DQEMU: A Scalable Emulator with Retargetable DBT on Distributed Platforms

Ziyi Zhao
troppingz@gmail.com
Nankai University
Tianjin, China

Zhang Jiang
1711333@mail.nankai.edu.cn
Nankai University
Tianjin, China

Ximing Liu
liuximing@mail.nankai.edu.cn
Nankai University
Tianjin, China

Xiaoli Gong*
gongxiaoli@nankai.edu.cn
Nankai University
Tianjin, China

Wenwen Wang
wenwen@cs.uga.edu
University of Georgia
Athens, GA, USA

Pen-Chung Yew
yew@umn.edu
University of Minnesota
Minneapolis, MN, USA

ABSTRACT

The scalability of a dynamic binary translation (DBT) system has become important due to the prevalence of multicore systems and large multi-threaded applications. Several recent efforts have addressed some critical issues in extending a DBT system to run on multicore platforms for better scalability. In this paper, we present a *distributed* DBT framework, called DQEMU, that goes beyond a single-node multicore processor and can be scaled up to a cluster of multi-node servers.

In such a distributed DBT system, we integrate a page-level directory-based data coherence protocol, a hierarchical locking mechanism, a delegation scheme for system calls, and a remote thread migration approach that are effective in reducing its overheads. We also proposed several performance optimization strategies that include page splitting to mitigate false data sharing among nodes, data forwarding for latency hiding, and a hint-based locality-aware scheduling scheme. Comprehensive experiments have been conducted on DQEMU with micro-benchmarks and the PARSEC benchmark suite. The results show that DQEMU can scale beyond a single-node machine with reasonable overheads. For "embarrassingly-parallel" benchmark programs, DQEMU can achieve near-linear speedup when the number of nodes increases - as opposed to flattened out due to lack of computing resources as in current single-node, multi-core version of QEMU.

CCS CONCEPTS

• **Software and its engineering** → **Virtual machines**; **Distributed systems organizing principles**; *Just-in-time compilers*; • **Computer systems organization** → *Distributed architectures.*

KEYWORDS

Dynamic binary translator, distributed system, distributed emulator

ACM Reference Format:
Ziyi Zhao, Zhang Jiang, Ximing Liu, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2020. DQEMU: A Scalable Emulator with Retargetable DBT on Distributed Platforms. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404403

1 INTRODUCTION

Dynamic binary translation (DBT) systems, such as QEMU [2], Bochs [18], Simics [20] and DynamoRio [5], are important tools for system emulation, profiling and instrumentation. DBT is also an enabling technology for system virtualization. Using DBT, we can translate a guest binary into its semantically-equivalent host binary with a different instruction set architecture at runtime, and run on the host machine.

To run multi-threaded guest binaries, DBT systems such as COREMU [28] creates multiple instances of QEMU, and runs each instance on a physical core. It then uses a thin software layer to handle the communication and synchronization among those QEMU instances. Guest threads are scheduled on these QEMU instances. PQEMU [13], on the other hand, creates multiple instances of virtual CPUs (i.e. vCPUs). Each vCPU is then 1-to-1 mapped to an emulation

*Corresponding author.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404403

thread, and these threads are scheduled by the host OS to run on a multicore platform. Guests threads are scheduled among those vCPUs. PICO [7] and the recent versions of QEMU (after version 2.8) use a scheme similar to PQEMU that maps the guest threads to the native threads of the host system, and take advantage of the multi-threading support in the host OS.

However, all of those DBT systems *only* target *single-node multicore processors* using a shared-memory multiprocessing model. The scalability of such DBT systems is thus limited by the number of physical cores available in *one* node.

It has been shown that, for "embarrassingly-parallel" benchmark programs, QEMU can achieve a comparable scalabilty to the native system as the number of threads grows [7]. However, based on our experiments, the scalablity saturates when the thread number approaches the number of physical cores, which is limited to around 64 cores on most multicore systems today. One way to continue scaling the DBT performance is to go beyond one node and allows more cores to be available. It can also take advantage of the DBT to allow nodes in a cluster to have different kinds of physical cores, and create a *heterogeneous* distributed system to provide even more flexibility.

In this paper, we discussed the design issues in building such a distributed DBT system. It includes thread management, support needed for a distributed shared-memory (DSM) model and synchronization/communication schemes. We have built a prototype based on QEMU, called DQEMU. It includes a page-level, directory-based data coherence protocol, a hierarchical light-weight locking mechanism and a centralized system call scheme with delegation. Several performance optimization strategies are also proposed to mitigate its overheads that include page splitting to mitigate false data sharing among nodes, data forwarding for memory latency hiding, and an hint-based, locality-aware thread scheduling scheme.

Comprehensive experiments have been conducted using micro-benchmarks and workloads from PARSEC[3] benchmark suite. The results show that DQEMU can achieve a good scalability on benchmarks such as `blackscholes` and `swaptions`. In summary, we have made the following contributions in this work.

(1) We have designed and implemented a *distributed* DBT system that scales to multi-node multicore systems. We have built a prototype, called DQEMU, to address the issues required to implement such a system with good scalability.
(2) We have proposed several performance optimization schemes that include page splitting to mitigate false sharing, data forwarding for memory latency hiding, and an hint based thread scheduling for data locality.
(3) Comprehensive experiments are conducted to evaluate the performance of DQEMU and the proposed optimization schemes.

