Efficient and Retargetable Dynamic Binary Translation on Multicores

Ding-Yong Hong, Jan-Jan Wu, Pen-Chung Yew, Wei-Chung Hsu, Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung

Abstract—*Dynamic binary translation* (DBT) is a core technology to many important applications such as system virtualization, dynamic binary instrumentation and security. However, there are several factors that often impede its performance: (1) emulation overhead before translation; (2) translation and optimization overhead, and (3) translated code quality. The issues also include its *retargetability* that supports guest applications from different instruction-set architectures (ISAs) to host machines also with different ISAs – an important feature to system virtualization. In this work, we take advantage of the ubiquitous multicore platforms, and use a *multithreaded* approach to implement DBT. By running the translator and the dynamic binary optimizer on different cores with different threads, it could off-load the overhead incurred by DBT on the target applications; thus, afford DBT of more sophisticated optimization techniques as well as its retargetability. Using QEMU (a popular retargetable DBT for system virtualization) and LLVM (Low-Level Virtual Machine) as our building blocks, we demonstrated in a multi-threaded DBT prototype, called *HQEMU* (*Hybrid-QEMU*), that it could improve QEMU performance by a factor of 2.6X and 4.1X on the SPEC CPU2006 integer and floating point benchmarks, respectively, for dynamic translation of x86 code to run on x86-64 platforms. For ARM codes to x86-64 platforms, HQEMU can gain a factor of 2.5X speedup over QEMU for the SPEC CPU2006 integer benchmarks. We also address the *performance scalability* issue of multi-threaded applications across ISAs. We identify two major impediments to performance scalability in QEMU: (1) coarse-grained locks used to protect shared data structures, and (2) inefficient emulation of atomic instructions across ISA's. We proposed two techniques to mitigate those problems: (1) using Indirect Branch Translation Caching (IBTC) to avoid frequent accesses to locks, and (2) using lightweight memory transactions to emulate atomic instructions across ISAs. Our experimental results show that, for multi-thread applications, HQEMU achieves 25X speedups over QEMU for the PARSEC benchmarks.

Index Terms—Dynamic Binary Translation, Multicores, Feedback-Directed Optimization, Hardware Performance Monitoring, Traces

1 Introduction

Dynamic binary translators (DBT) that emulate a *guest* binary executable code in one instruction-set architecture (ISA) on a *host* machine with a *different* ISA are gaining importance. It is because dynamic binary translation is a core technology of *system virtualization*. DBT is also frequently used in binary instrumentation, security monitoring and other important applications. However, there are several factors that could impede the effectiveness of a DBT: (1) emulation overhead before the translation; (2) translation and optimization overhead; (3) the quality of the translated code. *Retargetablity* of the DBT is also an important requirement. We would like to have a *single* DBT to take on application binaries from *several different* ISAs and retarget them to host machines with *different* ISAs. This requirement imposes additional constraints on the structure of a DBT and, thus, additional overheads.

As a DBT is running at the same time the application is being executed, the overall performance is very sensitive to the overhead of the DBT itself. A DBT could ill-afford to have sophisticated techniques and optimizations for better codes. However, with the ubiquity of the multicore processors today, most of the DBT overheads could be off-loaded to other cores when they are not in use. The DBT could thus leverage multithreading on multicores itself. This allows DBT to become more scalable when it needs to take on large-scale multithreaded applications.

In this work, we developed a *multithreaded* DBT prototype, called HQEMU (*Hybrid-QEMU*), which uses QEMU [1], a *retargetable* DBT system as its frontend for fast binary code *emulation* and *translation*. However, QEMU lacks a sophisticated optimization backend to generate more efficient code. To this, we use the LLVM compiler [2], also a popular compiler with sophisticated compiler optimization as its backend, together with a *dynamic binary optimizer* that uses on-chip hardware performance monitor (HPM) to dynamically improve code for higher performance. With the *hybrid* QEMU + LLVM approach, we successfully address the dual issues of high-quality translated code and low translation overhead. Significant performance improvement over QEMU has been observed. To our knowledge, our work is the first successful effort to integrate QEMU and LLVM to achieve significant improvement.

We also addressed the performance scalability issue in translating multi-threaded applications across ISAs. It requires reducing the amount of shared resources and more efficient synchronization mechanisms to handle the large number of application threads that need to be

- *D.-Y. Hong and Y.-C. Chung are with the Department of Computer Science, National Tsing Hua University, Taiwan. E-mail: dyhong@sslab.cs.nthu.edu.tw and ychung@cs.nthu.edu.tw*
- *P.-C. Yew is with the Department of Computer Science, University of Minnesota, USA. E-mail: yew@cs.umn.edu.*
- *C.-C. Hsu, P. Liu and W.-C. Hsu are with the Department of Computer Science and Information Engineering, National Taiwan University, Taiwan. E-mail: {d950006,pangfeng,hsuwc}@csie.ntu.edu.tw.*
- *J.-J. Wu and C.-M. Wang are with the Institute of Information Science, Academia Sinica, Taiwan. E-mail: {wuj,cmwang}@iis.sinica.edu.tw.*

translated and optimized.

The main contributions of this work are as follows:

- We developed a *multi-threaded retargetable DBT on muticores* that achieved *low translation overhead* and *good translated code quality* on the guest binary applications. We show that this approach can be beneficial to both *short-* and *long-running* applications.
- We propose a novel trace combining technique to improve existing trace formation algorithms. It could effectively combine/merge traces based on the information provided by the on-chip HPM. We demonstrate that such feedback-directed trace merging optimization can significantly improve the overall code performance.
- We use two optimization schemes, *indirect branch translation caching* (*IBTC*) and *lightweight memory transactions*, to reduce the contention on shared resources when emulating a large number of application threads. We show that these optimizations significantly reduce the emulation overhead of a DBT and make it more scalable.
- We built a HQEMU prototype, and the experimental results show it could improve the performance by a factor of 2.6X and 4.1X over QEMU for *x86 to x86-64* emulation using SPEC CPU2006 integer and floating point benchmarks, respectively. For *ARM to x86-64* emulation, HQEMU shows a gain of 2.5X speedup over QEMU for SPEC integer benchmarks. For the performance of multithreaded applications, HQEMU achieves 25X speedup over QEMU for the PARSEC benchmarks with 32 emulated threads.

This paper extends our previous work [3], which focuses on the techniques to enhance single-thread performance, with techniques to enhance scalability of emulating multi-threaded programs.

The rest of this paper is organized as follows. Section 2 provides a brief overview of our multi-threaded hybrid QEMU+LLVM DBT system. We then elaborate on three unique aspects of HQEMU: (1) Techniques to improve single-thread performance that include *trace formation* and *trace merging* in Section 3; (2) Techniques to enhance *scalability* that address the contention of shared resources among multiple threads, *IBTC*, and handling of atomic operations for synchronization using *light-weight memory transactions* in Section 4; and (3) Issues related to *retargetability* of DBT in Section 5. We detail some experimental results on the effectiveness of HQEMU in Section 6. Section 7 presents some related work. Finally, Section 8 concludes this paper.

2 A Trace-Based Hybrid Dynamic Binary Translator

QEMU is a state-of-the-art *retargetable* DBT system that enables both *full-system virtualization* and *process-level emulation*. It has been widely used in many applications. This motivates us to use QEMU as our base.

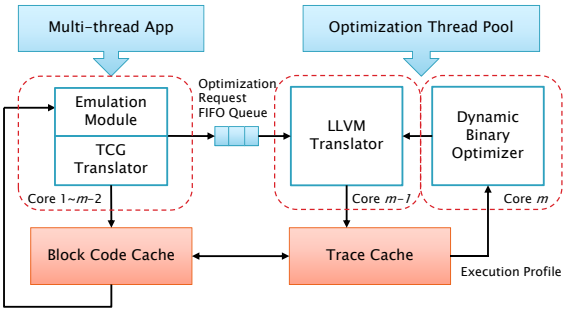


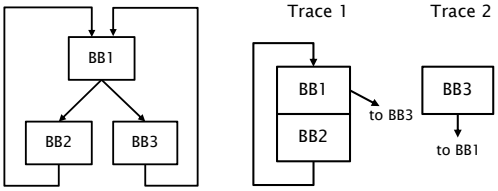
Fig. 1: Major components of HQEMU on an m-core platform.

The core translation engine of QEMU is called *Tiny Code Generator* (*TCG*), which provides a small set of IR (intermediate representation) operations. The main loop of QEMU translates and executes the emulated code one *basic block* at a time. TCG provides two simple optimization passes: *register liveness analysis* and *store forwarding optimization*. *Dead code elimination* is done as a by-product of these two optimizations. Finally, the intermediate code is translated into the host binary. The whole translation and optimization process is designed to be lightweight and with negligible overhead. Such design considerations make QEMU an ideal platform for emulating *short-running* applications or applications with *few* hot blocks, such as during the booting of an operating system.

Figure 1 shows the organization of HQEMU. It has an enhanced QEMU as its frontend, and an LLVM together with a *dynamic binary optimizer* (*DBO*) as its backend. QEMU is running by the execution thread(s), LLVM and DBO are running on separate threads depending on their workloads. Two code caches, a *block-code cache* and a *trace cache*, are used in HQEMU. They keep translated binary codes at different optimization levels.

There are two translators in HQEMU for different purposes. The translator in the enhanced QEMU (i.e. TCG) acts as a *fast* translator. TCG translates guest binary at the granularity of a *basic block*, and emits translated codes to the *block-code cache*. It also keeps the translated guest binary in its TCG IR format for further optimization in the HQEMU backend. The *emulation module* (i.e. the *dispatcher* in QEMU) coordinates the *translation* and the *execution* of guest binaries. When the emulation module detects that some code region has become *hot* and is worthy of further optimization, it sends a request to the *optimization request FIFO queue* with the translated guest binary in its TCG IR format. The requests will be serviced by the HQEMU backend optimizer running on another thread. We use an enhanced LLVM compiler as the backend because it consists of a rich set of aggressive optimizations and a just-in-time runtime system.

