Efficient and Retargetable Dynamic Binary Translation on Multicores

Ding-Yong Hong, *Student Member*, *IEEE*, Jan-Jan Wu, *Member*, *IEEE*, Pen-Chung Yew, *Fellow*, *IEEE*, Wei-Chung Hsu, Chun-Chen Hsu, Pangfeng Liu, *Member*, *IEEE*, Chien-Min Wang, *Member*, *IEEE*, and Yeh-Ching Chung, *Senior Member*, *IEEE Computer Society*

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 Low-Level Virtual Machine (LLVM) as our building blocks, we demonstrated in a multithreaded DBT prototype, called *Hybrid-QEMU* (*HQEMU*), that it could improve QEMU performance by a factor of $2.6\times$ and $4.1\times$ 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 $2.5\times$ speedup over QEMU for the SPEC CPU2006 integer benchmarks. We also address the *performance scalability* issue of multithreaded 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 ISAs. 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 multithread applications, HQEMU achieves $25\times$ 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; and 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, thus, 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 *Hybrid-QEMU* (HQEMU), which uses QEMU [1], a *retargetable* DBT system as its front end for fast binary code *emulation* and *translation*. However, QEMU lacks a sophisticated optimization back end to generate more efficient code. To this, we use the LLVM compiler [2], also a popular compiler with sophisticated compiler optimization as its back end, together with a *dynamic binary optimizer (DBO)* 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

- D.-Y. Hong and Y.-C. Chung are with the Department of Computer Science, National Tsing Hua University, 30013 No. 101, Section 2, Kuang-Fu Road, Hsinchu 30013, Taiwan.
E-mail: dyhong@sslab.cs.nthu.edu.tw, ychung@cs.nthu.edu.tw
- P.-C. Yew is with the Department of Computer Science, University of Minnesota, 4-192 Keller Hall, 200 Union St, Minneapolis, MN 55455.
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, No. 1, Section 4, Roosevelt Rd., Taipei 10617, Taiwan.
E-mail: {d95006, pangfeng, hsuwc}@csie.ntu.edu.tw
- J.-J. Wu and C.-M. Wang are with the Institute of Information Science, Academia Sinica, No. 128 Academia Road, Section 2, Taipei 11529, Taiwan. E-mail: {wuj, cmwang}@iis.sinica.edu.tw.

Manuscript received 19 Aug. 2012; revised 20 Jan. 2013; accepted 6 Feb. 2013; published online 27 Feb. 2013.
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-08-0747.
Digital Object Identifier no. 10.1109/TPDS.2013.56.

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 multithreaded 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 translated and optimized.

The main contributions of this work are as follows:

- We developed a *multithreaded 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 $2.6\times$ and $4.1\times$ 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 $2.5\times$ speedup over QEMU for SPEC integer benchmarks. For the performance of multithreaded applications, HQEMU achieves $25\times$ 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 multithreaded programs.

The rest of this paper is organized as follows: Section 2 provides a brief overview of our multithreaded 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. Also, extra supporting data, figures, and details are presented in our supplemental materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.56.

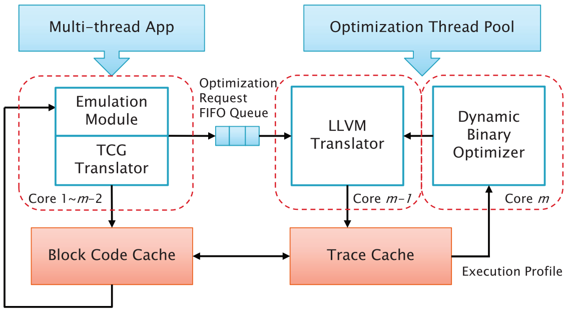


Fig. 1. Major components of HQEMU on an m-core platform.

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.

The core translation engine of QEMU is called *Tiny Code Generator* (*TCG*), which provides a small set of intermediate representation (IR) 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.

Fig. 1 shows the organization of HQEMU. It has an enhanced QEMU as its front end, and an LLVM together with a *DBO* as its back end. 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 back end. 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 back end 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

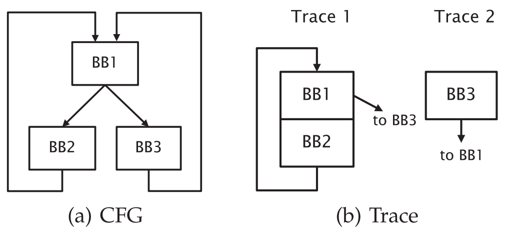


Fig. 2. A CFG of three basic blocks and traces generated with NET trace selection algorithm.

