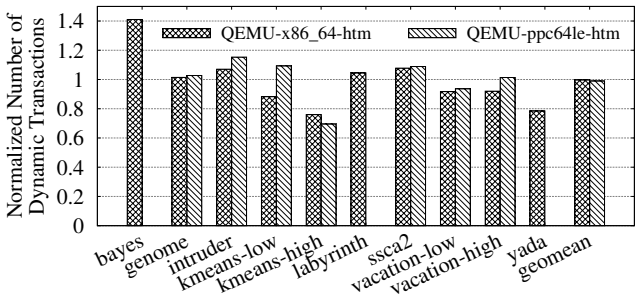

Figure 10. Normalized number of dynamic host HTs with the native execution as the baseline.

two sets of parameters corresponding to *low* and *high* contention among concurrent transactions, respectively. We use *kmeans-low/high* and *vacation-low/high* to denote them.

The host machine is equipped with a Quad-Core 3.5 GHz Intel Xeon E5-1620 v4 processor. The private L1 D-Cache and I-Cache are both 32 KB, the private L2 Cache is 256 KB, and the shared L3 Cache is 10 MB. The cache line size is 64 Bytes. The main memory is 32 GB, and the operating system is Ubuntu 16.04 with Linux-4.4.0. The host machine is set up exclusively to run the evaluated benchmarks. Also, each benchmark is run five times and their arithmetic means are used to reduce the influence of random factors.

6.1 Performance

Figure 8 shows the normalized execution time of each benchmark with its native execution time as the baseline. Here, the number of threads is four, which is the maximum number of physical cores on the host machine. As shown in the figure, we can improve performance for most of the benchmarks. For *vacation-low*, the speedup can be as high as 38.9X for PowerPC64 guest binaries. But, for some benchmarks, e.g., *bayes*, *labyrinth*, and *yada*, there is no noticeable performance improvement. This is because these benchmarks suffer very high transaction abort ratios in their original guest binaries, which can significantly offset the performance gained from our mechanism. Compared to the original QEMU, an average of 2.3X and 12.6X speedup are achieved for x86_64 and PowerPC64 guest binaries, respectively. This gives a favorable indication on the efficiency of our mechanism.

As shown in Figure 8, QEMU-ppc64le suffers a significant performance overhead compared to the native execution, i.e., 97.2X on average. As discussed earlier, it is because it does not utilize the host's HTM support. It has to emulate the fallback paths defined by the guest application, which typically employ a coarse-grained lock to guarantee the atomicity. Even worse, QEMU emulates PowerPC64 *atomic instructions* by stopping *all* concurrent threads to achieve the atomicity semantics. This is the most straightforward way to emulate atomic instructions supported by PowerPC64 on an x86-64 host with a different set of atomic instructions [15]. In contrast, with our mechanism, the fallback paths can be mostly avoided by emulating guest HTs with the host HTM.

Table 2. Percentages of software-based emulation (SBE) in emulated guest transactions and their abort ratios. QX: QEMU-x86_64-htm, QP: QEMU-ppc64le-htm.

	QX (%)		QP (%)	
	SBE	**Abort**	**SBE**	**Abort**
bayes	0.71	11.64	-	-
genome	0.06	0.78	0.01	4.92
intruder	< 0.01	11.67	< 0.01	11.32
kmeans-low	0.05	1.59	< 0.01	0
kmeans-high	0.01	2.07	< 0.01	0
labyrinth	1.18	10.53	-	-
ssca2	0.16	0.1	< 0.01	0
vacation-low	0.02	0	0.02	0
vacation-high	0.01	0	0.01	2.33
yada	< 0.01	11.86	-	-

Another interesting observation from Figure 8 is that existing popular DBI systems, such as Intel Pin and DynamoRIO, also suffer high performance overhead even without any instrumentation, i.e., 2.6X and 2.3X, respectively. More investigation is required to enhance the performance of these DBI systems. Figure 8 also shows that indirect branch optimizations can achieve good performance improvement for most of the benchmarks. But, such optimizations only aim to optimize the performance of a single thread, and thus cannot scale up as the number of threads increases.