When the LLVM optimizer receives an optimization request from the FIFO queue, it converts its TCG IR to LLVM IR directly instead of converting guest binary from its original ISA. This approach simplifies the backend translator tremendously (see Section 5 for more details). A rich set of program analysis facilities



(a) CFG (b) Trace

Fig. 2: A CFG of three basic blocks and traces generated with NET trace selection algorithm.

and powerful optimization passes in LLVM can produce very high quality host codes, and they are stored in the *trace cache*. Such analysis and optimization are running concurrently on another thread. Hence, their overhead can be hidden and without interfering with the execution of the guest program. The backend LLVM translator can also spawn more worker threads to accelerate the processing of optimization requests if there are many waiting in the queue. We also apply the structure of a non-blocking FIFO queue [4] to reduce the overhead of communication among these threads.

The DBO uses a *hardware performance monitor* based (i.e. HPM-based), feedback-directed runtime optimization scheme. It can detect separate traces with low overhead and work with the LLVM translator to re-optimize the merged traces (see Section 3 for more details).

With the hybrid QEMU+LLVM approach, we can benefit from the strength of both translators. This approach successfully addresses the dual issues of good translated code quality and low translation overhead.

3 Techniques to Enhance Single-Thread Performance

A typical binary translator needs to save and restore program contexts when the control switches between the *dispatcher* and the execution of translated code in the *code caches*, and also among small code regions in code caches. Such small code region transitions could incur significant overhead. Enlarging the code regions can alleviate such overheads. The idea is to merge many small code regions into larger ones, called *traces*, and thus eliminating the redundant load and store operations by promoting such memory operations to register accesses within traces. Through such *trace formation*, we not only can eliminate the high overhead of region transitions, but also can apply more code optimizations to a larger code region.

A relaxed version of *Next Executing Tail* (NET) [5] is chosen as our trace selection algorithm. In the original NET scheme, it considers every backward branch as an indicator of a cyclic execution path, and terminates the trace formation at such backward branches. We relax such a backward-branch constraint, and stop trace formation only when the same program counter (PC) is executed again. More details on *trace formation*, *hot trace detection*, and *optimization techniques* in HQEMU can be found in Appendix A.

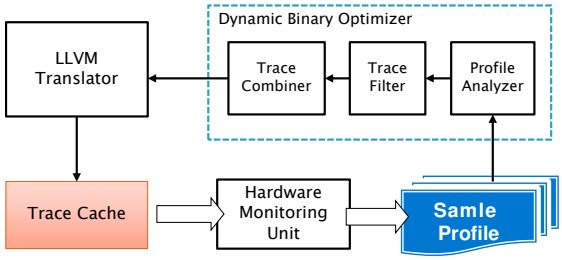


Fig. 3: Workflow of the HPM-based trace merging in DBO.

Although using such NET-based algorithm can generate high-quality traces with low cost, such trace formation techniques have some well-known weaknesses such as *trace separation* and *early exits* [6]. The main cause of the weaknesses is that NET-based algorithm can only handle code regions with *simple control flow graph* (CFG), such as straight fall-through paths or simple loops. It cannot deal with code regions with more complex control flow patterns. Figure 2(a) shows a code example with three basic blocks. Applying NET on this code region will result in two separate traces as shown in Figure 2(b). Trace 1 will have a frequent *early exit* to Trace 2 that could incur significant transition overhead. In order to overcome such problems, our improved trace-merging algorithm will force the merging of problematic traces that frequently jump between themselves.

The trace merging is different from conventional *trace chaining* [7] where two traces are chained together by patching a jump from the *side exit* of one trace to the *head* of the other. Conducting *trace chaining* does not solve the problem of *early exits* due to *trace separation*, i.e. control transfer occurs in the middle of a trace, instead of from the tail, to another trace. In contrast, *trace merging* can keep the execution staying in the combined trace.

The challenges for trace merging are (1) efficient detection of such problematic traces, and (2) implementation of such merging at runtime. One feasible approach is to insert detection routines to detect the separation of traces and early exits at each jump instruction in each trace. This approach, however, will incur substantial overhead because they are likely to be in frequently executed hot code regions. Instead, we use a feedback-directed approach with the help of on-chip hardware performance monitor to support trace merging. The workflow of such trace merging in DBO is shown in Figure 3.

The DBO consists of three components: *a profile analyzer*, *a trace filter* and *a trace combiner*. At first, the *profile analyzer* collects sampled PCs and accumulates the sample counts for each trace to determine its *hotness*. In the second step, the *trace filter* selects hot candidate traces for merging. In our algorithm, a trace has to meet three criteria to be considered as a *hot* trace: (1) the trace is in a *stable state*; (2) the trace is in the *90% cover set* (to be explained later), and (3) the sampled PC count of the trace must be greater than a threshold.

To determine if a trace has entered a *stable state*, a *circular queue* is maintained in the trace filter to keep track of the traces executed in the most recent N sampled

intervals. The collection of traces executed in the most recently sampled interval is put in an entry of the circular queue, and the oldest entry at the tail of the queue is discarded if the queue overflows. We consider a trace is in a *stable state* if it appears in *all* entries of the circular queue. The top traces that contribute to 90% of total sample counts are collected as the *90% cover set*.

The *trace combiner* then chooses the traces that are likely to cause trace separation for merging. Note that, in trace formation, we apply the concept in NET to collect the basic blocks that form a cyclic path to build a trace. The same concept is applied here in trace merging. The *trace combiner* collects all traces that form cyclic paths after merging. However, we do not limit the shape of the merged trace to a simple loop. Any CFG that has nested loops, irreducible loops, or several loops in a trace, can be formed as a merged trace. Moreover, it is possible to have several groups of traces being merged at a time.

Finally, the groups of traces merged by the *trace combiner* are placed in the *optimization request FIFO queue* for further optimization by LLVM, and the LLVM translator re-builds the LLVM IR of the merged traces from their component blocks' TCG IR. After a merged trace is optimized, its initial sample count is set to the maximum sample count of its component traces. Moreover, the sample counts of the component traces are reset to zero so that they will not affect the formation of the next 90% cover set for future trace combination.

4 Techniques to Enhance Scalability of Emulating Multi-Threaded Programs

QEMU has two modes in emulating an application binary: (1) *full-system emulation*, in which all OS kernels involved are also emulated, and (2) *process-level emulation*, in which only application binaries are emulated. In this work, we focus on *process-level emulation*. When emulating a multithreaded application, QEMU creates one host thread for each guest thread, and all these guest threads are emulated concurrently.

QEMU uses a *globally shared* code cache, i.e. all executing threads share a *single* code cache, and each guest block has only *one* translated copy in the shared code cache. All threads maintain a single directory that records the mapping from a guest code block to its translated host code region. An execution thread first looks up the directory to locate the translated code region. If not found, it kick-starts the TCG to translate the untranslated guest code block. Since all execution threads share the code cache and the directory, QEMU uses a *critical section* to *serialize* all accesses to the shared structures. Such a design yields very efficient memory space usage, but it could cause severe contention to the shared code cache and directory when a large number of guest threads are emulated.

In this section, we identify two problems in QEMU when emulating multithread programs, and then describe the optimization strategies used in HQEMU to mitigate those problems.

4.1 Indirect Branch Handling

Indirect branches, such as *indirect jump*, *indirect call* and *return* instructions, cannot be linked in the same way as direct branches because they can have *multiple jump targets*. Making the execution threads go back to the *dispatcher* for the branch target translation each time when an indirect branch is encountered may cause huge performance degradation. The degradation is due to the overhead from (1) saving and restoring program contexts when a context switch occurs, and (2) the contention for the shared directory (protected in a critical section) to find the branch target address when a large number of threads are emulated.

To mitigate this overhead, we try to avoid going back to the dispatcher for branch target translation. For the indirect branches that leave a block or exit a trace, the *Indirect Branch Translation Cache* (IBTC) [8] is used. The translation of an indirect branch target with IBTC is performed as a fast hash table look-up inside the code cache. Only upon an IBTC miss, the execution thread goes back to the dispatcher, performs expensive translation of indirect branch with the shared directory, and caches the lookup result in the IBTC entry. Upon an IBTC hit, the execution jumps directly to the next translated code region so that a context switch back to the dispatcher is not required. The IBTC in our framework is a large hash table shared by all indirect branches, including indirect jump/call and return instructions. That is, the translation of branch targets looks up the same IBTC for all indirect branches. We set up one IBTC for *each* execution thread. Such *thread-private* IBTC can avoid contention during the branch target translation. The detailed implementation of IBTC hash table and a comparison with Pin's indirect branch chain are described in Appendix B.

During *trace formation*, the prediction routine might record two successive blocks following the path of an indirect branch. We use *IB inlining* (*Indirect Branch Inlining*) [9] to facilitate the translation of indirect branch target. In *IB inlining*, when translating an indirect branch in the predecessor block, the code to compare the value in the indexing register against the address of the successor block is inlined in the trace. Upon a match, the execution will continue to the successor block without leaving the trace. If there is a no-match, meaning that the prediction fails, this indirect branch will leave the trace and the execution is redirected to the IBTC. Such *IB inlining* is advantageous because it must be hot to be included in a trace, and thus the prediction is most likely to succeed. Using *thread-private* IBTC and *IB inlining*, we can effectively reduce the overhead by avoding thread contention and keeping the execution threads staying in the code cache most of the time.

4.2 Atomic Instruction Emulation

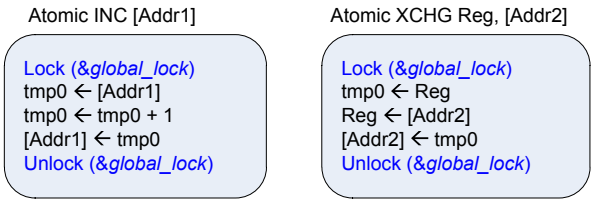
The emulation of *guest atomic instructions*, which are often used to implement synchronization primitives, poses another design challenge. The correctness and efficiency

of emulating atomic instructions are critical to multi-threaded applications. To ensure the correctness, DBT must guarantee that the translated host code be executed atomically. To emulate the *atomicity* of a *guest atomic instruction*, QEMU places the translated code region that corresponds to the *guest atomic instruction* in a *critical section*, protected with a *lock-unlock pair*, on the host machine. Thus, concurrent accesses to the critical section are serialized. However, QEMU uses the *same lock variable* for *all* such critical sections. Figure 4(a) shows how two guest atomic instructions, *atomic INC* and *atomic XCHG*, are protected by the *same* lock variable *global_lock*. The reason that QEMU uses the same global lock variable for all such critical sections is because it cannot determine if any two memory addresses are aliases at the translation time.