The rest of the paper is organized as follows. Section 2 provides the necessary background to build such a distributed DBT system. Section 3 elaborates on the issues important to building a distributed DBT. Section 4 presents the design and implementation of DQEMU. Section 5 describes several optimization schemes aimed to mitigate its overheads. Section 6 evaluates DQEMU using micro-benchmarks and PARSEC benchmark suite with discussions on their results. Section 7 presents some related work. Finally, Section 8 concludes the paper.

2 BACKGROUND

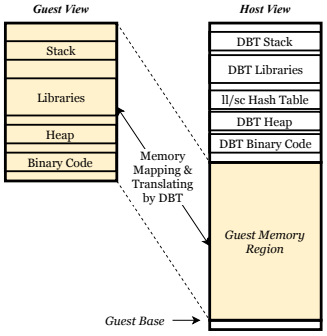

Figure 1: Memory address mapping between guest and host address spaces in a dynamic binary translator QEMU.

A dynamic binary translator takes one instruction or a sequence of instructions from the original binary code (*guest code*) and translates it into the target code (*host code*) on the fly to run on the host machine. The threads in the guest code are implemented as emulated CPU contexts in the DBT system. In most existing DBT systems, each guest thread is encapsulated in a host thread. The host thread is then switched between the *translation mode* (in which it *emulates* or *translates* the guest instructions and addresses) and the *execution mode* (in which it *executes* the translated instructions and maintains the guest CPU state). The CPU state and the register content are maintained in the execution mode to emulate the context of the guest thread. When building a distributed DBT, we keep this mechanism for thread management and thread migration across different nodes.

In addition to the binary translation, the address space of the guest binary is mapped to the host address space as shown in Figure 1. This allows each memory address in the guest code to be translated/mapped to its corresponding host address before the translated instructions are executed. Such

address translation is usually implemented in software. We take advantage of this address translation process to handle the required data coherence protocol *across* the nodes when building our distributed DBT system. The data coherence *within* a node of multicore processor is maintained in hardware by the cache coherence protocol on the host machine.

The guest *system calls* are emulated in detail at the binary level if the guest operating system is emulated at the binary level, i.e. if the DBT is running in the *system mode*. Otherwise, if the DBT is running in the *user mode*, the guest system calls are trapped and emulated by the equivalent host system calls. When building a distributed DBT, one challenge is to maintain a coherent system state across all nodes because most of the system calls have some side effects on the system states and resources.

3 IMPORTANT ISSUES IN A DISTRIBUTED DBT SYSTEM

3.1 Transparency to Programming Models

In general, a guest parallel program can be written in two different programming models: (1) the shared-memory model; or (2) the distributed-memory model. In the *shared-memory* programming model, a parallel program is partitioned into multiple *tasks*. All of the tasks share the *same* address space. These tasks can be formed automatically by the compiler, or by the programmer using the API supported by the runtime such as OpenMP[8], Cilk[4] and Chapel[6]. These tasks are handled by *threads*, e.g. pthreads, and scheduled by OS to run on physical cores. In the *distributed-memory* programming model, a parallel program is partitioned *explicitly* by the programmer into multiple independent *processes*. Each process has its *own* address space and is not visible to other processes. Communication and synchronization among processes are done through explicit message passing using the API supported by the runtime such as OpenMPI[16], MapReduce[11] and Spark[29].

When building a distributed DBT system across multiple nodes, we have to support guest parallel programs written in either programming model because only the binary code is available to the DBT system. It is more straightforward to support guest parallel programs written in the distributed-memory model because they assume the underlying platforms are distributed-memory multiprocessor systems. However, for guest parallel programs using the shared-memory model, we need to maintain a virtual shared-memory multiprocessor system across multiple physical nodes. To accomplish this, we have to support data coherence across multiple nodes transparently to maintain a virtually shared address space. We also have to support data movement and synchronization across multiple nodes that are transparent to the guest programs. As it is more challenging to support the shared-memory programming model, we focus our effort on the system support needed in a distributed DBT system for such a programming model in this paper.

3.2 Data Coherence Across Multiple Nodes

In a multi-node cluster system, each node has its own physical memory. Program data is stored and distributed among those nodes. To emulate a *shared-memory* multiprocessor on such a *distributed-memory* system, a data coherence protocol is maintained across all nodes. A significant amount of research has been dedicated to data coherence protocols, and some recent schemes rely on the emerging RDMA hardware for distributed systems (e.g. [14, 24]).

Those protocols can generally be classified into *centralized* protocols and *decentralized* protocols. The centralized protocols keep the state of each data block on a master node, while decentralized protocols maintain the coherence states separately in each node. In a simple MSI-based coherence protocol, the data coherence state of a data block can be in `Modified (M)`, `Shared (S)`, or `Invalid (I)` state.

In our implementation of the distributed DBT, a centralized page-level directory-based MSI protocol is employed. This is obviously not the most scalable design. Nevertheless, it provides a starting point that has an acceptable performance with minimal engineering effort. Designing more optimal data coherence protocols for such a distributed DBT system is beyond the scope of this paper. We give more details on the implementation of our MSI protocol in Section 4.2, and some of its optimization schemes are discussed in Section 5.1.

3.3 Memory Consistency

A memory consistency model describes the reordering allowed for the memory operations on a core that is observable to other cores. The memory operations within a core can be reordered or bypassed without strictly observing the order specified in the program to improve their performance, especially for multiple-issue out-of-order cores that support branch prediction using speculative execution. Some well-known consistency models include the *sequential consistency* model used in most programming languages, the *TSO* model on Intel X86 processors, and the *relaxed consistency* model on ARM processors. Guest binaries can be generated based on a consistency model that is different from that of the host machine. Some special instructions such as `fence` instructions can be used to make sure the memory operations are committed on the *host* machine that comply with the *guest* machine's consistency model.