Figure 9 shows the abort ratios of host HTs in native execution and QEMU with our mechanism. Note the missed bars for *bayes*, *labyrinth*, and *yada* are due to the random errors in original QEMU for PowerPC64 guest binaries. As shown, for some benchmarks, e.g., *genome* and *ssca2*, we achieve similar or slightly higher abort ratios. This is because of the additional memory accesses added in the translated host code. For instance, QEMU uses a set of host memory locations to emulate the guest's register file. It maintains their consistency at the boundaries of basic blocks. This can potentially result in additional access conflicts and transaction capacity overflows. Thus, developing more optimizations with a smaller memory footprint and fewer memory accesses to bring down the abort ratios is part of our future work.

Interestingly, Figure 9 shows for some benchmarks, e.g., *kmeans-low/high* and *vacation-low/high*, the transaction abort ratios of QEMU with our mechanism are lower than the native execution. A possible explanation is that the emulation in QEMU can change the original thread interleaving behavior, and thus avoids part of the concurrent execution that creates transaction conflicts. Figure 9 also shows that, with our mechanism, QEMU shares a similar distribution of abort reasons as the native execution. Moreover, the number of transaction aborts caused by *binary translation* is negligible.

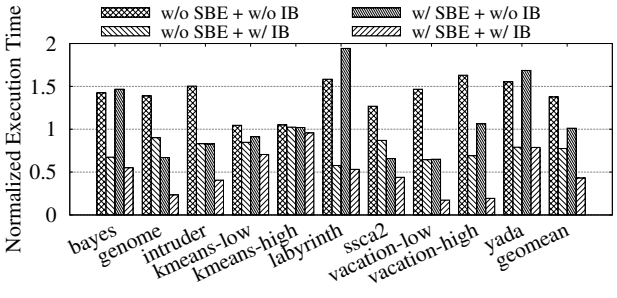

Figure 11. Performance impact of software-based emulation (SBE) and indirect branch optimizations (IB).

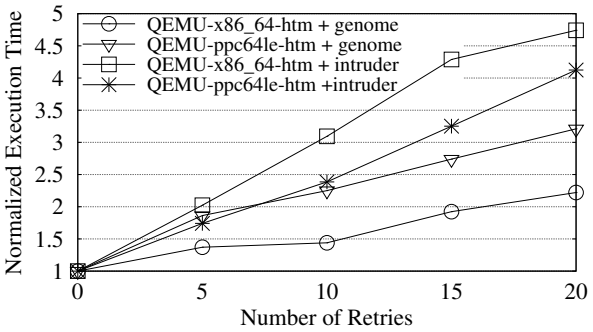

Figure 12. Performance impact of retries.

Figure 10 shows the normalized number of dynamically executed host HTs, including successful and aborted, with the corresponding number in native execution as the baseline. For most benchmarks, a similar number of host HTs are executed in QEMU with our mechanism compared to native execution. But for some benchmarks such as *kmeans-low/high*, fewer host HTs are executed by QEMU with our HTM mechanism. One possible reason is the higher transaction success ratios (as shown in Figure 9) can reduce the number of HT retries by the guest application after an abort.

Table 2 shows the percentages of dynamically emulated guest transactions in software-based emulation. As expected, for most benchmarks less than 1% of guest transactions are emulated using the software-based emulation. In another word, most of the guest HTs are emulated by host HTs after the translation. In addition, the abort ratio of software-based emulation for most benchmarks is less than 12%.

We also look into the impact of the software-based emulation and the indirect branch optimizations on the performance of the proposed HTM virtualization mechanism using QEMU-x86_64-htm. In this study, we individually disable the software-based emulation and the indirect branch optimizations in QEMU-x86_64-htm and observe the changes of the performance. Figure 11 shows the experimental results. Here, the execution time of the original QEMU, i.e., QEMU-x86_64, is used as the baseline. As shown in the figure, without the software-based emulation and the indirect branch optimizations, the performance degrades substantially due to the frequent transaction aborts. In fact, almost all of the host HTs are aborted due to the interference of the translator to the host HTs, as explained before. In contrast, the performance can be improved with the help of the software-based emulation and the indirect branch optimizations. But, they must work together to achieve a better performance.

