A Dynamic Binary Translation System in a Client/Server Environment

Chun-Chen Hsu[a], Ding-Yong Hong[b,*], Wei-Chung Hsu[a], Pangfeng Liu[a], Jan-Jan Wu[b]

[a]*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan*
[b]*Institute of Information Science, Academia Sinica, Taipei, 11529, Taiwan*

Abstract

With rapid advances in mobile computing, multi-core processors and expanded memory resources are being made available in new mobile devices. This trend will allow a wider range of existing applications to be migrated to mobile devices, for example, running desktop applications in IA-32 (x86) binaries on ARM-based mobile devices transparently using dynamic binary translation (DBT). However, the overall performance could significantly affect the energy consumption of the mobile devices because it is directly linked to the number of instructions executed and the overall execution time of the translated code. Hence, even though the capability of today's mobile devices will continue to grow, the concern over translation efficiency and energy consumption will put more constraints on a DBT for mobile devices, in particular, for *thin mobile clients* than that for severs. With increasing network accessibility and bandwidth in various environments, it makes many network servers highly accessible to thin mobile clients. Those network servers are usually equipped with a substantial amount of resources. This provides an opportunity for DBT on thin clients to leverage such powerful servers. However, designing such a DBT for a client/server environment requires many critical considerations.

In this work, we looked at those design issues and developed a distributed DBT system based on a client/server model. It consists of two dynamic binary translators. An *aggressive dynamic binary translator/optimizer* on the server to service the translation/optimization requests from thin clients, and a *thin DBT* on each thin client to perform lightweight binary translation and basic emulation functions for its own. With such a two-translator client/server approach, we successfully off-load the DBT overhead of the thin client to the server and achieve a significant performance improvement over the non-client/server model. Experimental results show that the DBT of the client/server model could achieve 37% and 17% improvement over that of non-client/server model for x86/32-to-ARM emulation using MiBench and SPEC CINT2006 benchmarks with test inputs, respectively, and 84% improvement using SPLASH-2 benchmarks running two emulation threads.

Keywords: Dynamic Binary Translation, Client/Server Model, Asynchronous Computing, Region Formation Optimization

1. Introduction

Dynamic binary translators (DBTs) that can emulate a guest binary in one instruction-set architecture (ISA) on a host machine with a different ISA are gaining importance. Dynamic binary translation allows cross-ISA migration of legacy applications which may not have their source code available, as well as porting applications across platforms such as moving Windows applications onto the Android platform. DBT can also handle code discovery problems and self-modifying code which are hard to be dealt with by static binary translators.

Considering the fast growing smart phone and tablet market, for example, many popular applications are either compiled for the ARM ISA or including libraries in ARM native code in the APK package (such libraries may be called via JNI). For vendors offering products in ISAs different from ARM, their customers may not be able to enjoy those applications. Using DBT to migrate such applications is one way to get around this dilemma. For example, DBT has been used to help application migration on workstations and PCs such as FX!32 [1, 2] and IA-32 EL [3]. They have enabled IA-32 applications to be executed on Alpha and Itanium machines successfully in the past.

With rapid advances in mobile computing, multi-core processors and expanded memory resources are being made available in new mobile devices. This trend will allow a wider range of existing applications to be migrated to mobile devices, for example, running desktop applications in IA-32 (x86) binaries on ARM-based mobile devices. However, performance of the translated binaries on such host mobile devices is very sensitive to the following factors: (1) emulation overhead before the translation, (2) translation and optimization overhead, and (3) the quality of

*Corresponding author
Email addresses: d95006@csie.ntu.edu.tw (Chun-Chen Hsu), dyhong@iis.sinica.edu.tw (Ding-Yong Hong), hsuwc@csie.ntu.edu.tw (Wei-Chung Hsu), pangfeng@csie.ntu.edu.tw (Pangfeng Liu), wuj@iis.sinica.edu.tw (Jan-Jan Wu)

the translated code. Such performance could significantly affect the energy consumption of the mobile devices because it is directly linked to the number of instructions executed and the overall execution time of the translated code. Hence, even though the capability of today's mobile devices will continue to grow, the concern over translation efficiency and energy consumption will put more constraints on a DBT for mobile devices, in particular, for *thin mobile clients* than that for severs.