Some recent research has produced efficient schemes to enforce memory consistency models across guest and host

machines with different consistency models for DBT systems [19]. Recent QEMU versions has adapted such schemes for cores within a node [7]. We use a two-level approach in DQEMU. At the first level, the current QEMU in each node has incorporated a memory consistency enforcement scheme similar to the one used in PICO [7]. At the second level, to enforce memory consistency across nodes, we use sequential consistency because our data coherence scheme is enforced at the page-level through explicit network communication (similar to message passing). It is by default a sequential consistency model, which enables us to take advantage of the existing consistency enforcement scheme in QEMU within a node, while maintaining strict sequential consistency across the nodes. Even though this may not be an optimal memory consistency enforcement scheme for a distributed shared-memory system, the experimental results in Section 6 show that this 2-level approach can yield acceptable performance.

3.4 Synchronizations

At the binary level, atomic instructions are used to implement various high-level synchronization mechanisms such as *semaphores* and *mutexes*, as well as lock-free and wait-free algorithms [1].

For example, the pair of instructions, `Load-Linked` and `Store-Conditional (LL/SC)`, are used on the reduced instruction set architecture (RISC) such as ARM processors. While `Compare-and-Swap (CAS)` is used on Intel X86 machines that have a complex instruction set architecture (CISC).

It is a challenging problem to provide equivalent semantics across ISAs with different atomic instructions and memory models in DBTs. For example, `LL/SC` can be used correctly to implement a lock-free data structure, while a `CAS`-based implementation may suffer the *ABA problem* [12]. The ABA problem can occur in a multi-threaded program because when two consecutive memory accesses from a thread to the same address that yield the same value (such as A) cannot guarantee that no value update (such as B) by another thread has been made to the same address between the two accesses. In the case of `LL/SC`, a memory update to the address accessed in the pair of `LL/SC` instructions can be detected through hardware that prevents the ABA problem (i.e. SC will fail). But, when the guest `LL/SC` pair are emulated by the host `CAS` instructions to protect the same memory address accessed by LL and SC, the ABA problem could occur without a careful implementation [19, 21, 26, 28].

When building a distributed DBT, we have to handle the synchronization events among all virtual CPUs. Considering that the virtual CPUs may be deployed on different nodes, we have implemented a two-level mechanism to handle atomic instructions, i.e. an inter-node synchronization protocol combined with an intra-node synchronization through atomic instruction translation. We have also avoided the ABA problem by emulating the `LL/SC` across nodes with a global *LL/SC hash table*. More details are presented in Section 4.4.

4 SYSTEM DESIGN AND IMPLEMENTATION

Figure 2 shows the overall organization of DQEMU. As mentioned in Section 3, we use a centralized protocol to enforce data coherence among nodes to simplify our implementation. Hence, we designate a node as the `Master Node` and the rest of the nodes are `Slave Nodes`. Each node runs an instance of DQEMU, i.e. we have a cluster of DQEMU instances in our system. However, to simplify our presentation, if there is no confusion, we will call the entire system DQEMU, instead of *"a cluster of DQEMU instances"*.

In a thread-based programming model, there is generally a *main thread* initiated from the `main` function in the guest binary (represented as the solid bold wavy line in the master node in Figure 2). The remaining guest threads are spawned by the main thread directly or indirectly, and are scheduled among the slave nodes and the master node. Every guest thread is encapsulated as a `TCG-thread` in DQEMU (represented as the solid thin wavy line in the nodes in the figure). A `TCG-thread` is a thread created by QEMU that switches between *translating* the guest binary and *executing* the translated host binary as needed.

There are two kinds of *helper threads* working for the guest threads to carry out various logistic functions. They are (1) `communicator thread` that handles data transfer and syscall requests from a slave node (shown as a dotted wavy line in each slave node), and (2) `manager threads` that handle data coherence protocol and execute syscalls requests from slave nodes (shown as the broken wavy lines in the master node). Each slave node will have a manager thread as its helper thread residing in the *master* node.

A unified distributed shared-memory address space is formed that includes all of the *guest memory regions* (shown in Figure 1) in all DQEMU instances . We use a page-level directory-based data coherence protocol in DQEMU, and the master node is responsible for maintaining the directory and enforcing the coherence protocol.

In the user mode, system resources such as semaphores and files are managed by the host operating system. Guest threads request and access the resources through system calls. The global state of the system and resources are maintained centrally by the master node. The system calls from a slave node will be forwarded to the master node through its communicator thread, and handled by its corresponding manager thread on the master node. The details of the syscall delegation mechanism are explained in Section 4.3.

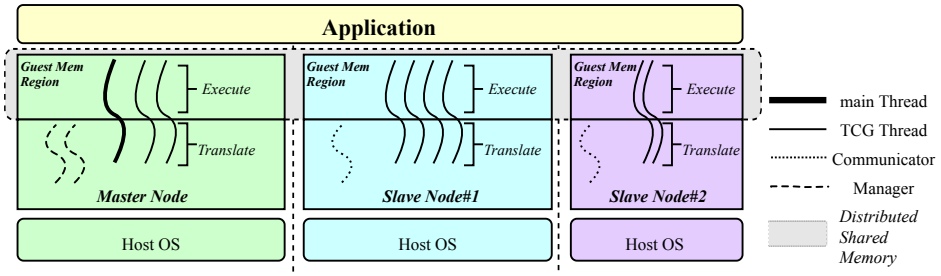

Figure 2: An overview of DQEMU organization

4.1 Thread Creation and Scheduling

When a guest thread needs to be created by an existing guest thread in DQEMU, the creation event is trapped by instrumenting `fork()`, `clone()` and `vfork()` in Linux. There are mainly two steps in emulating the creation of a guest thread: (a) creating a host thread, and (b) setting it up as a `TCG-thread`. To fully emulate this procedure remotely, we clone on the remote node the CPU context of the parent thread. The parameters of the thread creation syscalls are collected and sent to the remote node. And then, the thread creation syscall is executed on the remote node with the two steps mentioned earlier. It holds an identical execution environment as if a thread is created locally. The data required by the created thread will be transmitted to the node through the coherence protocol when accessed.