Finally, we study the impact of retry strategies on the proposed HTM virtualization mechanism. In this study, we vary the number of retries when a host HT is aborted not due to the translator before reporting it to the guest application (see §4.4), and observe the performance changes. Two benchmarks are employed to conduct the study: *genome* and *intruder*. Figure 12 shows the results. The baseline is the execution time without a retry if not caused by the translator. As shown in the figure, the performance decreases substantially as the number of retries increases. In particular, 4.7X slowdown can be observed for QEMU-x86_64-htm when emulating *intruder* with 20 retries for each aborted host HT not caused by the translator. This demonstrates the effectiveness and efficiency of our simple retry strategy.

6.2 Scalability

Figure 13 shows the scalability results of different evaluation configurations as the number of threads increases. Note that all threads are running on different physical cores. Here, each evaluation configuration uses its corresponding single-threaded performance as the baseline. For example, the speedup of QEMU-x86_64 is calculated using the performance of QEMU-x86_64 running the single-threaded guest application as the baseline. Hence, all lines in the figure start from (1, 1). As shown in the figure, the original QEMU, i.e., QEMU-x86_64, suffers extremely poor scalability. As discussed before, the main reason is that QEMU has to emulate the fallback paths defined by the guest application.

With the proposed HTM virtualization mechanism, many benchmarks can gain a significantly improved emulation scalability, e.g., *genome*, *ssca2*, and *vacation-low/high*, which can be very close to the native execution. Some benchmarks, e.g., *labyrinth* and *yada*, are hard to scale up even in their original guest applications [48, 74]. Overall, the proposed mechanism can effectively improve the emulation scalability of QEMU for *scalable* guest HTM applications.

7 Related Work

Due to the large performance overhead needed to monitor memory accesses in non-transactional code, most of STMs only preserve weak atomicity [20–23, 26, 28, 62]. They typically require programmers to avoid accessing shared data used in transactions from non-transactional code regions. Some other STMs leverage static analyses to detect conflict accesses in non-transactional code and protect them using