With network accessibility to wireless LAN, network servers are becoming accessible to thin clients. Those network servers are usually equipped with a substantial amount of resources. This opens up opportunities for DBT on thin clients to leverage much powerful servers. However, designing such a DBT for a client/server environment requires many critical considerations.

In this work, we looked at those design issues and developed a distributed DBT system based on the client/server model. We proposed a DBT system that consists of two dynamic binary translators: an *aggressive dynamic binary translator/optimizer* on the server to service the translation/optimization requests from thin clients, and a *thin DBT* on each thin client that performs lightweight binary translation and basic emulation functions for its own.

In our DBT system, we use QEMU [4] as the thin DBT. It could emulate and translate application binaries from several target machines such as x86, PowerPC, ARM and SPARC on popular host machines such as x86, PowerPC, ARM, SPARC, Alpha and MIPS. We use the LLVM compiler [5], also a popular compiler with sophisticated compiler optimizations, on the server side. With such a two-translator client/server approach, we successfully offload the DBT optimization overhead from the client to the server and achieve significant performance improvement over the non-client/server mode.

The main contributions of this work are as follows:

- We developed an efficient client/server-based DBT system whose two-translator design can tolerate network disruption and outage for translation/optimization services on a server.
- We showed that the asynchronous communication scheme can successfully mitigate the translation/optimization overhead and network latency.
- Experimental results show that the DBT of the client/server model can achieve 37% and 17% improvement over that of non-client/server model for x86/32-to-ARM emulation using MiBench and SPEC CINT2006 benchmarks with test inputs, respectively, and 84% improvement using SPLASH-2 benchmarks running two emulation threads on an ARMv7 dual-core platform; it is about 1.7X, 2.4X and 1.7X speedup over QEMU for these three benchmark suites, respectively.

The rest of this paper is organized as follows: Section 2 provides the details of our two-translator client/server-based DBT system, and Section 3 evaluates its effectiveness. Section 4 gives some related work. Finally, Section 5 concludes the paper.

2. A Client/Server-Based DBT Framework

In this section, we first give a brief overview of a DBT system, and present the issues that need be considered for a client/server-based distributed DBT system. We then elaborate on the implementation details of our proposed client/server-based DBT system.

2.1. Design Issues

A typical DBT system consists of three main components: an emulation engine, a translator, and a code cache. As a DBT system starts the execution of a guest program, it fetches a section of guest binary code, translates it into host binary code, and places the translated code into the code cache. The emulation engine controls the program translation and its execution. The section of guest binary code to be translated could be a *single basic block*, a *code trace* (a region of code with a single entry and possibly multiple exits), or an *entire procedure*.

It is crucial for a distributed DBT system to divide system functionality between server and client so as to minimize the communication overhead. The *all-server* approach places the entire DBT system on the server side. The client sends the application binary to the server at first. The DBT system on the server then translates the guest binary into server ISA binary, runs the translated code directly on the server, and sends the results back to the client.

However, the *all-server* DBT is not feasible for applications that need to access peripherals or resources (e.g. files) on the client. For example, an application running on the server will not be able to display a picture on the client because it can only access the screen of the server. To ensure correct execution of all applications, we must run at least portions of the translated code on the client when needed. Although it is possible to divide the execution of the translated code between a client and the server dynamically by the client, system resources such as heaps and stacks, must be carefully monitored so that the division of work can be carried out correctly. It is desirable to have such a capability, but supporting that is quite challenging, and is beyond the scope of this paper. In this work, we assume that the translated code is executed entirely on the client.

It is also crucial for the process of translation and code optimization in a distributed DBT system to tolerate network disruptions. To do so, we need the client to perform stand-alone translation and emulation when network connection or translation service on a remote server becomes unavailable. While in a normal operation mode, the DBT system should take advantage of the compute resources available on the server to perform more aggressive dynamic translation and optimization. Based on these

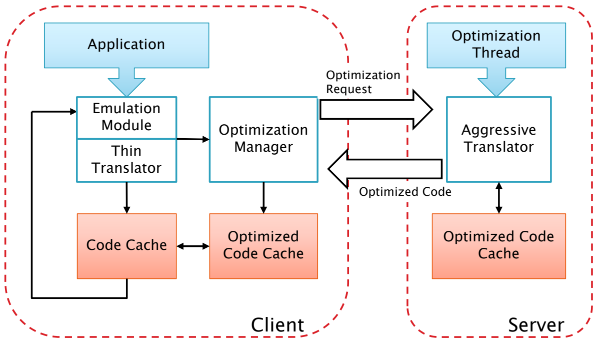


Figure 1: The architecture of the distributed DBT system.

considerations, we proposed a DBT system that consists of two dynamic binary translators: (1) a *thin DBT* that performs lightweight binary translation and basic emulation function on each thin client, and (2) an *aggressive dynamic binary translator/optimizer* on the server to service the translation/optimization requests from thin clients.