Thread scheduling and placement have been researched extensively. In a distributed system such as DQEMU, the communication/synchronization and data movement can cause significant overheads. *Locality-aware* thread scheduling schemes, i.e. placing threads that have a significant amount of data sharing on the same node, can eliminate a lot of data movement and communication/synchronization overheads [23, 27]. It is very important to detect data sharing among threads and group them accordingly. To demonstrate the scalability of DQEMU with an acceptable engineering effort, we instrument the application source code with thread semantic information, i.e. a hint-based scheme, so that the DQEMU framework can recognize the hints and schedule the threads correctly. More details are discussed in Section 5.3.

4.2 Distributed Shared Memory Protocol

A page-level MSI-based coherence protocol is used in DQEMU. Read and write operations to a specific page issued by guest threads are captured by a page protection mechanism in DQEMU, and a state machine is maintained and driven by the page fault events. It is worth noting that the page protection mechanism is only for the memory accesses to the *guest memory regions* from the *guest threads*. The *helper threads* in DQEMU will enforce the coherence protocol, but their own memory accesses will not be affected by such a protocol.

The directory of the coherence protocol is maintained by the master node. If a page is not available in a slave node, the `communicator thread` in the node will send a request to its corresponding `manager thread` on the master node. The `manager thread` looks up the directory to locate the page, fetch the page content and forward it to the requesting node. Taking advantage of the centralized page forwarding mechanism, the required page can be pushed to the slave node in advance to reduce memory latency, which is explained in Section 5.2.

Because of the potential *false data sharing* when we enforce data coherence at the page level, we propose an adaptive scheme to adjust the granularity of the coherence enforcement at runtime. Details of this optimization are in Section 5.1.

4.3 Delegation of Syscalls

As mentioned earlier, the system state in DQEMU is maintained in the master node. Each thread accesses the system state with the help of a delegation mechanism on the master node. This approach avoids the need to maintain consistency of the system state in each node, if all syscalls are to be executed locally. The guest thread on the slave node traps its system calls and send the necessary information to the master node that includes guest CPU context, syscall number and the parameters.

We classify the syscalls into *local* and *global* syscalls. *Local syscalls* such as `gettimeofday` can be processed locally without the need to send it to the master node. *Global syscalls* such as `read` and `write` that need to be made visible to all other guest threads, should be sent to the master thread and processed there. Currently, only 19 syscalls that are necessary to support all of our benchmarks are implemented as *global syscalls* while others are left as *local syscalls*. This list could be updated as more benchmarks are supported.

In the *user mode* (as opposed to the *system mode*), DQEMU translates the guest syscalls to their equivalent host syscalls. If the argument of the syscall contains a pointer to the guest

address space, it will be converted to the corresponding host address in the guest memory region in DQEMU. To maintain data coherence, if the arguments of the syscall is a pointer, its corresponding page will be migrated to the master node and updated there. If the syscall modifies the page, the modified page will be set to `Invalidate` on the slave nodes. It will then be forced to fetch the latest version from the master node when it is required.

4.4 Atomic Instructions and Mutex

We use a two-level locking mechanism to handle the synchronization instructions in DQEMU. At the first level, each node handles its atomic instructions through the existing scheme in QEMU. At the next level, the *master node* will work as an arbiter for the atomic operations from different nodes because the data coherence protocol needs to be involved in such atomic operations, which is centralized at the master node.

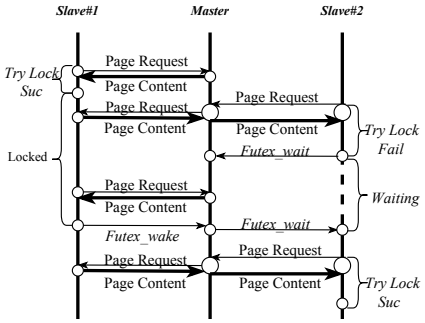

Figure 3: An example of spin lock competition between two nodes.

As shown in Figure 3, a `CAS` instruction is handled similarly to a *write* operation that requires its page to be in the `Modified` state to complete its execution. Taking advantage of the coherence protocol, only one node can exclusively hold the `Modified` page. Threads in other nodes will spin and wait. They may use the syscall *futex_wait* after certain period of time, and be waken up later by the syscall *futex_wake* when the lock is released.

In DQEMU, the `futex` is handled by the syscall delegating mechanism explained in Section 4.3. Generally, a wait queue is maintained in OS to record the status of threads waiting for the `futex` semephore. To emulate this functionality in a distributed environment, we have implemented a *futex table* to support a distributed `futex` syscall.

On the other hand, if the two threads competing for the lock are from the same node and the page is in the `Modified` state, the guest atomic instructions will be translated into the equivalent host atomic instructions, which can be handled efficiently using the existing scheme in QEMU [7].