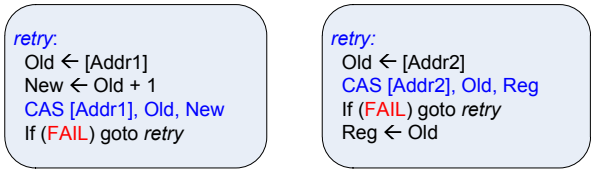
Although the global lock scheme of QEMU is portable, it has several problems: (1) Wang et al. [10] proved that this approach could still have correctness issues that may cause deadlocks; (2) accesses to non-aliased memory locations (e.g. two independent mutex variables in the guest source file) by different threads are serialized because of the same global lock; (3) the performance is poor due to the high cost of the locking mechanism. The overhead of accessing the global lock depends on the design of the locking mechanism. For example, the locking mechanism in QEMU is implemented using NPTL synchronization primitives, which use Linux *futex* (a fast user-space mutex). When an execution thread fails to acquire or release the global lock, the thread is put to sleep in a wait-queue, and is waken later via an expensive futex system call. Such expensive switching between user and kernel mode and the additional contention caused by false protection of non-aliased memory accesses could result in significant performance degradation.

To solve the problems incurred by the global lock, we use *lightweight memory transactions* proposed in [10] to address the correctness issues, as well as to achieve efficient atomic instruction emulation. The lightweight memory transaction based on the multi-word compare-and-swap (CASN) algorithm [11] allows translated code of atomic instructions to be executed optimistically. It detects data races while emulating an atomic instruction using the atomic primitives supported by the host architecture, and re-executes this instruction until the entire emulation is atomically performed. Figure 4(b) illustrates the translation of the same two guest atomic instructions using lightweight memory transactions. At first, the value of the referenced memory is loaded to the temporary register, *Old*. The new value after the computation is atomically stored in the memory if the value in the memory is the same as *Old*. Otherwise, the emulation keeps retrying if the CAS transaction fails.

Based on this approach, the protection of memory accesses with a global lock can be safely removed because the lightweight memory transactions can guarantee correct emulation of atomic instructions. Moreover, the performance will not degrade much because the false protection of non-alias memory accesses and the overhead of expensive locking mechanism are eliminated as a result of the removal of global lock.



(a) The same global lock variable *global_lock*



(b) Lightweight memory transactions

Fig. 4: An example of translating two atomic instructions using global lock and lightweight memory transactions.

5 Retargetability

The goal of HQEMU is to have a *single* DBT framework to take on application binaries from *several different ISAs* and retarget them to host machines with *different ISAs*. Using a common intermediate representation (IR) is an effective approach to achieve retargetability, which is used in both QEMU (i.e. TCG) and LLVM. By combinig these two frameworks, HQEMU inherits their retargetability with minimum effort. In HQEMU, when LLVM optimizer receives an optimization request from the FIFO queue, it converts its TCG IR to LLVM IR directly instead of converting guest binary from its original ISA. Such *two-level IR conversion* simplifies the translator tremendously because TCG IR only consists of about 142 different operation codes – much smaller than in most existing ISAs. Without such two-level IR conversion, for example, supporting full x86 ISA requires implementing more than 2000 x86 opcode to LLVM IR conversion routines.

A retargetable DBT does not maintain a fixed register mapping between the guest architectural states and the host architectural states. It thus has extra overhead compared to same-ISA DBTs (e.g. Dynamo [7]) or dedicated DBTs (e.g. IA-32 EL [12]), which usually assume the host ISA has the *same* or *richer* register set than the guest ISA. Moreover, retargetable DBTs allow flexible translation, such as adaptive SIMDization to any vector size or running 64-bit binary on 32-bit machines. This is hard to achieve by same-ISA and dedicated DBTs.

6 Performance Evaluation

In this section, we present the performance evaluation of HQEMU by using both single-threaded and multi-threaded benchmarks. To show the performance portability of HQEMU across different ISAs, we also compare

the results with QEMU. Other DBT systems, such as Pin or DynamoRIO, are not compared because they are not cross-ISA DBTs, and most of their execution is *done in native mode* with no need for translation, and hence, no performance degradation.

6.1 Emulation of Single-Thread Programs

We first evaluate the performance of HQEMU on single-threaded programs. SPEC CPU2006 benchmark suite is chosen as the test programs in this experiment.

6.1.1 Experimental Setup

All performance evaluation is conducted on three host platforms listed in Table 1. The SPEC CPU2006 benchmark suite is tested with both test and reference inputs and for two different guest ISAs, ARM and x86, to show the retargetability of HQEMU. All benchmarks are compiled with GCC 4.4.2 for the x86 guest ISA and GCC 4.4.1 for the ARM guest ISA. LLVM version 3.0 is used for the x86 and PPC host, and version 2.8 for the ARM host. The default optimization level (-O2) is used for JIT compilation. We run only one thread with the LLVM translator and this thread is capable of handling all optimization requests. The trace-profiling threshold is set at 50 and the maximum length of a trace is 16 basic blocks. We use Perfmon2 for performance monitoring with HPM. The sampling interval is set at 1 million cycles/sample. The size of the circular queue, N, for trace merging in the dynamic optimizer is set at 8. We compare the results to the native runs whose programs are compiled to the host executable with SIMD enabled. All compiler optimization flags used are listed in Table 1. Four different configurations are used to evaluate the effectiveness of HQEMU:

- **QEMU** which is the vanilla QEMU version 0.13 with the fast TCG translator.
- **LLVM** which uses the same modules of **QEMU** except that the TCG translator is replaced by the LLVM translator.
- **HQEMU-S** which is the single-threaded HQEMU with TCG and LLVM translators running on the same thread.
- **HQEMU-M** which is the multi-threaded HQEMU, with TCG and LLVM translators running on separate threads.

In both QEMU and LLVM configurations, code translation is conducted at the granularity of *basic blocks* without trace formation. In HQEMU-S and HQEMU-M configurations, *trace formation* and *trace merging* are used. IBTC is used in all configurations except QEMU.

6.1.2 Overall Performance of SPEC CPU2006

Figure 5 illustrates the overall performance of *x86-32 to x86-64*, *ARM to x86-64*, *ARM to PPC64*, and *x86-32 to ARM* emulations against the native runs with *reference inputs*. The Y-axis is the normalized execution time over native execution time. Note that in all figures, we do not provide the confidence intervals because there was no noticeable performance variation among different runs. Detailed performance results with *test inputs* are shown in Appendix C.

TABLE 1: Configurations. DEFAULT="-O2 -fno-strict-aliasing"

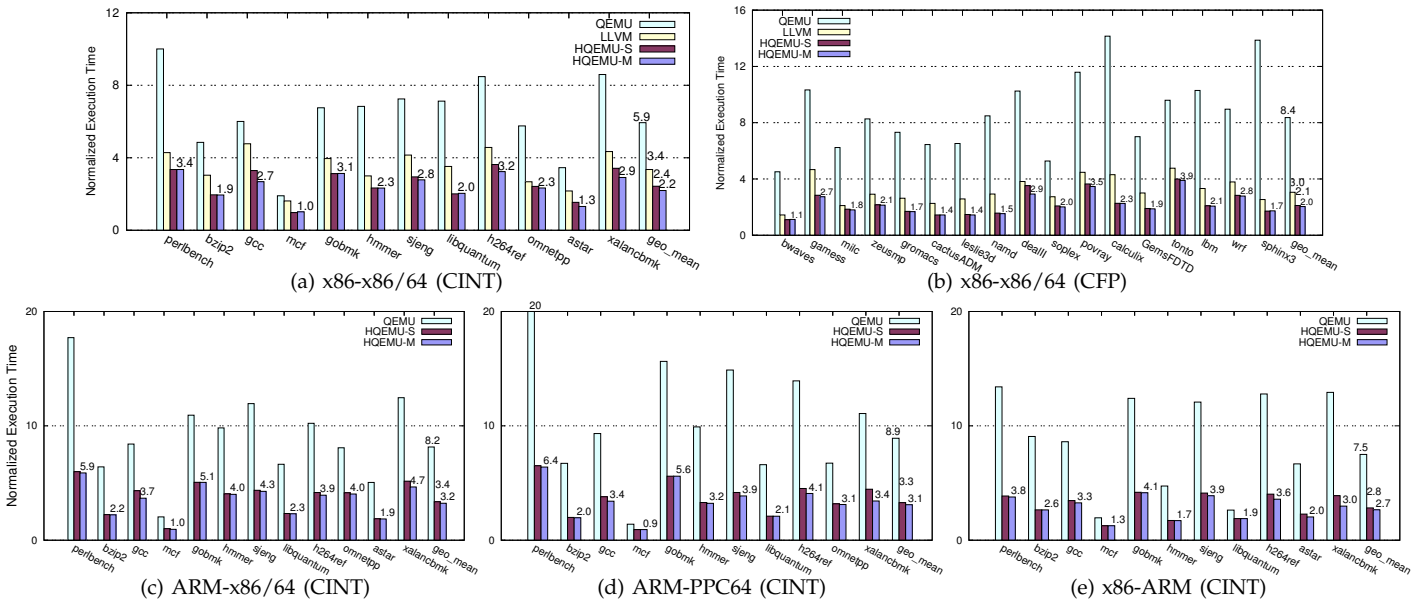
Hardware settings – CPU / Memory size	
x86/64	3.3 GHz quad-core Intel Core i7 975 / 12 GB
PPC/64	2.0 GHz dual-core PPC970FX / 4 GB
ARM	1.3 GHz quad-core ARMv7r / 2 GB
Optimization flags	
Native-x86/64	`$DEFAULT`
Native-PPC/64	`$DEFAULT -maltivec`
Native-ARM	`$DEFAULT -mfpu=vfp`
Guest-x86/32	`$DEFAULT -m32 -msse2 -mfpmath=sse`
Guest-ARM	`$DEFAULT -ffast-math -msoft-float -mfpu=neon -ftree-vectorize`

Figure 5(a) and 5(b) present the *x86-32 to x86-64* emulation results for integer and floating point benchmarks. Unlike *test inputs*, the programs spend much more propotion of time running in the code caches. As the results show, the LLVM configuration outperforms QEMU since optimization overhead is mostly amortized. The speedup from LLVM includes some DBT-related optimizations such as *indirect branch prediction*, as well as regular compiler optimizations such as *redundant load/store elimination*. Redundant load/store elimination is effective in reducing expensive memory operations. *Trace formation* and *trace merging* in HQEMU further eliminate many redundant load/store instructions related to architecture state transitions. Through trace formation, HQEMU achieves significant improvement over both QEMU and LLVM. Using *reference inputs*, the benefit of HQEMU-M is not as significant as that of using test inputs when compared to HQEMU-S. This is because the translation overhead is playing less of a role using *reference inputs*. As shown in Figure 5(a) and 5(b), HQEMU-M achieves about 45.5% and 50% of the native speed for CINT and CFP benchmarks, respectively. Compared to QEMU, HQEMU-M is 2.6X and 4.1X faster for CINT and CFP, respectively.