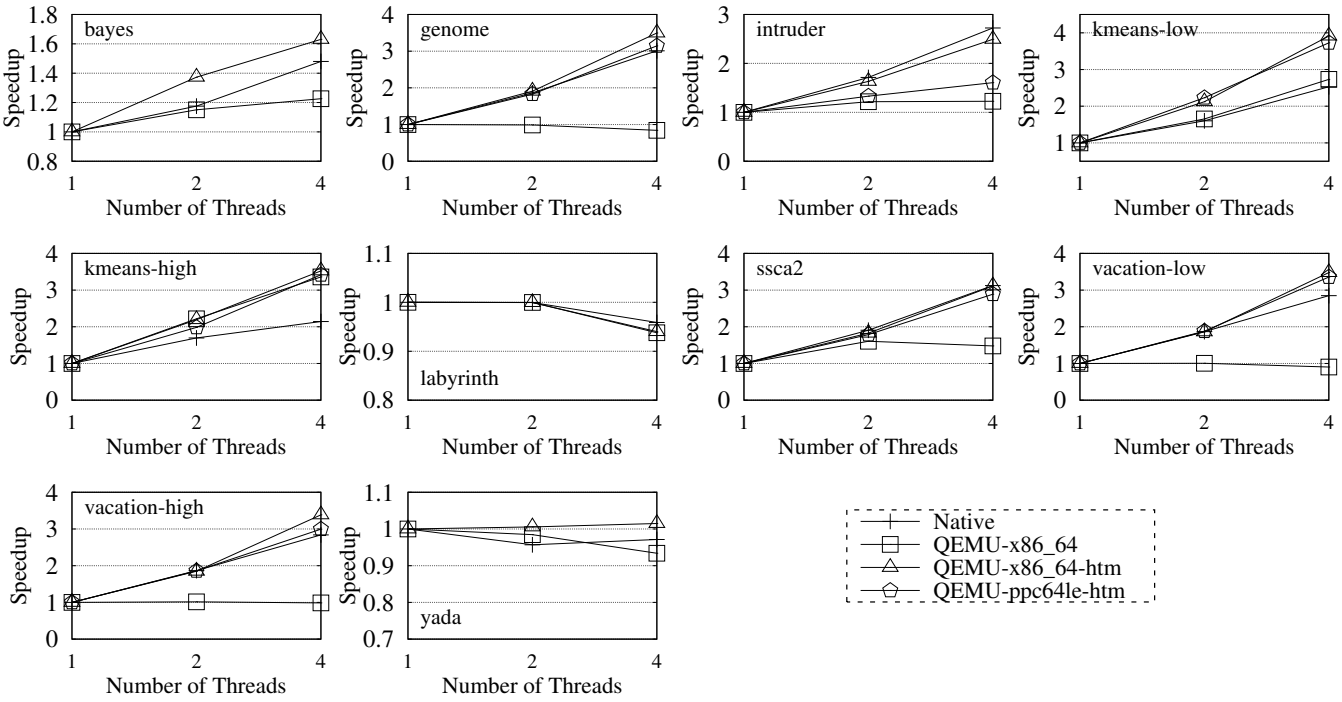

Figure 13. Scalability of individual benchmarks with different system configurations.

critical sections [59, 75] or synchronize them with transactional accesses using barriers [12, 54, 56]. However, most of them rely on a *not-accessed-in-transaction* (NAIT) analysis, which makes it hard to apply for binaries because the NAIT analysis relies on source code analyses to determine what memory locations are *never* accessed inside transactions, allowing the schemes to skip protection for these locations.

A hardware mechanism was proposed to assist STMs to achieve strong atomicity [10]. It provides access protection on each cache line, and thus can detect conflicts at a finer granularity. Similar approaches at the page level were also used in previous work to preserve strong atomicity semantics for transactions [3] and lock-based critical sections [52]. But, static program analysis is still required to reduce the overhead introduced by potential conflicts in non-transactional code. In contrast, the software-based emulation in our mechanism is used mainly to perform the binary translation, and most of the guest HTs are actually emulated by the host HTs.

Pico [15] proposes a design for a scalable DBT system that enables efficient emulation of multi-core guests on multi-core hosts. It even leverages the host HTM to efficiently emulate guest atomic instructions. However, it does not address the issue of how to emulate guest HTM in such a DBT system, which is the main focus of this paper.

In addition, there are several schemes proposed to optimize DBT systems [43, 58, 65–67, 77]. Basically, our HTM virtualization mechanism can cooperate with these optimizations to further improve the emulation performance.

8 Conclusion

Due to the limited cross-ISA support for emerging architecture-specific hardware extensions, existing virtualization techniques suffer from poor performance and scalability when emulating guest applications that take advantage of these hardware features. This paper addresses this limitation with the focus on HTM. In particular, we propose an efficient and scalable cross-ISA virtualization mechanism for HTM, which leverages host HTMs to emulate guest HTMs by translating guest HTs into host HTs. A software-based emulation scheme is employed to reduce the abort-causing interferences from the translator to host HTs. A prototype based on such a mechanism has been implemented on QEMU. Experimental results on benchmarks from STAMP demonstrate that this mechanism can achieve 2.3X and 12.6X speedup for x86_64 and PowerPC64 guest binaries, respectively, on an x86_64 host machine. Moreover, it can attain similar scalability to the original native execution on the host machine.

Acknowledgments

We are very grateful to the anonymous reviewers for their valuable comments and feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1514444. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2018. Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors. http://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances.

[2] 2019. Dolphin GameCube/Wii Emulator. http://dolphin-emu.org.

[3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional Memory with Strong Atomicity Using Off-the-shelf Memory Protection Hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 185–196. https://doi.org/10.1145/1504176.1504203

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-End Performance Isolation through Virtual Datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 233–248.