For the `LL/SC` instructions, a global *LL/SC Hash Table* is maintained in each DQEMU instance. Each time an `LL` instruction is encoutered, its guest thread ID and the guest address are recorded in the hash table. Its corresponding `SC` instruction will look up the hash table and check if the thread ID matches. If the thread ID does not match, the `SC` instruction fails. If the `SC` succeeds, the entry in the hash table is deleted. Also, if the hash table is not empty, every `store` instruction needs to be instrumented to check the hash table and record its thread ID if it has a hit on the hash table. Because the time between an SC and LL is usually very short, the number of such checks on stores is usually very small.

To correctly handle the race condition between the thread ID check and the store operation in `SC`, the `SC` is executed sequentially. For simplicity, we do not check the inter-node `store` instructions. Taking advantage of the coherence protocol, we invalidate the corresponding table entry if the page containing the exclusive address is marked as `Invalidate` state. In other words, if the page containing the exclusive variable is updated on another node, we simply consider the invalid flag has been set. It is a false positive solution, i.e. the `SC` instruction may fail even though the exclusive variable is not modified. However, this only causes additional retries without hurting the correctness of the result.

5 SYSTEM OPTIMIZATION

5.1 Mitigating False Sharing with Page Splitting

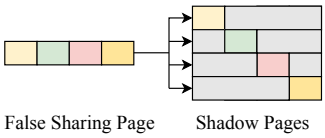

Figure 4: Page Splitting Example. A false-sharing page is split into 4 shadow pages. Each shadow page holds a separate region of the original page with the same page offset.

To take advantage of the existing system support at the page level, we implement our data coherence protocol at the page level. However, two threads on different nodes writing to different areas of the same guest page will cause this page to be passed back and forth between the two nodes, known as *false sharing*. To mitigate this problem, we propose a page splitting scheme to cut down such false sharing, while at the same time maintaining the original page-based coherence protocol.

To do so, a guest page suffering false sharing is split into several independent guest pages (shown as `Shadow Pages`

in Figure 4). Each page holds a separate part of the original page content with *the same page offset*. In this way, the page content is distributed to different pages without false sharing, and each new page can be monitored with its respective protection mechanism using the original data coherence scheme.

A mapping table is also created to hold the mapping of the non-overlapping regions in the false-sharing page to their corresponding shadow pages. Such address translation can be done during of the address translation phase from a guest address to a host address, which is part of the binary translation and thus brings very minimal additional runtime overhead (due to the table lookup).

In the DQEMU framework, the master node is responsible for handling page requests from threads. False data sharing can be detected if a page is written by multiple threads to different parts of the page. The page splitting is then activated by the master node when detected. The master node probes the guest space to find available continuous space for shadow pages, i.e. the address region not used by the guest application. And then, the translation table is updated and broadcast to all salve nodes. The shadow pages will consume some of the virtual space of the application, but the coherence protocol requires no change. Currently, most of the modern processors support 64-bit address space, which is more than enough to accommodate needed shadow pages.

5.2 Latency Hiding with Data Forwarding

Network latency is a critical performance issue in distributed shared memory (DSM) system. There has been a tremendous amount of research on how to mitigate such latency by using prefetching [22], forwarding [25], multithreading, [22, 24], and high speed hardware [14]. In DQEMU, we use data forwarding to hide some of the network latency.

A page-request history is maintained on the master node. The master node will use the history information to forward a page to the target node with a high use probability. We use an algorithm similar to the read-ahead mechanism in the Linux virtual file system [15] to identify the streams of most recently requested pages.

The page forwarding operation is managed by the master node. And the operation is handled by the helper threads. The forwarded pages are inserted in the guest address space, and marked as in `Shared` state to mitigate the overhead of mis-prediction.

5.3 Hint-Based Locality-Aware Scheduling

The memory access latency caused by true data sharing cannot be eliminated by page splitting and page forwarding. It can be an update to the globally-shared variables, or data transfer in programs with a pipelining model, e.g. those using a producer-consumer model. To avoid excessive data transfer among nodes, a hint-based locality-aware thread scheduling scheme is used to balance the workload of computation and memory access.

The data sharing behavior among threads is first analyzed. A group number is assigned to each thread, and threads with extensive data sharing are assigned with the same group number. The group number is specified as an operand of the `no-op` instruction so that the semantic of the program is not changed. The no-op instruction is instrumented, and used as a hint that can be detected by the DBT during the execution. It is used for locality-aware thread scheduling.

It is worth noting that grouping threads/tasks for scheduling is very common in thread-based parallel programming. Pipelining is another popular parallel programming model, which can also be grouped after pipeline unrolling to eliminate complex data dependencies. We find such a scheme can also reduce excessive cross-node memory accesses.

6 EVALUATION

6.1 Experimental Setup

In order to evaluate the DQEMU framework, we have implemented a prototype based on QEMU-3.0.0, which was the most up-to-date version when we started this project. We use a small-scale cluster composed of 7 workstations as the hardware testbed. They are connected by a TP-Link TL-SG1024DT Gigabit Switch with Ethernet cables. The average TCP round-trip latency is on average 55 microseconds (a typical latency in Ethernet-based clusters [10]). The bandwidth of the network is 1Gb/s. Each cluster node is equipped with an Intel i5-6500 quad-core CPU running at 3.30GHz with a 6MB last-level cache and 12GB main memory. The operating system is Ubuntu 18.04 with Linux kernel 4.15.0.