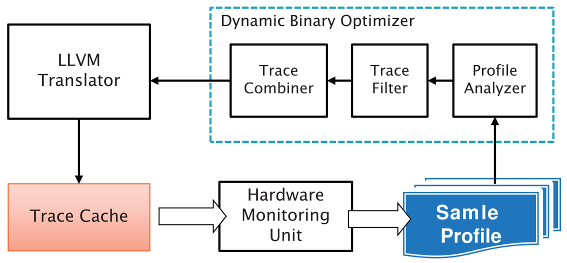


Fig. 3. Workflow of the HPM-based trace merging in DBO.

from its original ISA. This approach simplifies the back-end translator tremendously (see Section 5 for more details). A rich set of program analysis facilities 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 back-end 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 nonblocking 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 reoptimize 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 Section 1 of the supplemental materials, available online.

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. Fig. 2a shows a code example with three basic blocks. Applying NET on this code region will result in two separate traces as shown in Fig. 2b. Trace 1 will have a frequent *early exit* to Trace 2 that could incur significant transition overhead. 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 Fig. 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 percent 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 percent of total sample counts are collected as the *90 percent 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 then the LLVM translator rebuilds 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 percent cover set for future trace combination.

4 Techniques to Enhance Scalability of Emulating Multithreaded 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

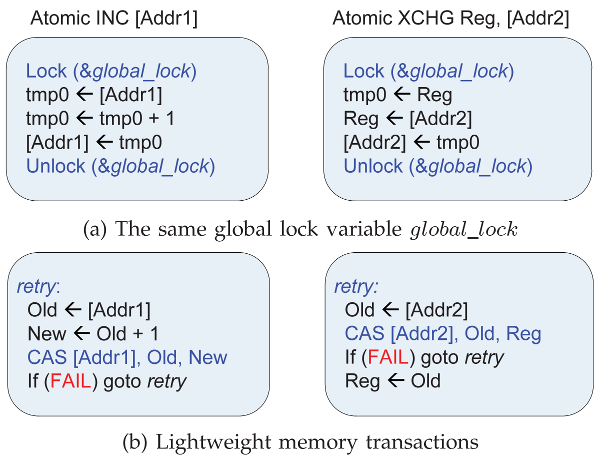
Indirect branches, such as *indirect jump*, *indirect call*, and *return* instructions, cannot be linked in the same way as direct branches because they can have *multiple jump targets*. Making the execution threads go back to the *dispatcher* for the branch target translation each time when an indirect 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 *IBTC* [8] is used. The translation of an indirect branch target with IBTC is performed as a fast hash table lookup 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 Section 2 of the supplemental materials, available online.

During *trace formation*, the prediction routine might record two successive blocks following the path of an indirect branch. We use *Indirect Branch Inlining (IB 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 avoiding 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 multithreaded 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. Fig. 4a shows how two guest atomic instructions, *atomic INC* and *atomic XCHG*, are protected by



```
Atomic INC [Addr1]              Atomic XCHG Reg, [Addr2]

Lock (&global_lock)             Lock (&global_lock)
tmp0 ← [Addr1]                  tmp0 ← Reg
tmp0 ← tmp0 + 1                 Reg ← [Addr2]
[Addr1] ← tmp0                  [Addr2] ← tmp0
Unlock (&global_lock)           Unlock (&global_lock)
```

(a) The same global lock variable *global_lock*

```
retry:                          retry:
 Old ← [Addr1]                   Old ← [Addr2]
 New ← Old + 1                   CAS [Addr2], Old, Reg
 CAS [Addr1], Old, New           If (FAIL) goto retry
 If (FAIL) goto retry            Reg ← Old
```

(b) Lightweight memory transactions

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

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 nonaliased memory locations (e.g., two independent mutex variables in the guest source file) by different threads are serialized because of the same global lock; and 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 nonaliased 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 multiword 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. Fig. 4b 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 nonalias memory accesses and the overhead of expensive locking mechanism are eliminated as a result of the removal of global lock.

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 IR is an effective approach to achieve retargetability, which is used in both QEMU (i.e., TCG) and LLVM. By combining 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 2,000 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 multithreaded 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

TABLE 1
Configurations

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`

DEFAULT= "-O2 -fno-strict-aliasing"

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 multithreaded 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

Fig. 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 Section 3 of the supplemental materials, available online.

