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. Retargetability 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 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
translated and optimized.

The main contributions of this work are as follows:
• We developed a multi-threaded retargetable DBT on
  muticore that achieved low translation overhead and
good translated code quality on the guest binary
  applications. We show that this approach can be ben-
eficial 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 re-
ources when emulating a large number of applica-
tion 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 per-
formance, with techniques to enhance scalability of em-
ulating 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 combining in Section 3; (2) Techniques to en-
hance 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 experi-
mental 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.

2 A TRACE-BASED HYBRID DYNAMIC BI-
NARY 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
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.
DBO uses a hardware performance monitor based (i.e.
HPM-based), feedback-directed runtime optimization
scheme. QEMU is running by the execution thread(s),
LLVM and DBO are running on separate threads de-
pending on their workloads. In its current implementa-
tion, LLVM and DBO are in the same thread. 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 translator/optimizer
running on another thread (and on another core). We use
an enhanced LLVM compiler as the backend because it
consists of a rich set of aggressive optimizations and a

![Fig. 1: Major components of HQEMU on an m-core platform.](image)
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 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 (and on another core). 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. 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 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.

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

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 such 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 (HPM) 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

Fig. 3: Workflow of the HPM-based trace merging in DBO.
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 algorithm 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. 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 Section 2 of the supplemental materials.

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 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 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 avoids checking if memory addresses in such critical sections are aliases at runtime.

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 woken 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 (CAS) 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.

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 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 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 (i.e. Dynamo [7] or dedicated DBTs (i.e. 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. 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. 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 Section 3 of the supplemental materials.

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 proportion 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 *block chaining* and *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 CINT. This is partly due to the translation ability of the current QEMU/TCG. The current QEMU/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. 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 *reference inputs* on x86-64 and PPC64 hosts, respectively, and are about 31.2% and 32.3% of the native speed, respectively. As

---

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

<table>
<thead>
<tr>
<th>Hardware settings – CPU / Memory size</th>
<th>Optimization flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86/64</td>
<td>$DEFAULT</td>
</tr>
<tr>
<td>PPC/64</td>
<td>$DEFAULT -maltivec</td>
</tr>
<tr>
<td>ARM</td>
<td>$DEFAULT -maltivec</td>
</tr>
<tr>
<td>Guest-x86/32</td>
<td>$DEFAULT -m32 -msse -mfpmath=sse</td>
</tr>
<tr>
<td>Guest-ARM</td>
<td>$DEFAULT -ffast-math -msse -mfpu=neon -ftree-vectorize</td>
</tr>
</tbody>
</table>

---
for x86-32 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 re-targetability 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 load/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. The results for CFPs are listed in Table 4 of the supplemental materials.

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 Section 4 of the supplemental materials.

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(a) to 7(c) illustrates 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. 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(c), 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 8.

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. Figure 7(a) to 7(c) show 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 thread 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 7(d) 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
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. Figure 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 8X speedup over the global lock scheme with 32 threads for canneal. The result of fluidanimate is shown in Figure 18 of the supplemental materials. No significant improvement is observed for the rest of benchmarks because they have fewer thread contentions for shared memory locations at runtime.

### 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. Figure 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 8X speedup over the global lock scheme with 32 threads for canneal. The result of fluidanimate is shown in Figure 18 of the supplemental materials. 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 10 illustrates the results of three benchmarks for x86-32 to ARM emulation with similar input sets. As shown in Figure 10(a) and 10(b), the performance of QEMU does not scale well for benchmark blackscholes 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 10(c) shows the performance results of canneal compared with the global lock scheme. In Figure 10(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.
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.

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.

8 Conclusions

In this paper, we present 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

Ding-Yong Hong received the B.S. and the M.S. degree in Computer Science from National Tsing Hua University in 2004 and 2005, respectively. He is currently a Ph.D. student in the Department of Computer Science at National Tsing Hua University, and a research assistant in 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 IEEE.

Jan-Jan Wu received the B.S. and the M.S. degree in Computer Science from National Taiwan University in 1985 and 1987 respectively, and the M.S. and Ph.D. in Computer Science from Yale University in 1991 and 1995 respectively. She is now a research fellow and the leader of the Computer Systems Laboratory in 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 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, D.C. Pen-Chung Yew is an IEEE Fellow. 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 multi-core/many-core systems.

Wei-Chung Hsu received his PhD degree in computer science from the University of Wisconsin, Madison, in 1987. Currently, he is a professor in the department of computer science, at the National Chiao Tung University, Hsinchu, Taiwan. From 1999 to 2009, he was a professor at the computer science and engineering department, at University of Minnesota, twin cities. From 1997 to 1999, he was a runtime optimization architect in the California Language Lab, at Hewlett Packard Company. From 1993 to 1997, he was a technical lead in the compiler team which developed and released an optimizing compiler for the HP PA-8000 systems. Prior to joining HP, he was a senior architect at Cray Research, in Chippewa Falls, Wisconsin. His current research interests include high-performance computing systems and architectures, in particular, runtime optimization systems. At Minnesota, he and his students developed the ADORE runtime optimization system that can dynamically optimize programs on-the-fly for the underlying architecture.

Chun-Chen Hsu received the M.S. degree in computer science from National Taiwan University in 2004. He currently is a Ph.D. candidate in computer science at National Taiwan University. His research interests include dynamic binary translation, virtual machines, run-time code generation and optimizations for Just-In-Time compilation systems, and feedback-directed optimizations.

Pangfeng Liu received his B.S. degree in computer science from National Taiwan University in 1985, and the M.S. and Ph.D. degrees in computer science from Yale University in 1990 and 1994. He is now 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 ACM and IEEE.

Chien-Min Wang received a B.S. and a Ph.D. in electrical engineering from National Taiwan University in 1987 and 1991, respectively. Since then he joined the Institute of Information Science, Academia Sinica, as an assistant research fellow. In January 1996, he became an associate research fellow of the institute. Dr. Wang’s major research interest includes cloud computing, parallel and distributed computing, distributed file systems, dynamic binary translation, and service-oriented architecture. He is a member of IEEE Computer Society.

Yeh-Ching Chung received a B.S. degree in Information Engineering from Chung Yuan Christian University in 1983, and the M.S. and Ph.D. 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.