2.2. Architecture

The organization of the proposed two-translator distributed DBT system and its major components are shown in Figure 1.

2.2.1. Client

The design goal of the DBT on a thin client is to emit high-quality translated code while keeping the overhead low. The DBT on a thin client consists of all components available in a typical DBT system (the left module enclosed by the dotted line in Figure 1). In order to achieve low translation overhead, the thin client uses a *lightweight translator*. It contains only basic function to translate a guest binary to its host machine code. It does not have aggressive code optimizations nor analysis during translation. Although the performance of the generated host binary may be low due to non-optimized code generation, the thin client can be self-sufficient without the server if the network becomes unavailable.

When the lightweight DBT detects a section of code worthy of further optimization, the optimization manager issues a request to the server for optimization service. After the optimized code is received back from the server, the lightweight DBT needs to perform relocation on relative addresses (e.g. chaining code regions or trampoline) and commits the optimized code to the code cache. The old code is patched with a branch to the optimized code so that the subsequent execution will be directed to the optimized, better-quality host binary.

The optimization manager is run on a dedicated thread. It is responsible for network communication and the relocation of received optimized codes. Unless there are large amount of code segments requiring optimizations, the optimization manager thread is usually idle and does not interfere with the execution thread.

Sending all sections of code in a program to the server for optimization is impractical because it will result in substantial amount of communication overhead, and not all optimized code can achieve the desired performance gain. In particular, optimizing *cold* code regions might not be beneficial to the overall performance. Hence, we must ensure that the performance gain brought on by the optimization can amortize the communication and translation overhead on the server. Our strategy is to translate *all* code regions by the lightweight DBT on the thin client. Only the *hot* code regions that require repeated execution are sent to the server for further optimization. In this work, we first target *cyclic execution paths* (e.g. *loops* or *recursive functions*) as the optimization candidates since the optimized code is supposed to be highly-utilized. Frequently used procedures are also ideal candidates, however, recognizing procedure boundaries for an executable with debugging symbols stripped is challenging. Optimizations for hot procedures are currently under design and implementation. The details of the hot code region detection algorithm are discussed in Section 2.3.

As a hot code region is detected, the client does not attach any address information in the optimization request regarding where the optimized code should be placed. The reason is that, to provide such address information, the client needs to lock and reserve a sufficiently large memory region in the optimized code cache before sending the request to the server. There are three potential drawbacks in such an approach: (1) the client has no way of estimating how large the server-optimized code will be. Hence, it tends to over-commit the reserved space and cause substantial memory fragmentation in the optimized code cache; (2) a long wait time may be incurred by the locking; (3) the next optimization requests are blocked until the pending optimization is completed. Instead, our approach avoids such blocking, and the serialization only occurs when the optimized code needs to be copied into the optimized code cache.

Since the starting address where the optimized code will be placed is not known to the server at runtime, the server cannot perform relocation on the relative addresses for the client. To assist the client on relocation, the server attaches the patching rules to the optimized code and wrap them as a directive to the client so the client can patch the addresses accordingly.

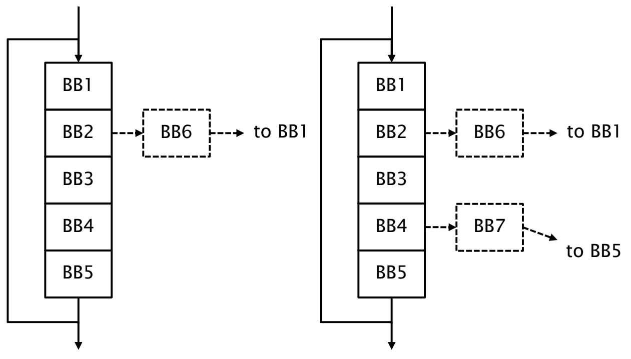
In our implementation, we use the Tiny Code Generator (TCG) [6] in QEMU, a popular and stable retargetable DBT system that supports both full-system virtualization and process-level emulation, as our lightweight translator. TCG translates guest binary at the granularity of a *basic block*, and emits translated code to the code cache. The *emulation module* (i.e. the *dispatcher* in QEMU) coordinates the translation and the execution of the guest program. It kicks start TCG when an untranslated block is encountered. The purpose of the *emulation module* and the *lightweight translator* is to perform the translation as quickly as possible, so we could switch the execution to the translated code as early as possible. When the emulation module detects that some cyclic execution path has become *hot* and is worthy of further optimization, it sends a request to the server together with the translated guest binary in its TCG IR format. The request will be serviced by the *aggressive translator/optimizer* running on the server.