Figs. 5a and 5b present the *x86-32 to x86-64* emulation results for integer and floating point benchmarks. Unlike *test inputs*, the programs spend much more proportion of time running in the code caches. As the results show, the LLVM configuration outperforms QEMU because 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 Figs. 5a and 5b, HQEMU-M achieves about 45.5 and 50 percent of the native speed for CINT and CFP benchmarks, respectively. Compared to QEMU, HQEMU-M is $2.6\times$ and $4.1\times$ 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.

Figs. 5c, 5d, and 5e illustrate 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 $2.5\times$ and $2.9\times$ faster than QEMU for CINT with ARM guest to x86-64 and PPC64 host, respectively, and are about 31.2 and 32.3 percent of the native speed, respectively. As for x86-32 to ARM emulation, HQEMU-M achieves $2.8\times$ speedup over QEMU for CINT with *reference inputs*, and is about 37 percent 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. 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

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

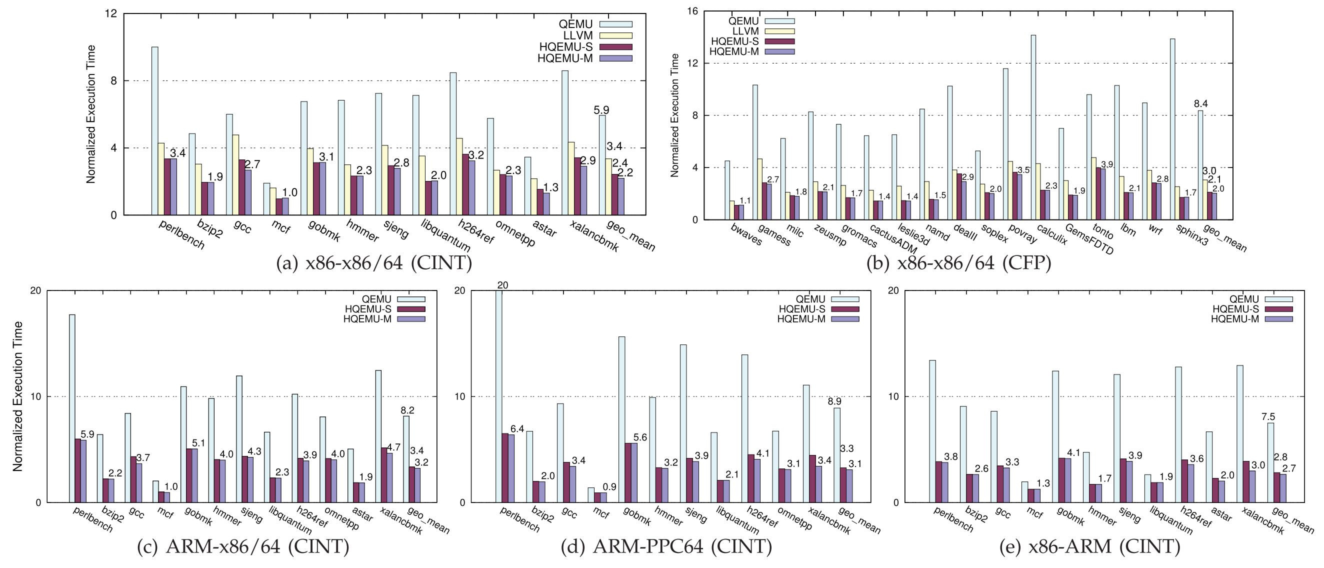


Fig. 5. SPEC CPU2006 results of x86/32-x86/64, ARM-x86/64, ARM-PPC64, and x86/32-ARM emulation with reference inputs.

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*. The results for *CFPs* are listed in Table 4 of the supplemental materials, available online.

Column 2 in Table 2 presents the total number of traces generated in each benchmark. *Column 3* 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 *columns 4* and *5*, respectively. The reduced numbers of memory operations after trace formation (b)-(a) and trace merging (c)-(b) are listed in *columns 6* and *7*, respectively. The improvement rate by trace merging is shown in Fig. 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 Fig. 6).

As for `libquantum`, its hottest code region is composed of three basic blocks, and its CFG is shown in Fig. 2a. 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 Fig. 2a. It keeps the execution staying within this region. Thus, its performance is improved by 82 percent. The analysis of translation overhead and the breakdown of time with *reference inputs* are presented in Section 4 of the supplemental materials, available online.