[5] ARM. 2019. NEON Technology. http://developer.arm.com/technologies/neon.

[6] ARM. 2019. Security On ARM - TrustZone. http://www.arm.com/products/security-on-arm/trustzone.

[7] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. 2003. IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 191–201. http://dl.acm.org/citation.cfm?id=956417.956550

[8] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[9] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[10] Lee Baugh, Naveen Neelakantam, and Craig Zilles. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 115–126. https://doi.org/10.1109/ISCA.2008.34

[11] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC '05)*. USENIX Association, Berkeley, CA, USA, 41–46. http://dl.acm.org/citation.cfm?id=1247360.1247401

[12] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. 2009. Feedback-directed Barrier Optimization in a Strongly Isolated STM. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 213–225. https://doi.org/10.1145/1480881.1480909

[13] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[14] Jiunn-Yeu Chen, Bor-Yeh Shen, Quan-Huei Ou, Wuu Yang, and Wei-Chung Hsu. 2013. Effective Code Discovery for ARM/Thumb Mixed ISA Binaries in a Static Binary Translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '13)*. IEEE Press, Piscataway, NJ, USA, Article 19, 10 pages. http://dl.acm.org/citation.cfm?id=2555729.2555748

[15] 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 (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 210–220. http://dl.acm.org/citation.cfm?id=3049832.3049855

[16] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*. ACM, New York, NY, USA, 555–566. https://doi.org/10.1145/2714576.2714635

[17] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code MorphingTM 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 (CGO '03)*. IEEE Computer Society, USA, 15–24.

[18] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*. IEEE Computer Society, Washington, DC, USA, 269–280. https://doi.org/10.1109/DSN.2006.31

[19] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/1508244.1508263

[20] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/1542476.1542494

[21] Aleksandar Dragojević and Tim Harris. 2012. STM in the Small: Trading Generality for Performance in Software Transactional Memory. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2168836.2168838

[22] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 237–246. https://doi.org/10.1145/1345206.1345241

[23] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/1941553.1941579

[24] Google. 2019. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator.

[25] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 3–19. https://doi.org/10.1109/SP.2015.8

[26] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. 2006. Optimizing Memory Transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 14–25. https://doi.org/10.1145/1133981.1133984

[27] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing Binary Translation of Dynamically Generated Code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 68–78. http://dl.acm.org/citation.cfm?id=2738600.2738610

[28] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/2901318.2901348

[29] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, and Bruce R. Childers. 2006. Evaluating Fragment Construction Policies for SDT Systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 122–132. https://doi.org/10.1145/1134760.1134778

[30] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, and Bruce R. Childers. 2006. Evaluating Fragment Construction Policies for SDT Systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 122–132. https://doi.org/10.1145/1134760.1134778

[31] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. 2007. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 61–73. https://doi.org/10.1109/CGO.2007.10

[32] Raymond J. Hookway and Mark A. Herdeg. 1997. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Tech. J.* 9, 1 (Jan. 1997), 3–12. http://dl.acm.org/citation.cfm?id=268940.268941

[33] Intel. 2019. Intel Advanced Vector Extensions 512 (Intel AVX-512). http://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-animation.html.

[34] Intel. 2019. Intel Software Guard Extensions (Intel® SGX). http://software.intel.com/en-us/sgx.

[35] Intel. 2019. Intel Streaming SIMD Extensions Technology. http://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.

[36] Intel. June, 2017. Programming with Intel Transactional Synchronization Extensions. In *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Chapter 16*.