For CFPs, the speedup of LLVM and HQEMU over QEMU is greater than that of CINTs. This is partly due to the translation ability of the current QEMU/TCG. The current TCG translator does not emit *floating point* instructions for the host machine. Instead, all floating point instructions are emulated via helper function calls. By using the LLVM compiler infrastructure, such helper functions can be inlined and allow floating point host instructions to be generated directly in the code cache.

Figure 5(c) to 5(e) illustrates the performance results of ARM to x86-64, ARM to PPC64 and x86-32 to ARM emulation over native execution (i.e. running binary code natively). For PPC64 and ARM host, trace merging is not used[1]. The performance results are similar to those of x86-32 to x86-64 emulation – HQEMU-M is 2.5X and 2.9X faster than QEMU for CINT with ARM guest to x86-64 and PPC64 host, respectively, and are about 31.2% and 32.3% of the native speed, respectively. As for x86-32

1. We failed to enable hardware counters on these two platforms.



(a) x86-x86/64 (CINT) (b) x86-x86/64 (CFP)

(c) ARM-x86/64 (CINT) (d) ARM-PPC64 (CINT) (e) x86-ARM (CINT)

Fig. 5: SPEC CPU2006 results of x86/32-x86/64, ARM-x86/64, ARM-PPC64 and x86/32-ARM emulation with reference inputs.

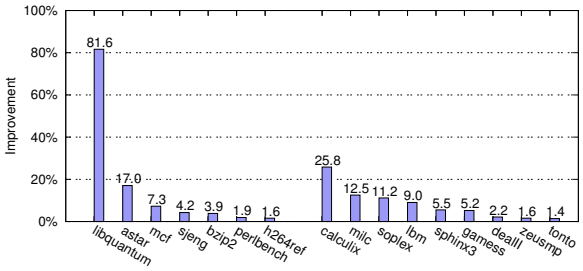


Fig. 6: Improvement of trace merging with x86 to x86-64 emulation for SPEC CPU2006 with reference inputs.

to ARM emulation, HQEMU-M achieves 2.8X speedup over QEMU for CINT with *reference inputs*, and is about 37% of the native speed. The results show the retargetability of HQEMU and that the optimizations used in HQEMU can indeed achieve performance portability.

The above results show that QEMU is suitable for emulating *short* runs or programs with very few hot blocks (Appendix C). The LLVM configuration is better for long running programs with heavy reuse of translated codes. HQEMU has successfully combined the advantages of *both* QEMU and LLVM, and can efficiently emulate both short- and long-running applications. Furthermore, the trace formation and merging in HQEMU expand the power of LLVM optimization to significantly remove redundant loads/stores. With HQEMU, cross-ISA emulation is getting closer to the performance of native runs.

6.1.3 Results of Trace Formation and Trace Merging

To evaluate the impact of *trace formation* and *trace merging*, we use x86-32 to x86-64 emulation with SPEC CPU2006 benchmarks as an example to show how much emulation overhead can be removed from reducing code region transitions. In this experiment, the total number of memory operations in each benchmark is measured for (a) LLVM, (b) HQEMU with trace formation only, and (c) HQEMU with both trace formation and merging. The difference between (a) and (b) represents the number of redundant memory accesses eliminated by *trace formation*; the difference between (b) and (c) represents the impact of *trace merging*. Hardware monitoring counters are used to track the events, MEM_INST_RETIRED:LOADS/STORES, and to collect the total number of memory operations. Table 2 lists the results of such measurements for CINTs and CFPs.

TABLE 2: Measures of traces with x86 to x86-64 emulation for SPEC CPU2006 benchmarks with reference input. (Unit of time: second. Unit of column six and seven: 10^{10} memory-ops.)

CINT2006						
Benchmark	# Trace	Trans. Time	# Mg.	Ver.	(b)-(a)	(c)-(b)
perlbench	13102	20.9 (1.7%)	6	4	126.7	7.1
bzip2	3084	5.2 (0.5%)	43	4	224.0	27.3
gcc	159769	215 (25.4%)	40	5	210.6	3.1
mcf	276	.6 (0.6%)	13	3	31.3	9.0
gobmk	43228	54.5 (3.9%)	57	4	136.8	3.4
hmmer	938	1.9 (0.2%)	0	0	136.6	.0
sjeng	1438	1.8 (0.1%)	33	4	159.6	36.4
libquantum	221	.4 (0.1%)	2	1	24.8	319.4
h264ref	6308	12.6 (0.6%)	1	1	396.4	16.4
omnetpp	1773	3.4 (0.4%)	7	3	39.9	6.4
astar	1074	1.8 (0.3%)	37	8	87.2	34.8
xalancbmk	3220	7.4 (1.0%)	0	0	136.2	.0
CFP2006						
Benchmark	# Trace	Trans. Time	# Mg.	Ver.	(b)-(a)	(c)-(b)
bwaves	364	1.3 (0.1%)	1	1	81.5	.7
gamess	10624	27.7 (1.2%)	61	5	402.9	27.1
zeusmp	1659	6.5 (0.6%)	25	6	163.5	9.5
cactusADM	977	2.2 (0.1%)	1	1	226.0	-9.3
namd	1085	3.3 (0.5%)	6	1	224.1	3.7
dealII	3911	6.3 (0.6%)	11	3	79.6	11.8
soplex	2461	6.3 (1.2%)	11	1	47.0	48.9
povray	1958	4.5 (0.6%)	1	1	76.9	-6.4
calculix	3484	6.8 (0.3%)	15	3	393.5	372.9
tonto	4997	12.0 (0.6%)	12	2	177.3	17.9
lbm	164	0.4 (0.1%)	1	1	78.3	13.6
wrf	5441	13.2 (0.7%)	20	2	349.0	7.4

Column two in Table 2 presents the total number of traces generated in each benchmark. *Column three* lists the total translation time by the LLVM compiler and its percentage over total execution time. Each trace is

associated with a version number and is initially set to zero. After trace merging, the version number of the new trace is set to the maximum version number of the traces merged plus one. The number of traces merged and the maximum version number are listed in *column four* and *five*, respectively. The reduced numbers of memory operations after trace formation (b)-(a) and trace merging (c)-(b) are listed in *column six* and *seven*, respectively. The improvement rate by trace merging is shown in Figure 6. From Table 2, we can see that most redundant memory operations can be eliminated by trace formation in almost all benchmarks. `libquantum` has the most redundant memory operations eliminated and the most significant performance improvement from trace merging (see Figure 6).

As for `libquantum`, its hottest code region is composed of three basic blocks, and its CFG is shown in Figure 2(a). The code region is split into two separate traces by the NET trace selection algorithm. During trace transitions, almost all general-purpose registers of the guest architecture need to be stored and reloaded again. In addition, there are billions of transitions between these two traces during the entire execution. Through trace merging, HQEMU successfully merges these two traces with its CFG shown in Figure 2(a). It keeps the execution staying within this region. Thus, its performance is improved by 82%. The analysis of translation overhead and the breakdown of time with *reference inputs* are presented in Appendix D.

6.2 Emulation of Multi-Thread Programs

In the following experiments, we evaluate the performance of HQEMU for multi-threaded programs. PARSEC [13] version 2.1 is used as the testing benchmarks.

6.2.1 Experimental Setup

The experiments are conducted on two systems: (1) eight six-core AMD Opteron 6172 processors (48 cores in total) with a clock rate of 2 GHz and 32 GBytes main memory; (2) the ARM platform listed in Table 1. The PARSEC benchmarks are evaluated with the `native` and `simlarge` input sets for x86-64 and ARM platform, respectively. All benchmarks are parallelized with the *Pthread* model and compiled for x86-32 guest ISA with PARSEC default compiler optimization and SIMD enabled. We compare their performance to native execution with three different configurations: (1) **QEMU**, (2) **HQEMU** (multi-thread mode), and (3) **QEMU-Opt** which is an enhanced QEMU with IBTC optimization and block chaining across page boundary. For all configurations, atomic instructions are emulated with lightweight memory transactions so that the benchmarks can be emulated correctly.

6.2.2 Overall Performance of PARSEC

Figure 7 illustrates the performance results of all PARSEC benchmarks with *native* input sets for x86 to x86-64

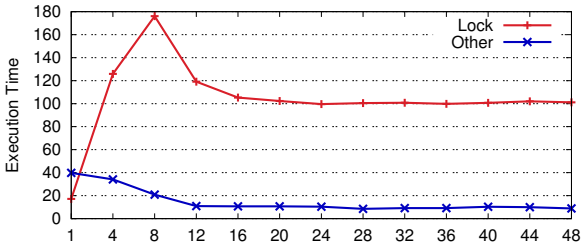


Fig. 8: Breakdown of time with `blackscholes` with simlarge input. The unit of Execution Time in Y-axis is second. X-axis shows the number of threads.