6.2 Emulation of Multithread Programs

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

TABLE 2
Measures of Traces with x86 to x86-64 Emulation for SPEC CPU2006 Benchmarks with Reference Input

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

(Unit of time: second. Unit of column six and seven: 10^{10} *memory-ops.)*

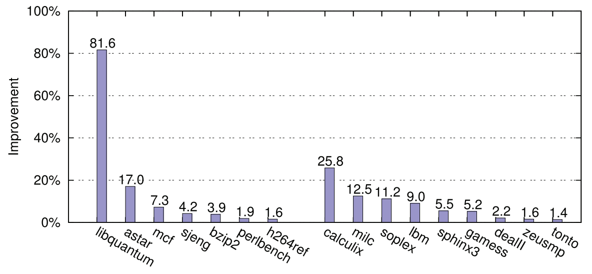


Fig. 6. Improvement of trace merging with x86 to x86-64 emulation for SPEC CPU2006 with reference inputs.

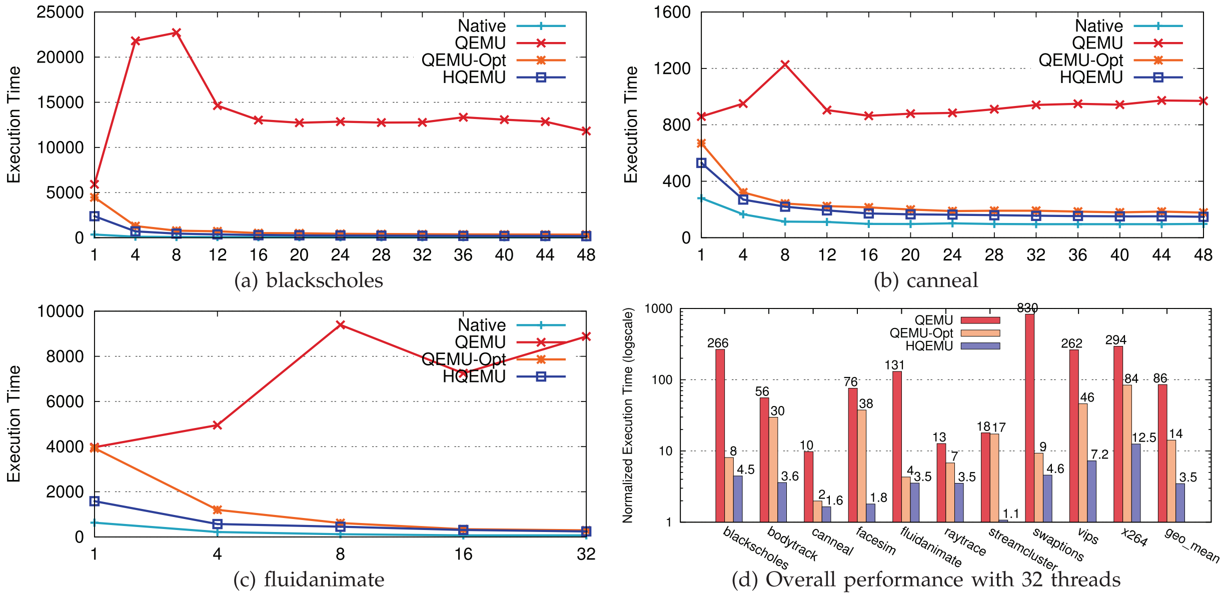


Fig. 7. PARSEC performance results of x86 to x86-64 emulation and overall performance with 32 threads using native input set. The unit of execution time in Y-axis is second. X-axis shows the number of threads.

6.2.1 Experimental Setup

The experiments are conducted on two systems: 1) eight six-core AMD Opteron 6,172 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* (multithread 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

Figs. 7a, 7b, and 7c illustrate the performance results of three PARSEC benchmarks with *native* input sets. The results of all benchmarks are shown in Section 5 of the supplemental materials, available online. The X-axis is the number of worker threads created via the command line argument. The Y-axis is the total time measured in seconds with the `time` command. As shown in Figs. 7a, 7b, and 7c, the performance of QEMU does not scale well. 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.

Fig. 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. Figs. 7a, 7b, and 7c show that emulating single thread with the vanilla QEMU results in the best performance compared to its multithreaded 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

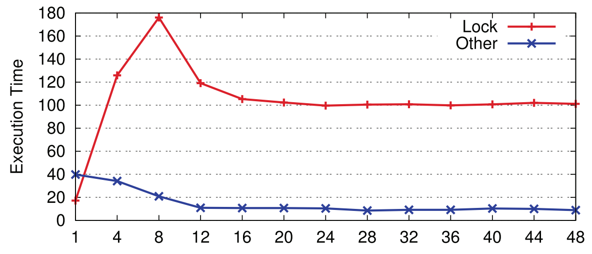


Fig. 8. Breakdown of time with `blackscholes` with simlarge input. The unit of execution time in Y-axis is second. X-axis shows the number of threads.