The workload used in the evaluation includes benchmark programs from the PARSEC 3.0 benchmark set [3], and several micro-benchmarks we developed to measure the critical performance of DQEMU. In all of our experiments, we take ARM as the guest ISA and x86_64 as the host. The benchmark programs are compiled to `ARM` binaries with all libraries statically linked. We use QEMU 4.2.0 as the baseline for performance comparison, which is the latest version of QEMU when we did our evaluation. Every experiment is executed three times and their arithmetic means are used to represent the performance results.

6.1.1 Performance Study with Micro-benchmarks. To better understand the performance of the DQEMU, we have designed several simple micro-benchmarks to measure the critical performance parameters including performance scalability, atomic operations and page access latency.

Performance Scalability. In this study, we let the main thread create 120 threads to work on a micro-benchmark, and then wait until the last thread to complete. Each thread tries to calculate π using Tyler series for 65,536 times to simulate a compute-intensive workload. We measure the time from the start of the task creation to the completion of the last thread. We incrementally increase the number of slave nodes in the cluster, and schedule the threads equally among the nodes. We measure the speedup of DQEMU with 120 threads running on different number of slave nodes. The baseline is a single node running 120 threads on QEMU 4.2.0.

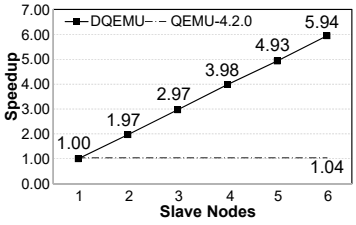

Figure 5: Performance scalability: speedup over a single node for π calculation using Taylor series with 120 threads. Each thread calculates π for 64K times with no data sharing among threads.

Figure 5 shows the performance scalability of DQEMU. This is an "ideal" scalability of DQEMU as the workload is an "embarrasingly parallel" program without any data exchange or synchronization (except a barrier synchronization at the end). It shows that DQEMU can support highly-scalable workload with minimal performance overhead. The dash line in the figure refers to the speedup ratio when running the same workload on one node (with 4 cores). It shows a slowdown of about 4%, which is caused by the remote thread creation and the final barrier synchronization.

Atomic Instructions and Mutex. To measure the synchronization overhead, we use the main thread to create 32 threads and schedule them evenly among the nodes. All these 32 threads try to acquire a `mutex` lock, and release it immediately after acquiring it. We design two scenarios to do the measurements. In the *worst* case scenario, the 32 threads are competing for a single global lock, and each thread tries to acquire and release the lock 5,000 times. In the *best* case scenario, there is a separate private lock for each thread, and each thread tries to operate on the lock 500,000 times. We measure the performance for both scenarios and the results are in Figure 6.

It can be seen that, in the worst case scenario, the best performance outcome is achieved when there is only one slave node. The node will hold the page containing the `mutex` lock variable. The performance gets worse when more nodes are involved. There will be contention among the nodes and

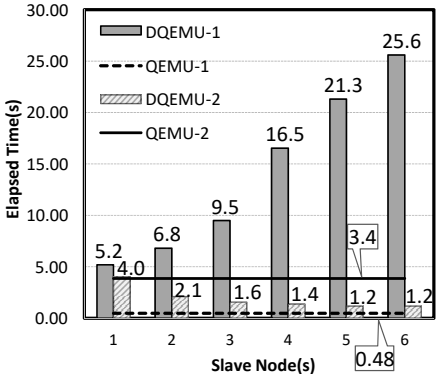

Figure 6: `Mutex` performance. The time to complete `mutex` operations in two scenarios. Scenario-1 (worst case): 32 threads are created to acquire and release a global lock 5000 times (shown as DEQEUMU-1 and QEMU-1). Scenario-2 (best case): 32 threads are created to acquire and release their own private locks 500,000 times (shown as DEQEMU-2 and QEMU-2).

the data coherence overhead for the page holding the `mutex` lock variable. They will become the performance bottleneck. As more nodes get included in the cluster, the more lock contention will be incurred, which will cause `mutex` to fall back to a remote `futex` syscall, leading to the worst performance.

In the best case scenario, the performance of the intra-node atomic operation is almost identical to the original single-node QEMU. However, as the number of nodes increases, the execution time begins to come down linearly as there is less resource contention among 32 threads. Comparing with the single-node DQEMU, a 3.33 times speed up is achieved when there are 6 nodes in the cluster.

Page Access Latency. We measure memory access latency in different situations that include local accesses, remote accesses, and the improvement due to data forwarding and false-sharing mitigation. To perform the measurements, we reserve 1GB guest space on the master node, and use a test program to issue memory access requests from the slave nodes. The test program walks through the reserved space sequentially with an increment of 1 byte. Page forwarding is triggered after there are 4 sequential page requests. We also measure the latency caused by the network transmission when the page accessed is not locally available, i.e. the time consumed by the page fault handler.

In the test of false sharing, 32 threads are created and scheduled evenly among 4 slave nodes. Each thread accesses different sections of the same page, with 128 bytes in each section. Each thread walks through its section sequentially for 20M times with an increment of 1 byte to calculate the average bandwidth. Page splitting is triggered after a page is

requested by different nodes at different addresses over 10 times. The results are shown in Table 1. To make a comparison, we run the same workload on vanilla QEMU-4.2.0 on single node as the baseline.

Access Type	Throughput(MB/s)	Latency(us)
QEMU Sequential Access	173.06	-
Remote Sequential Access	7.88	410.5
Page forwarding Enabled	108.01	83.2
QEMU Access of 128 bytes	20,259	-
False Sharing of 1 Page	2,216	-
Page Splitting Enabled	75,294	-

Table 1: Memory performance of DQEMU. Throughput (MB/s) is the average bandwidth when accessing the target memory region. Latency (microseconds) is the average time needed for the page fault handler to transmit a remote page via coherence protocol. Only the latency of remote memory accesses is measured.