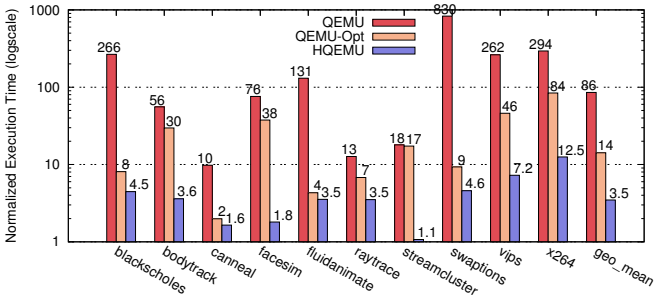


Fig. 9: PARSEC results of x86 to x86-64 emulation with 32 threads and native input sets.

emulation. The X-axis is the number of worker threads created via the command line argument. The Y-axis is the elapsed time measured in seconds with the `time` command. As shown in Figure 7(a) to 7(j), the performance of QEMU does not scale well. In all sub-figures, the execution time increases dramatically when the number of guest threads increases from 1 to 8, then decreases with more threads. It remains mostly unchanged as the number of threads is above 16. The poor scalability of QEMU is mostly due to the sequential translation of branch targets within the QEMU dispatcher because the mapping directory is protected in a critical section. Since IBTC optimization is not used in QEMU, the execution threads frequently enter *dispatcher* for branch target lookups. Although the computation time can be reduced through parallel execution with more threads, the overhead incurred by thread contention can result in significant performance degradation.

Figure 8 shows the breakdown of time for `blackscholes` with *simlarge* input set for QEMU. *Lock* and *Other* represent the average time of a worker thread spent in critical sections (including wait and directory lookup time) and for the remaining code portions, respectively. As the figure shows, the time of *Other* decreases linearly with the number of threads because of increased parallelism. The time of *Lock* increases significantly because the worker threads contend for the critical section within the dispatcher where the serialization lengthens the wait time. Moreover, the time increased from such serialization outweighs the reduced execution time when more worker threads are added. Such high locking overhead dominates the total execution time, and results in poor performance of the parallel PARSEC benchmarks. We can see from several benchmarks (e.g. `blackscholes`, `bodytrack` and

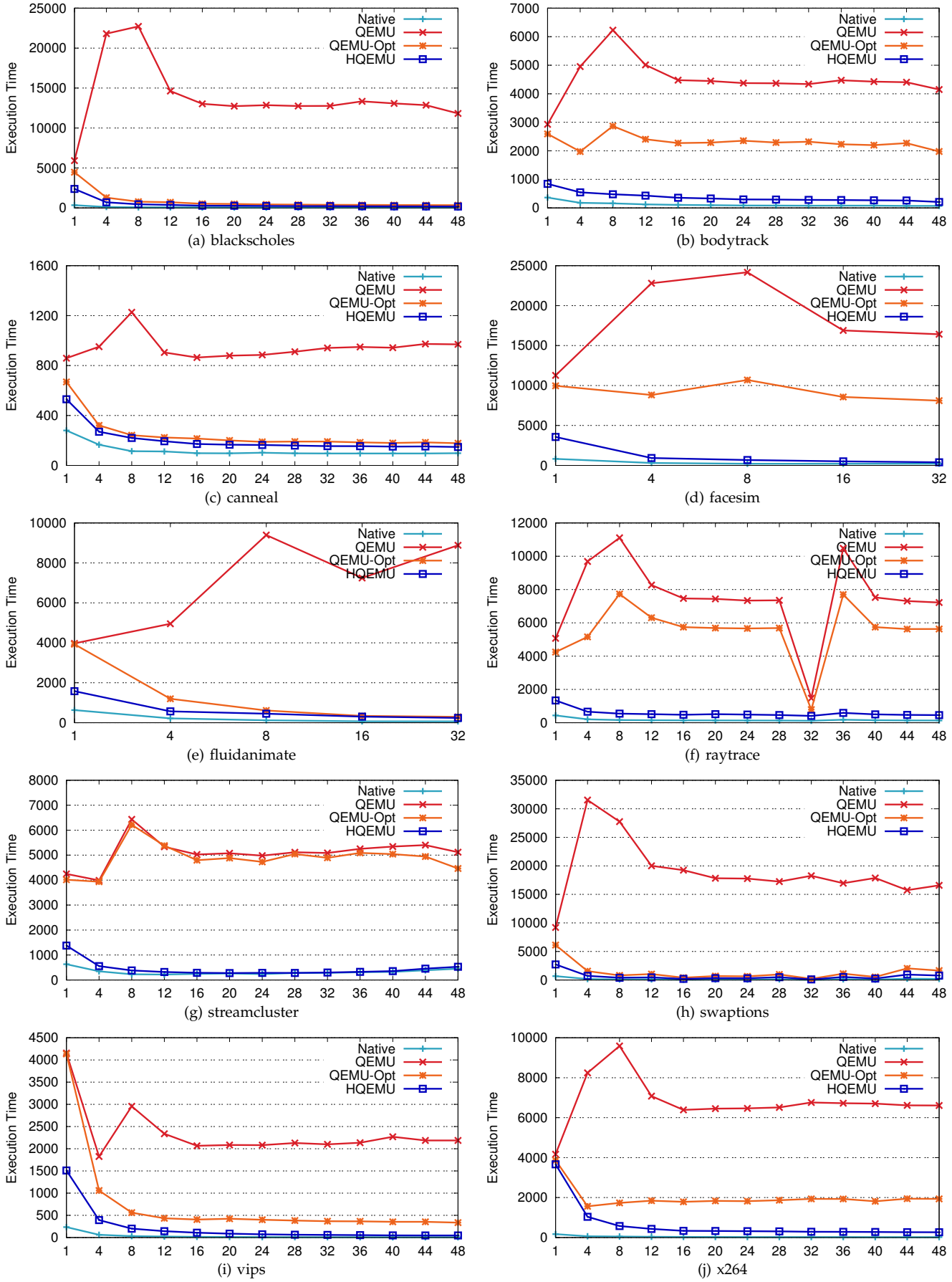


Fig. 7: PARSEC performance results of x86 to x86-64 emulation using native input set. The unit of Execution Time in Y-axis is second. X-axis shows the number of threads.

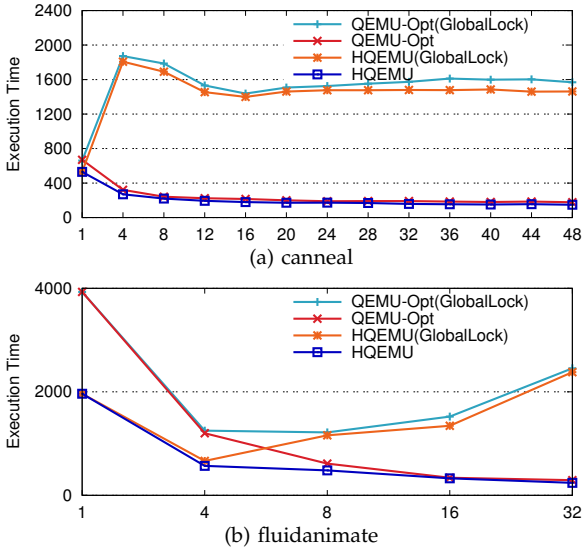


Fig. 10: Comparison of using software memory transactions and global lock scheme for `canneal` and `fluidanimate`. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

`swaptions` in Figure 7) that emulating single thread with the vanilla QEMU results in the best performance compared to its multi-threaded counterparts.

With IBTC optimization, keeping the execution threads staying in the code cache alleviates the large overhead that includes the cost of context switching and threads contention in the dispatcher. Performance gains from IBTC can be observed by comparing QEMU-Opt and QEMU. The trace formation of HQEMU further improves the indirect branch prediction via indirect branch inlining. It also eliminates a large amount of redundant load/store instructions related to storing and reloading the architecture states. Highly-optimized host code generated by LLVM makes HQEMU achieve significant performance improvement over both QEMU and QEMU-Opt. The performance curve with HQEMU is very similar to that of native execution.

Figure 9 shows the overall performance of QEMU, QEMU-Opt and HQEMU over native execution with 32 guest threads. A significant improvement of about 6X speedup on average is achieved with QEMU-Opt over QEMU because of the IBTC optimization. This result shows that the design of shared mapping directory in the QEMU dispatcher is inadequate for emulating multi-threaded guest applications. HQEMU achieves about 25X and 4X speedup over QEMU and QEMU-Opt, respectively. It runs at 28.6% of the native speed on average with 32 emulated threads.

6.2.3 Performance of Lightweight Memory Transactions

In this experiment, we evaluate performance of emulating atomic instructions by comparing *lightweight memory transactions* with the *global lock* scheme. The comparison is conducted using QEMU-Opt and HQEMU. Because the global lock scheme could sometime cause deadlocks while running PARSEC benchmarks, we annotate the addresses that could cause data races and insert lock-unlock pair while translating the instructions that reference these addresses. Figure 10 shows the results for benchmark `canneal` and `fluidanimate`. As the figure shows, the scalability is poor with the *global lock scheme* for both QEMU-Opt and HQEMU. The performance remains poor when emulating multiple threads, and the performance of `fluidanimate` even starts to degrade when the number of threads increases over 4 threads. In contrast, the performance improves linearly with the number of threads using *lightweight memory transactions*, about 8X and 9X speedup over the *global lock scheme* with 32 threads for `canneal`. and `fluidanimate`, respectively. Benchmark `streamcluster` also shows slight improvement (2X with 32 threads) using lightweight memory transactions. No significant improvement is observed for the rest of benchmarks because they have fewer thread contentions for shared memory locations at runtime.