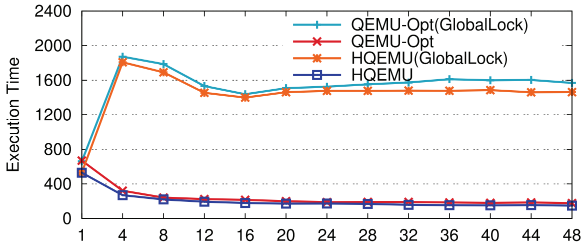


Fig. 9. Comparison of using software memory transactions and global lock scheme for `cannea.l` X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

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.

Fig. 7d shows the overall performance of QEMU, QEMU-Opt, and HQEMU over native execution with 32 guest threads. A significant improvement of about $6\times$ 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 multithreaded guest applications. HQEMU achieves about $25\times$ and $4\times$ speedup over QEMU and QEMU-Opt, respectively. It runs at 28.6 percent 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. Fig. 9 shows the results for benchmark `canneal`. 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. In contrast, the performance improves linearly with the number of threads using *lightweight memory transactions*, about $8\times$ speedup over the *global lock scheme* with 32 threads for `canneal`. The result of `fluidanimate` is shown in Fig. 18 of the supplemental materials, available online. 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

Fig. 10 illustrates the results of three benchmarks for x86-32 to ARM emulation with *simlarge* input sets. As shown in Figs. 10a and 10b, 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., Fig. 7a). 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 eight 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. Fig. 10c shows the performance result of `canneal` compared with the global lock scheme. In Fig. 10c, 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 percent improvement when emulating four 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.

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 [17] 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.

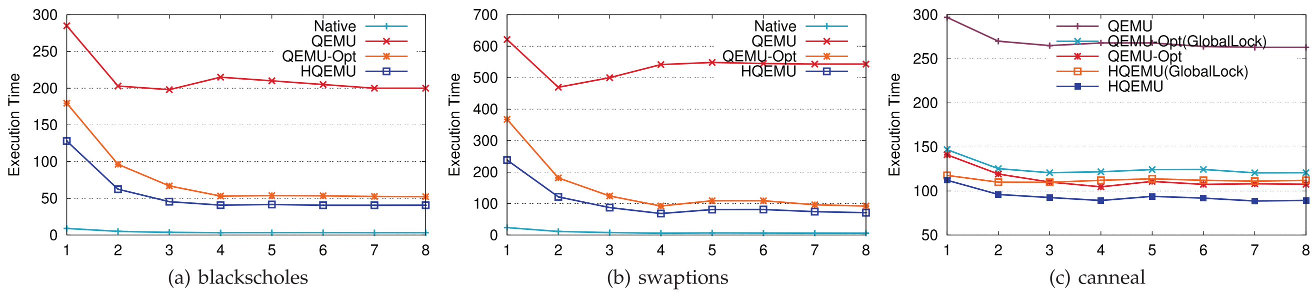


Fig. 10. PARSEC results of x86-32 to ARM emulation with simlarge input sets. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

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. More related works are discussed in Section 7 of the supplemental materials, available online.

8 Conclusion

In this paper, we presented HQEMU, a multithreaded 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 multithreaded $\mathrm{QE\ MU} + \mathrm{LLVM}$ 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.

Acknowledgments

This work was supported in part by Taiwan National Science Council (NSC) grant NSC99-2221-E-001-003-MY3, and was also supported in part by the US National Science Foundation (NSF) grant CNS-0834599.

References

[1] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," *Proc. USENIX Ann. Technical Conf.*, pp. 41-46, 2005.

[2] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proc. Int'l Symp. Code Generation and Optimization: Feedback-Directed and Runtime Optimization (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," *Proc. 10th Int'l Symp. Code Generation and Optimization (CGO)*, pp. 104-113, 2012.

[4] M. Michael and M. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," *Proc. 15th Ann. ACM Symp. Principles of Distributed Computing (PODC)*, 1996.

[5] E. Duesterwald and V. Bala, "Software Profiling for Hot Path Prediction: Less Is More," *ACM SIGPLAN Notices*, vol. 35, pp. 202-211, 2000.