It can be seen in Table 1 that the memory access bandwidth drops dramatically when the target page is not available locally. According to the measurements, a request to a remote page takes about 410.5 microseconds (i.e. about 1.35M cycles) on average on our platform. Since the network bandwidth of our experimental platform is about 1Gb/s, ideally it takes about 40 microseconds (i.e. about 132K cycles) to transmit one page, which is the lower bound of remote page access cost. Compared to a page fault, which usually takes around 2,000 cycles [9], it poses a significant performance overhead. It gets worse if the data locality is poor or there is data sharing among nodes. Therefore, eliminating massive page fault latency with data forwarding and page splitting are necessary and effective measure.

It can be seen that after using page forwarding, the memory bandwidth is increased by about 13.7 times (from 7.88MB/s to 108.01MB/s), which is approaching upper bound of the network bandwidth (1Gb/S). It can also be seen that the memory bandwidth is significantly improved after the false sharing eliminated by page splitting. The memory access performance can be improved by about 33.98 times, and even exceeds the single-node QEMU since the memory access workload is running in parallel on different nodes.

6.1.2 Performance Study on PARSEC Benchmarks. We use several programs from PARSEC 3.0 benchmark suite to evaluate the performance of DQEMU. Since memory access latency is a major bottleneck of the system, we select two types of programs from the benchmark suite based on their memory behavior.

The `blackscholes`, `swaptions` have shown good data locality with light data sharing. They are distributed-system friendly programs, and can be easily fitted into the DQEMU framework. The two programs are configured to run on 32 threads, and the threads are scheduled evenly among the nodes. The native dataset is used as the input.

The test results are shown in Figure 7. It can be seen that these two programs show good scalability as the number of nodes increases. Especially, `blackscholes` shows near linear speedup as the node number increases. We also measure the performance gain of our optimization, i.e. page forwarding and page splitting. Since `blackscholes` is a data intensive workload with a regular access pattern, the performance is improved significantly by page forwarding, from 15.7% to 22.7%, with 17.98% on average. When the two optimizations are applied together, the performance can be improved from 16.6% to 30.9%, wtih 23.8% on average. The `swaptions` is a data-parallel program with little data sharing and has no input. Its false data sharing can be improved by page splitting from 6.1% to 14.7%.

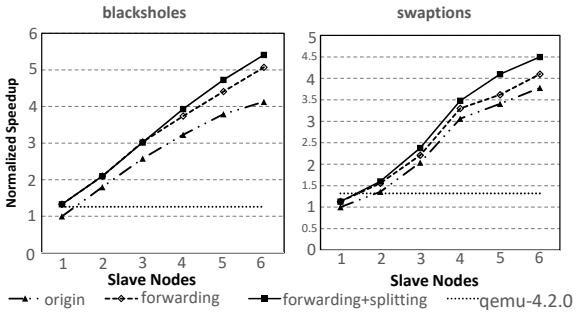

Figure 7: Speedup of two benchmarks, `blackschole` and `swaptions` from PARSEC, running in DQEMU with number of node increase. The speedup is normalized to the system with one slave node.

The program `x264` and `fluidanimate` are the type of parallel programs that use a fork-joint model [3]. They are not very amenable to cluster-based shared-memory systems. The `x264` benchmark uses a pipeline for a stream process with a fork-joint model for each video frame. There is heavy data sharing when it is encoding dependant frames. To improve its inherent scalability and to show the effect of the DQEMU optimization schemes, we slightly modify the code and divide the frames into independent groups and bind them to the threads. Hint information is inserted into the code to notify the grouping information. Note that the purpose of such modification is not to improve the parallelism of the guest binaries, but rather to make them more amenable to show the impact of our proposed optimization schemes.

The `fluidanimate` benchmark divides a large matrix into a grid of small blocks. Each block is assigned to a thread. In every iteration, the threads synchronize with their neighbors. To reduce the data communication, the threads are grouped based on their assigned blocks in the matrix. To fit different

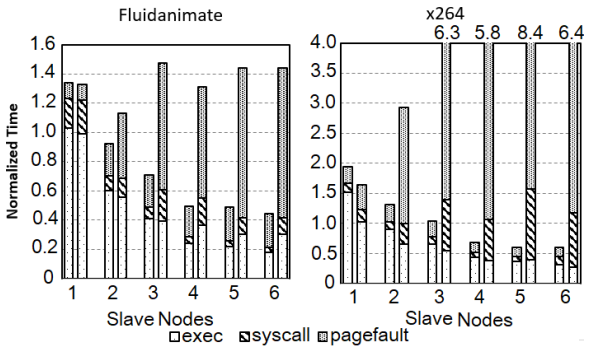

Figure 8: Performance comparison and breakdown of `x264` and `fluidanimate` with 128 threads running on DQEMU. Each bar shows the average time of each thread, with a breakdown on the time spent on execution, page fault and syscalls. The left bar is the performance of using the hint-based locality aware scheduling. The right bar is for scheduling with equal number of threads on each node. The time is normalized to the execution time of QEMU-4.2.0.

number of nodes in the cluster, we embed several grouping strategies, and DQEMU selects the best strategies based on the number of nodes available. Those changes affect less than 1% of lines of the codes, but allow us to see the impact of the optimization schemes provided in DQEMU.