6.2.4 PARSEC Results for x86-32 to ARM Emulation

Figure 11 illustrates the results of three benchmarks for x86-32 to ARM emulation with *simlarge* input sets. As shown in Figure 11(a) and 11(b), the performance of QEMU does not scale well for benchmark `blacksholes` and `swaptions`. This is because larger number of threads will likely cause serialization of threads in the QEMU dispatcher. The performance of QEMU on the ARM platform does not degrade as significantly as that on the x86-64 platform (e.g. Figure 7(a)). This is because synchronization on the ARM platform's single chip multiprocessor (CMP) is much less expensive than that on the AMD Opteron machine whose 8 processors are based on the nonuniform memory architecture (NUMA). For QEMU-Opt and HQEMU, the results are similar to those on the x86-64 host. Figure 11(c) shows the performance result of `canneal` compared with the global lock scheme. In Figure 11(c), the result of native run is not shown because `canneal` includes some code written in assembly language, currently not supporting the ARM architecture. The only way to run `canneal` on the ARM platform is through binary translation. As the figure shows, using lightweight memory transactions, it also achieves better performance than the global lock scheme, with about 26% improvement when emulating 4 execution threads with HQEMU.

7 Related Work

Dynamic binary translation is widely used for many purposes: transparent performance optimization [7], runtime profiling [14], [15] and cross-ISA emulation [16]. With the advances of multicore architectures, several multithreaded DBT systems exploiting multicore resources for optimization have been proposed in the literatures. However, most of them have very different objectives and approaches in their designs.

A very relevant work to HQEMU is [17] which also integrates QEMU and LLVM. In their system, the authors

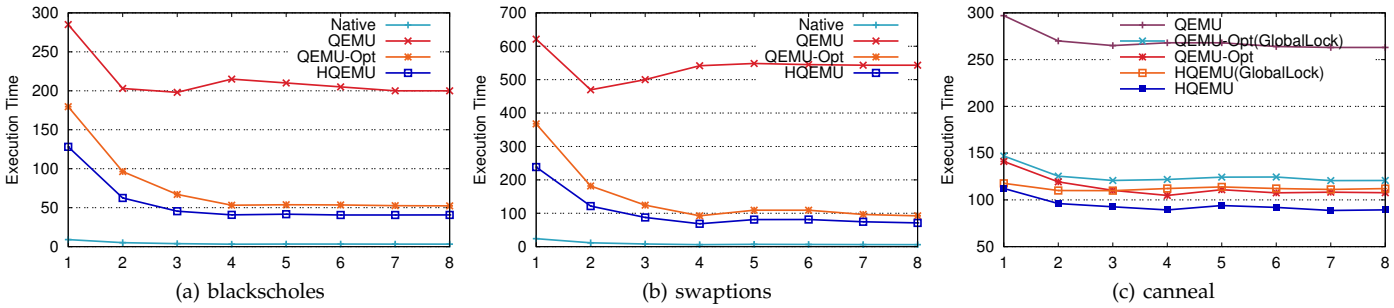


Fig. 11: PARSEC results of x86-32 to ARM emulation with simlarge input sets. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

target small programs with ARM to x86-64 emulation. It sends one block at a time to LLVM when the TCG translator determines it is worthy of optimization. Hence, the performance of the translated code was very poor. They also did not consider the retargetability issue in their DBT. It requires a different binary to LLVM translator for each different ISA, instead of using TCG as an IR as in HQEMU. Unlike their framework, HQEMU applies sophisticated LLVM optimization on traces. Therefore, it can benefit from the advantage of long traces. HQEMU also exposes the opportunities to eliminate redundant load/store instructions during code region transitions.

Brander et al. [18] also use LLVM for JIT compilation. They perform hot region profiling with a much higher threshold to keep the LLVM compile time low. In contrast, we can use a low profiling threshold (i.e. 50) because our LLVM translator runs on another thread, i.e., it does not interfere with the execution threads, and thus the translation overhead can be hidden.

Ha [19] and Bohm [20] proposed the strategy of spawning one or multiple helper thread(s) for JIT trace compilation so that concurrent interpretation and JIT trace compilation can be achieved. Their approach used trace profiling and prediction while interpreting guest programs. Instead of using interpreter, our emulation process is based on JIT compilation. We instrument trace detection routine in the block binary code, and efficiently redirect execution to trace cache as soon as the optimized code is ready. They also did not use HPM to reduce profiling overhead as in HQEMU during trace merging.

COREMU [10], a full-system emulator based on QEMU. It emulates multiple cores by creating multiple instances of sequential QEMU emulators. The system is parallelized by assigning multiple QEMU instances to multiple threads. With the same goal of COREMU, PQEMU [21] takes a different approach to have only one instance of QEMU but parallelizes it internally. Through sophisticated arrangement of critical sections, PQEMU achieves minimal overhead in locking and unlocking shared data. The advantage of their approach is that the performance of emulating multi-threaded programs can be enhanced because each guest thread is handled by a separate emulator thread. However, the emulation of single-threaded program cannot benefit as much because they did not try to optimize the target guest code in each thread. In contrast, HQEMU assigns DBT function to separate threads so very sophisticated optimizations can be applied to each guest thread without incurring overheads on the application threads. The performance of both single-threaded and multi-threaded guest programs can be improved on multicore systems.

Hiniker et al. [6] addresses the trace separation problem in two trace selection algorithms, NET and LEI. The authors focus on the issues of code expansion and locality for same-ISA DBT systems. A software-based approach for trace merging is also proposed. Davis and Hazelwood [22] also use software-based approach to solve trace separation problem by performing a search for any loop back to the trace head. Our work targets cross-ISA DBT systems and addresses issues of trace separation problem especially for performance and emulation overhead. We reduce redundant memory operations during region transitions and use a novel trace combination approach based on HPM sampling techniques.

Kistler and Franz [23], [24] proposed an optimization framework which continually reoptimize programs based on the execution profiles collected from instrumentation and hardware counters. Their system is based on source code optimizations which reoptimize a high-level and type-safe intermediate format. In contrast, HQEMU operates on the binary images directly. ADORE [25] is a lightweight dynamic optimization system based on HPM. Using HPM sampling profiles, performance bottlenecks of the running applications are detected and optimized. Chen et al. [26] proposed some techniques to improve the accuracy of HPM sampling profiles. These work motivate us to exploit HPM-based sampling techniques for our trace merge algorithm. However, [26] and [25] are not multi-threaded DBTs.

Optimization for indirect branch handling in DBT systems has been studied in several literatures [14], [16]. Pin [14] uses an indirect branch chain, and for each indirect branch instruction, Pin associates it with one chain list. Unlike Pin, we use a big per-thread hash table shared by all indirect branches. Shadow stack [16] is used to optimize the special type of indirect branch, *return*, by using a software return stack. This approach works fine for the guest architecture that has explicit call and return instructions, but it does not work for ARM because function return in ARM can be implicit. In

contrast, we use IBTC for all indirect branch instructions, including returns. Our approach is retargetable for any guest architecture.

8 Conclusion

In this paper, we presented HQEMU, a multi-threaded retargetable dynamic binary translator on multicores. HQEMU runs a fast translator (QEMU) and an optimization-intensive translator (LLVM) on different processor cores. We demonstrated that such multi-threaded QEMU+LLVM hybrid approach can achieve low translation overhead and with good translated code quality on the target binary applications. We showed that this approach could be beneficial to both short-running and long-running applications. We have also proposed a novel trace merging technique to improve existing trace selection algorithms. It can effectively merge separated traces based on the information provided by the on-chip hardware HPM and remove redundant memory operations incurred from transitions among translated code regions. It can also detect and merge traces that have trace separation and early exit problems using existing trace selection algorithms. We demonstrate that such feedback-directed trace merging optimization can significantly improve the overall code performance. We also use the IBTC optimization and lightweight memory transactions to alleviate the problem of thread contention when a large number of guest threads are emulated. They can effectively eliminate the huge contention overhead incurred from indirect branch target lookups and the emulation of atomic instructions.

References

[1] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005, pp. 41–46.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004.

[3] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proc. CGO*, 2012, pp. 104–113.

[4] M. Michael and M. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. PODC*, 1996.

[5] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," in *Proc. ASPLOS*, 2000, pp. 202–211.

[6] D. Hiniker, K. Hazelwood, and M. Smith, "Improving region selection in dynamic optimization systems," in *Proc. MICRO'05*.

[7] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proc. PLDI*, 2000, pp. 1–12.

[8] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa, "Overhead reduction techniques for software dynamic translation," in *Proc. IPDPS*, 2004, pp. 200–207.

[9] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, Sep. 2004.

[10] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "COREMU: a scalable and portable parallel full-system emulator," in *Proc. PPoPP*, 2011, pp. 213–222.

[11] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proc. DISC*, 2002, pp. 265–279.

[12] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems," in *Proc. MICRO*, 2003.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT*, 2008.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavy-weight dynamic binary instrumentation," in *Proc. PLDI*, 2007.

[16] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.

[17] A. Jeffery, "Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU," Master's thesis, University of Adelaide, Australia, 2009.

[18] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, "Fast and accurate simulation using the LLVM compiler framework," in *Proc. RAPIDO*, 2009.

[19] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley, "A concurrent trace-based just-in-time compiler for single-threaded javascript," in *Proc. PESPMA*, 2009.

[20] I. Böhm, T. J. E. von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *Proc. PLDI*, 2011.

[21] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung, "PQEMU: A parallel system emulator based on QEMU," in *International Conference on Parallel and Distributed Systems*, 2011, pp. 276–283.

[22] D. M. Davis and K. Hazelwood, "Improving region selection through loop completion," in *Proc. RESoLVE*, 2011.

[23] T. Kistler and M. Franz, "Continuous program optimization: Design and evaluation," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 549–566, 2000.

[24] T. Kistler and M. Franz, "Continuous program optimization: A case study," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 4, pp. 500–548, 2003.

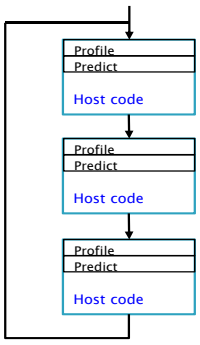
[25] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, vol. 6, pp. 1–24, 2004.

[26] D. Chen, N. Vachharajani, R. Hundt, S.-W. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for FDO compilation," in *Proc. CGO*, 2010, pp. 42–52.

[27] H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani, "Improving the performance of trace-based systems by false loop filtering," in *Proc. ASPLOS*, 2011, pp. 405–418.

[28] K. Hazelwood, G. Lueck, and R. Cohn, "Scalable support for multithreaded applications on dynamic binary instrumentation systems," in *Proc. ISMM*, 2009, pp. 20–29.