[6] D. Hiniker, K. Hazelwood, and M. Smith, "Improving Region Selection in Dynamic Optimization Systems," *Proc. IEEE/ACM 38th Ann. Int'l Symp. Microarchitecture (MICRO '05)*, 2005.

[7] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 1-12, 2000.

[8] K. Scott, N. Kumar, B.R. Childers, J.W. Davidson, and M.L. Soffa, "Overhead Reduction Techniques for Software Dynamic Translation," *Proc. 18th Int'l Parallel and Distributed Symp. (IPDPS)*, pp. 200-207, 2004.

[9] D.L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," PhD dissertation, MIT, Sept. 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," *Proc. 16th ACM Symp. Principles and Practice of Parallel Programming (PPoPP)*, pp. 213-222, 2011.

[11] T.L. Harris, K. Fraser, and I.A. Pratt, "A Practical Multi-Word Compare-and-Swap Operation," *Proc. 16th Int'l Conf. Distributed Computing (DISC)*, pp. 265-279, 2002.

[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," *Proc. IEEE/ACM 36th Ann. Int'l Symp. Microarchitecture (MICRO)*, 2003.

[13] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (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," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2005.

[15] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (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, Mar./Apr. 1998.

[17] D.M. Davis and K. Hazelwood, "Improving Region Selection through Loop Completion," *Proc. ASPLOS Workshop Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE)*, 2011.

Ding-Yong Hong received the BS and MS degrees in computer science from National Tsing Hua University in 2004 and 2005, respectively. He is currently working toward the PhD degree in the Department of Computer Science at National Tsing Hua University. He is a research assistant at the Institute of Information Science, Academia Sinica. His research interests include dynamic binary translation, virtualization, compiler optimization, and parallel and distributed computing. He is a student member of the IEEE.

Jan-Jan Wu received the BS and MS degrees in computer science from National Taiwan University in 1985 and 1987, respectively, and the MS and PhD degrees in computer science from Yale University in 1991 and 1995, respectively. She is currently a research fellow and the leader of the Computer Systems Laboratory at the Institute of Information Science, Academia Sinica. Her current research interests include parallel and distributed computing, cloud computing, cloud databases, virtualization, and dynamic binary translation on multicores. She is a member of the ACM and IEEE.

Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He also served as the director of the Institute of Information Science (IIS) at Academia Sinica in Taiwan between 2008 and 2011. Before joining the University of Minnesota, he was an associate director of the Center for Supercomputing Research and Development (CSRD) at the University of Illinois at Urbana-Champaign. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington. He served as the editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems* between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization, compilers, and architectural issues related multicore/many-core systems. He is a fellow of the IEEE.

Wei-Chung Hsu received the PhD degree in computer science from the University of Wisconsin-Madison. He first joined Cray Research in 1987 after receiving the PhD degree. He is a professor in the Department of Computer Science and Information Engineering at the National Taiwan University, Taiwan. From 1992 to 1999, he was at the Hewlett Packard Company in California. In 1999, he joined the University of Minnesota. In 2009, he joined the National Chiao-Tung University. In 2013, he joined the National Taiwan University. His recent research concerns the development of both static and dynamic binary translation and optimization systems.

Chun-Chen Hsu received the MS degree in computer science from National Taiwan University in 2004. He is currently working toward the PhD degree in computer science at National Taiwan University. His research interests include dynamic binary translation, virtual machines, runtime code generation and optimizations for Just-In-Time compilation systems, and feedback-directed optimizations.

Pangfeng Liu received the BS degree in computer science from National Taiwan University in 1985, and the MS and PhD degrees in computer science from Yale University in 1990 and 1994. He is currently a professor in the Department of Computer Science and Information Engineering of National Taiwan University, Taiwan, and the deputy director of the TrendMicro Cloud Computing Program of National Taiwan University, the first cloud computing program sponsored by industry in Taiwan. His research interests include parallel and distributed computing, cloud computing, dynamic binary translation, and the design and analysis of algorithms. He is a member of the ACM and IEEE.

Chien-Min Wang received the BS and PhD degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1987 and 1991, respectively. Since then he joined the Institute of Information Science, Academia Sinica, Taipei, Taiwan, as an assistant research fellow, where he is currently an associate research fellow. His major research interest includes parallel and distributed computing, cloud computing, virtualization technology, and dynamic binary translation. He is a member of the IEEE.

Yeh-Ching Chung received the BS degree in information engineering from Chung Yuan Christian University in 1983, and the MS and PhD degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed processing, embedded systems, and cloud computing. He is a senior member of the IEEE Computer Society.