The two program are configured for 128 threads. The native dataset is used as the input. For comparison, we also measure the performance of the scheduling scheme that assign equal number of threads to each slave node in a round-robin manner. Figure 8 shows the execution time and their breakdowns of the two applications. It can be seen that there is a decrease in execution time as the number of nodes increases. However, the time spent on page fault handling (the top shaded section on each bar) increases dramatically if the threads are not properly scheduled. The left bars in Figure 8 show that the hint-based locality-aware scheme can improve the performance quite substantially.

7 RELATED WORKS

Early versions of QEMU [2] emulates multithreading on a multicore processor using a single thread, which significantly limits its performance. There has been a lot of work to improve the parallelism and scalability of QEMU, which allows it to take advantage of the multicore processors.

HQEMU [17] improves QEMU performance by forming traces from the translated basic blocks and optimizing the traces using an LLVM backend. Such a trace formation and optimization process is done concurrently on a separate thread, which can improve the overall performance of QEMU on a multicore host, even though multithreaded guest binaries are still being emulated sequentially by a single thread.

PQEMU[13] tries to emulate multi-threaded guest binaries on a multicore host machine. It improves the scalability of the emulation by mapping the virtual CPU (vCPU) structure in QEMU to separate threads on the host machine using a shared-memory model. COREMU[28] creates multiple instances of QEMU to emulate multiple cores using QEMU's full-system emulation mode.

PICO[7] uses a holistic approach to further improve the scalability of QEMU, which has been adopted in QEMU after version 2.8. All of the above work has made significant contribution not only to the critical issues of scaling QEMU to run on multicore hosts, but also addressing other important issues related to emulating multi-threaded guest binaries such as correctness of emulating atomic instructions, synchronizations and memory consistency models across ISAs.

DQEMU tries to further improve the scalability of QEMU beyond a single-node multicore host to a multi-node distributed system. To the best of our knowledge, DQEMU is the first attempt to extend QEMU to a distributed shared-memory system using a cluster of multicore processors. It has been built without specific hardware support.

8 CONCLUSION

In this work, we have designed and implemented a distributed shared-memory dynamic binary translator (DBT), called DQEMU. DQEMU extends the existing multi-threaded, shared memory-based DBT such as QEMU on one multicore node to a multi-node cluster in which each node is a multicore processor. It has several important features that include a page-level directory-based data coherence scheme, a two-level memory consistency model, a delegated system call scheme, and a distributed shared-memory model.

It also has several unique optimization schemes to improve its performance. It uses a page-splitting scheme to mitigate false sharing among nodes, a data forwarding approach to hide data access latency, and a hint-base locality-aware thread placement scheme to improve data locality.

Comprehensive experiments have been conducted to evaluate the performance of DQEMU. The result shows that DQEMU can achieve good scalability with minimal overhead for highly-scalable multi-threaded guest binaries. For guest binaries with adequate inherent parallelism, the optimization schemes implemented in DQEMU can effectively mitigate overheads caused by false sharing (via page splitting), memory access latency across nodes (via page forwarding), and delegated system calls to maintain coherent system state.

The source code of DQEMU is publicly available at `https://github.com/NKU-EmbeddedSystem/DQEMU`

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (61702286), the National Key Research and Development Program of China (2018YFB1003405), the Natural Science Foundation of Tianjin, China (18JCYBJC15600), the CERNET Innovation Project (NGII20190514), and a faculty startup funding of the University of Georgia.

REFERENCES

[1] [n.d.]. Compare-and-swap-WikiPedia. https://en.wikipedia.org/wiki/Compare-and-swap.

[2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41.

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[4] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, Washington, DC, USA, 265–275.

[6] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[7] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, Piscataway, NJ, USA, 210–220.

[8] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998), 46–55.

[9] Damon. 2019. Cost of a page fault trap. (2019). Accessed 19 Apirl 2019. https://stackoverflow.com/questions/1022 3690/cost-of-a-page-fault-trap.

[10] Jeff Dean. 2007. Software engineering advice from building large-scale distributed systems. *CS295 Lecture at Stanford University* 1, 2.1 (2007), 1–2.

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[12] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 185–192.

[13] J. Ding, P. Chang, W. Hsu, and Y. Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 276–283.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[15] WU Fengguang, XI Hongsheng, and XU Chenfeng. 2008. On the design of a new linux readahead framework. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 75–84.

[16] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[17] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 104–113.

[18] Kevin P. Lawton. 1996. Bochs: A Portable PC Emulator for Unix/X. *Linux J.* 1996, 29es, Article 7 (Sept. 1996).

[19] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 388–400.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (Feb 2002), 50–58.

[21] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[22] Todd C Mowry, Charles QC Chan, and Adley KW Lo. 1998. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. IEEE, 300–311.

[23] Ananya Muddukrishna, Peter A Jonsson, Vladimir Vlassov, and Mats Brorsson. 2013. Locality-aware task scheduling and data distribution on NUMA systems. In *International Workshop on OpenMP*. Springer, 156–170.

[24] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 291–305.

[25] David K Poulsen and Pen-Chung Yew Pen-Chung Yew. 1994. Data prefetching and data forwarding in shared memory multiprocessors. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. IEEE, 280–280.

[26] Philippas Tsigas and Yi Zhang. 2001. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Crete Island, Greece) *(SPAA '01)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/378580.378611

[27] Ke Wang, Xraobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. 2014. Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 119–128.

[28] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: A Scalable and Portable Parallel Full-system Emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. ACM, New York, NY, USA, 213–222.

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.