[29] H.-S. Kim and J. E. Smith, "Hardware support for control transfers in code caches," in *Proc. MICRO*, 2003.



Algorithm 1: Trace profile and prediction

```
1:  profile_stub_label:
2:  jump < target >
3:  if enable_profile[i] = TRUE then
4:      counter ← counter + 1
5:      if counter ≥ threshold then
6:          enable_predict ← TRUE
7:      end if
8:  end if

9:  predict_stub_label:
10: if enable_predict = TRUE then
11:     if PC exists in recording_list then
12:         enable_predict ← FALSE
13:         CompileTrace(recording_list)
14:     else
15:         Add PC to recording_list
16:     end if
17: end if
```

Fig. 12: An example of trace detection and the pseudo code of the profiling and prediction stubs.

APPENDIX A
TRACE FORMATION AND OPTIMIZATION

In the execution of the translated code from the code caches, we may need to load and store guest registers during code region transition. The problem is that DBT usually translates one code region at a time, often at the granularity of one basic block. Hence, it performs register mapping only within one code region. To ensure the correctness of emulation, the values of the guest registers are required to be stored back to the memory before control is transferred to the next code region, and be reloaded at the beginning of the next code region. Even if two code regions have a direct transition path (e.g. through block chaining, shadow stack [16] or IBTC [8]) and also have the same guest to host register mappings, values of the guest registers still need to be stored and reloaded because we cannot be sure if any of these code regions could be the jump target of an unknown code region.

Because of these independently translated code regions and the resulting frequent storing and reloading of registers during code region transitions, the performance could suffer significantly. Enlarging the code regions, i.e. *trace formation*, can alleviate such overheads. A relaxed version of *Next Executing Tail* (NET) [5] is chosen as our trace selection algorithm, where trace formation is terminated only when the same program counter (PC) is executed again. This relaxed algorithm is similar to the *cyclic-path-based* repetition detection scheme in [27].

In HQEMU, a trace is detected and formed by locating a hot execution path through an instrumentation-based scheme. Figure 12 gives an example of the trace detection and formation scheme. Two small pieces of codes: a *profiling stub* and a *prediction stub*, are inserted at the beginning of each translated code region in the block-code cache. The *profiling stub* determines whether a block is becoming hot or not. The *prediction stub* will then append a hot code block to a list, called *recording list*. The code blocks on the *recording list* will be combined into traces later. The pseudo code of these two stubs is shown in Algorithm 1 in Figure 12.

To detect a *hot trace* for further optimization, we have to locate the head code block of the candidate trace first. During the emulation, the QEMU dispatcher gets the starting PC of the next guest basic block to be executed. The dispatcher looks up a directory to locate the translated host code block pointed to by this PC. If there is a miss in the directory, the emulation module translates the guest block and adds an entry to the directory. If it is a hit, the basic block has been translated before and a cyclic execution path is found. This basic block is a potential trace head, and its associated profiling routine is enabled. The counter is incremented each time this block is executed. When the counter reaches a threshold, the prediction routine is activated to record the blocks following the head block executed in *recording list*. When the prediction routine detects that a head block is already in the recording list, a cyclic path is formed and the trace prediction stops. A request is issued to the LLVM translator through the *optimization request FIFO queue*. The LLVM translator periodically checks whether there are requests on the FIFO queue.

After optimizations by LLVM, the head block of the trace is patched a direct jump (line 2 in Algorithm 1) and the execution is redirected from the unoptimized codes to the optimized codes. This jump patching is processed asynchronously by the LLVM translator, and is transparent to the executing threads. We use self-branch patching mechanism proposed in [28] to ensure the patching is performed correctly when a multi-thread application is being emulated. The store/load of registers to/from memory within a trace is optimized by promoting these memory operations to register accesses (i.e. LLVM Mem2Reg pass). Since a trace is formed because of its hotness, significant block transition overhead is avoided.

APPENDIX B
IMPLEMENTATIONS OF INDIRECT BRANCH TRANSLATION CACHE (IBTC)

Figure 13 illustrates the implementation of our IBTC hash table. The IBTC lookup is implemented as a *helper function*. The function takes the guest address as its parameter and returns the corresponding host address. Each indirect branch instruction (the left box in Figure 13) is translated as calling the lookup routine and then jumping indirectly (JMPr) to the address returned from this lookup routine. The IBTC hash table is declared with the `__thread` identifier. Thus, each execution thread has its own private IBTC hash table and no table locking is required. This implementation has the following advantages: (1) the use of helper function is portable across different host architectures, (2) the threads incur no locking overhead while performing table lookup, and (3) Luk [14] and Kim [29] reported that the hardware BTB prediction rate of the indirect jump will be poor if the lookup's indirect jump is shared by all indirect branches in the application (i.e. all indirect jumps lead to the same dispatch code and a single BTB entry is required to provide all the target addresses). HQEMU does not

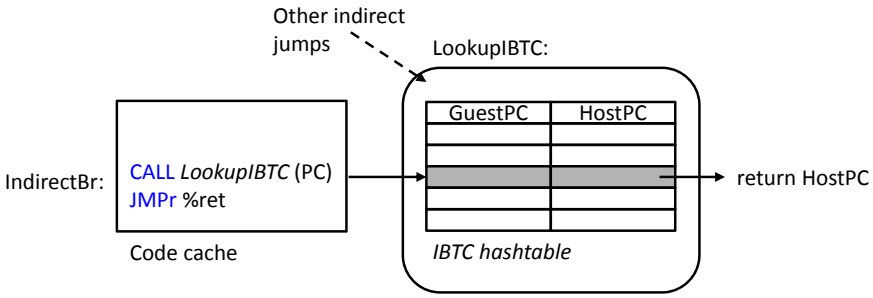


Fig. 13: Implementation of IBTC hash table lookup routine in HQEMU.

TABLE 3: Average search depth of Pin's indirect branch chain for CINT2006 with reference inputs.

	perl	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquan.	h264ref	omnetpp	astar	xalanc.
Tail	6.69	5.50	3.44	1.03	4.52	1.22	3.44	3.22	5.77	3.92	1.02	3.22
Head	7.15	5.25	5.27	1.47	7.81	3.02	5.42	1.36	5.67	4.23	1.01	3.56

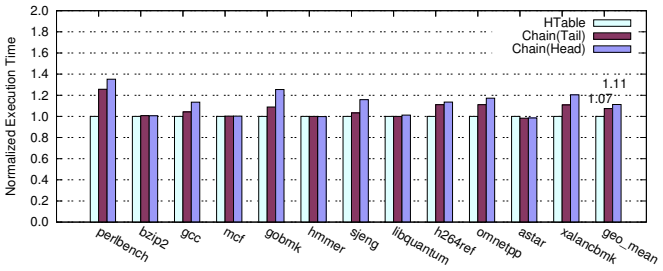


Fig. 14: Performance comparison of Pin's indirect branch chain with IBTC hash table for CINT2006 with reference inputs.

suffer such problem because we inline the IBTC lookup's indirect jump (i.e. the *JMPr* instruction in Figure 13) in the code cache.

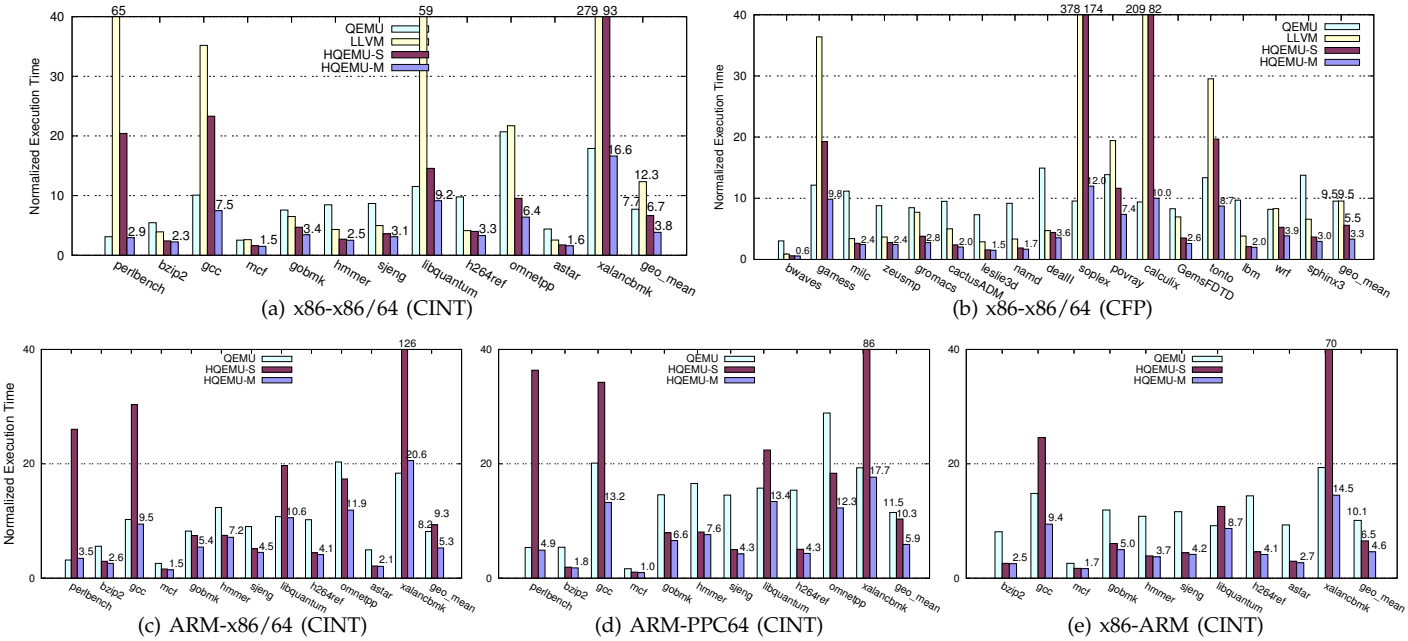
The performance of IBTC hash table depends on its number of hash entries. In HQEMU, we set the number of entry to 65536 and it can achieve more than 99% lookup hit rate for all CPU2006 benchmarks with reference inputs. We also implemented another IBTC with Pin's indirect branch chain [14] for comparison purpose. Figure 14 shows the performance of the indirect branch chain approach (using the IBTC hash table approach as the baseline for comparison) for CINT2006 benchmarks with reference inputs. In this experiment, the maximum chain length is set to 16. We made one change to original Pin's algorithm that when the list of chain reaches the maximum chain length, the address comparison will go to our IBTC hash table of replacement instead of using another hashed comparison chain list. We also compare the performance of attaching a new chain entry to the tail or head of the chain list.