[37] Daehoon Kim, Hwanju Kim, Nam Sung Kim, and Jaehyuk Huh. 2015. vCache: Architectural Support for Transparent and Isolated Virtual LLCs in Virtualized Environments. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 623–634. https://doi.org/10.1145/2830772.2830825

[38] Ho-Seop Kim and James E. Smith. 2003. Dynamic Binary Translation for Accumulator-oriented Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 25–35. http://dl.acm.org/citation.cfm?id=776261.776264

[39] Ho-Seop Kim and James E. Smith. 2003. Hardware Support for Control Transfers in Code Caches. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 253–. http://dl.acm.org/citation.cfm?id=956417.956565

[40] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251375.1251393

[41] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. 2015. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development* 59, 1 (Jan 2015), 8:1–8:14. https://doi.org/10.1147/JRD.2014.2380199

[42] Tongping Liu and Emery D. Berger. 2011. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048066.2048070

[43] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2017. Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*. IEEE Computer Society, Washington, DC, USA, 343–355. https://doi.org/10.1109/PACT.2017.15

[44] 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 (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[45] Milo Martin, Colin Blundell, and E. Lewis. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.* 5, 2 (July 2006), 17–20. https://doi.org/10.1109/L-CA.2006.18

[46] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, USA, 459–473.

[47] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. 35–46. https://doi.org/10.1109/IISWC.2008.4636089

[48] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/2749469.2750403

[49] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[50] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1317–1328. https://doi.org/10.1145/2660267.2660281

[51] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 2014. Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/2555243.2555247

[52] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. 2009. ISOLATOR: Dynamically Ensuring Isolation in Comcurrent Programs. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/1508244.1508266

[53] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. 2014. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *Proceedings of the 2014 International Symposium on Memory Management (ISMM '14)*. ACM, New York, NY, USA, 105–115. https://doi.org/10.1145/2602988.2602992

[54] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. 2008. Dynamic Optimization for Efficient Strong Atomicity. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/1449764.1449779

[55] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2017. Legato: End-to-end Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 1–13. http://dl.acm.org/citation.cfm?id=3049832.3049834

[56] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. 2007. Enforcing Isolation and Ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 78–88. https://doi.org/10.1145/1250734.1250744

[57] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[58] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. 2019. Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 77–89.

[59] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. 2010. Using Data Structure Knowledge for Efficient Lock Generation and Strong Atomicity. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 281–292. https://doi.org/10.1145/1693453.1693490

[60] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. 2016. HIPStR: Heterogeneous-ISA Program State Relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 727–741. https://doi.org/10.1145/2872362.2872408

[61] M. M. Waliullah and Per Stenstrom. 2014. Removal of Conflicts in Hardware Transactional Memory Systems. *Int. J. Parallel Program.* 42, 1 (Feb. 2014), 198–218. https://doi.org/10.1007/s10766-012-0210-0

[62] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. 2013. FastLane: Improving Performance of Software Transactional Memory for Low Thread Counts. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 113–122. https://doi.org/10.1145/2442516.2442528

[63] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC'07)*. Springer-Verlag, Berlin, Heidelberg, 4–15. http://dl.acm.org/citation.cfm?id=2392163.2392166

[64] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 84–97. https://doi.org/10.1145/3173162.3177160

[65] Wenwen Wang, Chenggang Wu, Tongxin Bai, Zhenjiang Wang, Xiang Yuan, and Huimin Cui. 2014. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development* 51, 10 (2014), 2336–2347. http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018

[66] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. 2018. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/3186411.3186413

[67] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, USA, 591–603.

[68] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM, New York, NY, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[69] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. 2015. SecPod: A Framework for Virtualization-Based Security Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, USA, 347–360.

[70] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling Concurrent Search Trees Under Contention Using HTM. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 385–399. https://doi.org/10.1145/3018743.3018752

[71] 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 (PPoPP '11)*. ACM, New York, NY, USA, 213–222. https://doi.org/10.1145/1941553.1941583

[72] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using Restricted Transactional Memory to Build a Scalable In-memory Database. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 26, 15 pages. https://doi.org/10.1145/2592798.2592815

[73] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article Article 19, 15 pages. https://doi.org/10.1145/3300061.3300122

[74] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/2503210.2503232

[75] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. 2015. Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2688500.2688510

[76] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 159–173. https://doi.org/10.1145/2872362.2872384

[77] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. 2015. HERMES: A Fast cross-ISA Binary Translator with Post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 246–256. http://dl.acm.org/citation.cfm?id=2738600.2738631