As Figure 14 shows, the indirect branch chain approach results in about 7% and 11% slowdown on average with attachment in tail and head, respectively, compared with IBTC hash table. Table 3 lists the average search depth with indirect branch chain. The result shows that the indirect branch chain approach requires several steps to reach the matched chain entry. Since HQEMU can achieve 99% hit rate (i.e. 99% indirect branches executed requires only one comparison), we demonstrate that our IBTC hash table can achieve better performance.

APPENDIX C
PERFORMANCE OF SHORT-RUNNING PROGRAMS FOR SPEC2006 BENCHMARKS

Figure 15 illustrates the overall performance of *x86-32 to x86-64*, *ARM to x86-64*, *ARM to PPC64*, and *x86-32 to ARM* emulations against the native runs with *test* inputs. The Y-axis is the normalized execution time over native execution time. Note that in all figures, we do not provide the confidence intervals because there was no noticeable performance variation among different runs. Figure 15(a) and 15(b) shows x86/32 to x86/64 emulation results for integer and floating point benchmarks, respectively. In Figure 15(a), the slowdown factors of QEMU over native execution range from 2.5X to 21X and the geometric mean is 7.7X. Most performance results of LLVM are better than or close to those in QEMU except for four benchmarks: `perlbench`, `gcc`, `libquantum` and `xalancbmk`. The reason that LLVM configuration has large slowdowns in these four benchmarks is because too much translation overhead is incurred without being sufficiently amortized. On the other hand, benchmarks `hmmer` and `h264ref` are two of the cases in which the benefit of optimized code outweighs the translation overhead, so that the LLVM configuration outperforms the QEMU configuration.

As for HQEMU-M, all benchmarks run faster than in both QEMU and LLVM configurations, including the four benchmarks that did not perform well in LLVM configuration compared to QEMU. The performance difference is significant. In Figure 15(a), the average slowdown of CINT is 7.7X for QEMU, and 12.3X for LLVM, while the slowdown to native run is only 3.8X for HQEMU-M. In Figure 15(b), the average slowdowns of CFP are both 9.5x for QEMU and LLVM, while the slowdown is only 3.3X for HQEMU-M. Although the performance of HQEMU-S is not as impressive as in HQEMU-M, it still outperforms both QEMU and LLVM.



(a) x86-x86/64 (CINT)

(b) x86-x86/64 (CFP)

(c) ARM-x86/64 (CINT)

(d) ARM-PPC64 (CINT)

(e) x86-ARM (CINT)

Fig. 15: SPEC CPU2006 results of x86/32-x86/64, ARM-x86/64, ARM-PPC64 and x86/32-ARM emulation with test inputs.

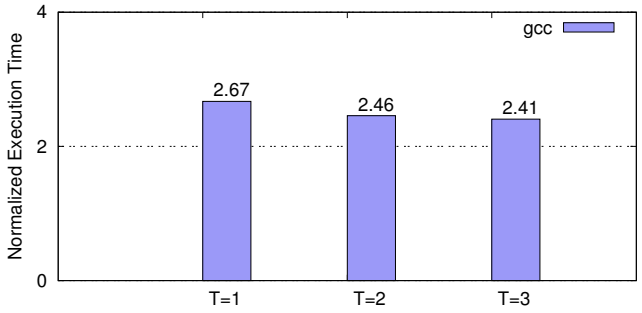


Fig. 16: x86 to x86-64 emulation performance of `gcc` with different number of LLVM translation threads with reference inputs. X-axis shows the number of LLVM threads.

Using small test inputs, fast translation becomes more important, and QEMU outperforms LLVM, based on the averaged slowdown numbers. However, many of the benchmarks can still benefit from better optimized code even with short runs (i.e. with test inputs). This is where HQEMU shines. It benefits from quick start-up as in QEMU, but when the code reuse is high, it switches its execution to the optimized trace code. The longer the code runs, the greater the benefit from optimized traces. Similar results are also observed for ARM to x86-64, ARM to PPC64, and x86-32 to ARM emulations from Figure 15(c) to 15(e).

APPENDIX D
OVERHEAD OF TRACE FORMATION AND MERGING

As shown in column three of Table 2 (see main paper), most CPU2006 benchmarks spend less than 1% of the total time conducting trace translation. `gcc` is an exception. It has a lot of basic blocks, but no clear hot regions. About 160 thousand traces were generated at runtime that took about 215 seconds for emulating the x86-32 guest architecture (280 seconds for emulating ARM). The translation time is about 25% of the total execution time. Thanks to the multi-threaded approach in HQEMU, this significant translation overhead can be hidden by running the translation thread on a different core to minimize the impact to the emulation thread. Compared to the block translation overhead, the QEMU/TCG spends only 3 seconds translating all basic blocks (0.3% of the total execution time).

To evaluate the impact of using different numbers of LLVM optimization threads, we conduct the x86-32 to x86-64 emulation with reference inputs on the quad-core i7 machine and vary the number of LLVM threads from 1 to 3. Since most benchmarks spend less time performing trace translation, there is no noticeable performance difference with the change in the number of LLVM threads for all benchmarks except for `gcc`. The performance result of `gcc` is presented in Figure 16. As the result shows, the normalized execution time is reduced to 2.46X when adding one more LLVM optimization thread and further reduced to 2.41X with three threads (about 11% improvement compared with using only one LLVM thread).

Figure 17 illustrates the breakdown of instructions the emulation thread spends within block-code cache, trace cache and dispatcher in the HQEMU-M configuration for SPEC CPU2006 benchmarks with reference inputs. As the figure shows, most of the instructions are executed within the trace cache. On average, 90% and 95% of the total instructions in CINT and CFP benchmarks are executed within the trace cache, respectively. These

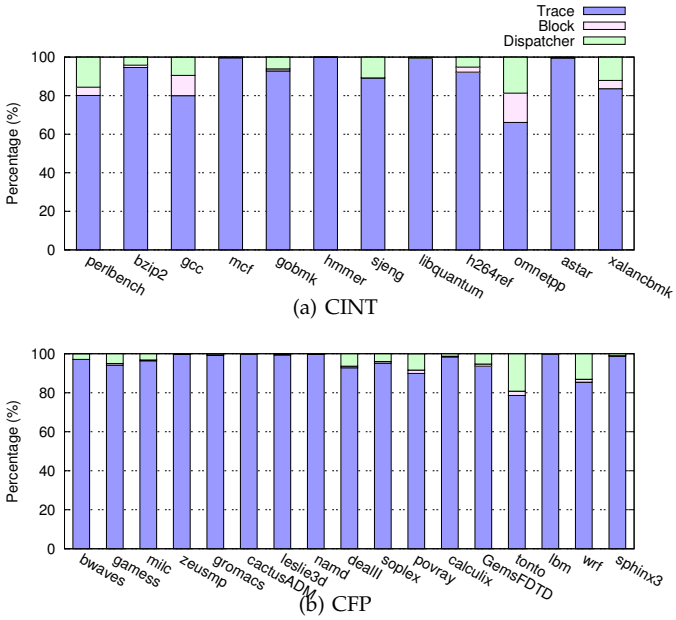


Fig. 17: Breakdown of time with x86 to x86-64 emulation for SPEC CPU2006 benchmarks with reference inputs.

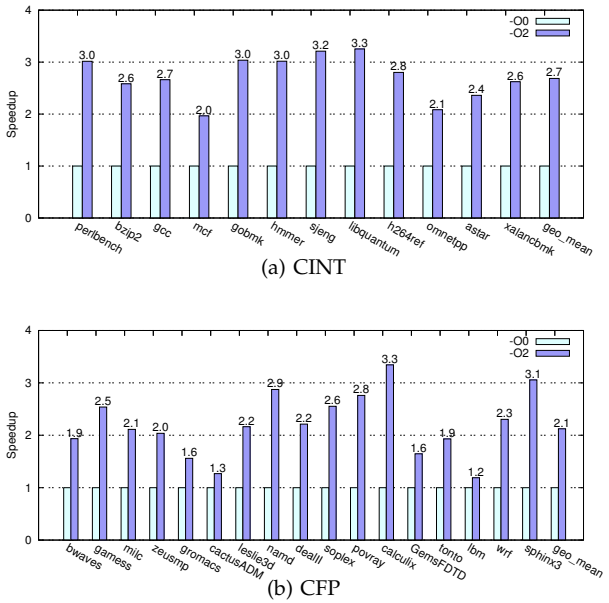


Fig. 18: x86 to x86-64 emulation performance for CPU2006 benchmarks with reference inputs with different LLVM optimization levels.

results show that the optimized code from the LLVM translator is effectively utilized by the emulation thread most of the time in those benchmarks.

APPENDIX E
COMPARISON OF LLVM OPTIMIZATION LEVELS

To evaluate the impact of LLVM optimization, we compare the performance of SPEC2006 benchmarks with two different LLVM JIT optimization levels, -O0 and -O2. The results of -O1 and -O3 are not shown because they use the same set of optimization passes as the -O2 optimization level and thus with the same performance. The -O0 level only applies a fast intra-block register allocation algorithm and dead code/block elimination; the -O2 level applies greedy global register allocation together with several optimizations such as LICM, machine code sinking, instruction scheduling, peephole optimizations, and sophisticated pattern matching for instruction selection, etc. Figure 18 shows the performance speedup of -O2 over -O0 for x86-32 to x86-64 emulation with reference inputs on the quad-core i7 machine. As the result shows, the performance is improved by about 2.7X and 2.1X on average by applying such aggressive optimizations for CINT and CFP benchmarks, respectively. The performance gain mostly comes from better host instructions selected and better register allocation, which achieves minimal stack memory operations.