2.2.2. Server

The more powerful server allows its translator to perform more CPU-intensive optimizations and analyses that often require a large amount of memory resources, and thus likely be infeasible on the resource-limited thin clients. For example, building and traversing a large control flow graph (CFG) requires considerable computation and memory space. Furthermore, the server can act as a memory extension to the thin clients. The optimized code cache on the server can keep all translated and optimized codes for the thin clients throughout their emulation life time. The thin client can take advantage of this persistent code cache when it requests the same application that has been translated for another client previously. When the server receives an optimization request on a section of code that is already available in its optimized code cache, the server can send the optimized code back to the client immediately without re-translation/optimization. The response time can be significantly reduced.

After an optimization request is processed whose binary is not translated before, the guest binary fragments are saved and associated to the optimized code in the persistent code cache on the server. When a new optimization request is received by the server, two information are checked to determine if it had been translated/optimized in the following order: start address of each binary fragment (i.e. basic block) and then the complete binary stream are checked. Upon a match to the previous cached one, the cached and optimized code is sent to the client.

For the more aggressive translator/optimizer on the server, we use an enhanced LLVM compiler because it consists of a rich set of aggressive optimizations and a just-in-time (JIT) runtime system. When the LLVM optimizer receives an optimization request from the client, it converts its TCG IRs to LLVM IRs directly, instead of converting guest binary from its original ISA [7]. This approach simplifies the translation process on the server tremendously because TCG IR consists of only about 142 different operation codes instead of a much larger instruction set for most guest ISAs. A rich set of program analyses and powerful optimizations in LLVM can help to generate very high quality host code. For example, redundant memory operations can be eliminated via register promotion in one of LLVM's optimizations. LLVM can also select the best host instructions sequences, e.g., it can replace several scalar operations by one single SIMD instruction, an optimization usually called SIMDization or vectorization, or replace several load/store guest instructions with one load-multiple/store-multiple host instruction.



(a) NETPlus Region (b) Extended NETPlus Region

Figure 2: An example of code region formation with NETPlus and extended NETPlus region detection algorithms.

2.3. Hot Region Formation

Hot code region formation can improve the performance not only from the increased optimization granularity but also from the reduction of network communication overhead in the client/server DBT framework. Next Execution Tail (NET) is a popular trace detection algorithm [8]. However, NET is known for its flaws that often lead to *trace separation* and *early exits* [9], both of which incur trace transition overhead (i.e. saving and restoring guest register states between host registers and main memory during each trace transition) and could significantly degrade the overall performance of typical retargetable DBT systems [10]. Moreover, the problem of trace separation increases the total amount of traces detected and thus increases the number of network communication with the client/server DBT model.

In this work, we use the NETPlus [11] algorithm to detect the hot code regions and overcome such problems. NETPlus enhances NET by expanding the trace formed by NET in the following way: it first uses NET to locate a hot execution path through an instrumentation-based scheme. Two small pieces of stub code are inserted at the beginning of each translated basic block. When a guest basic block is executed the second time, a potential cyclic execution path is found and the first stub, the profiling stub, of the basic block is enabled. Each basic block is associated with a profiling counter. The counter is incremented by the profiling stub each time this block is executed. When the counter reaches a threshold, the second stub, for the purpose of prediction, is started to record the following executed basic blocks. The prediction is terminated when the first basic block is executed again or a maximum length is reached. It then performs a forward search on each exit of the detected trace to expand any possible path going back to the trace head. When the hot region formation is finished, the profiling and prediction procedures are disabled so that no more instrumentation overhead is incurred in future execution. In order to quickly find the hot code region and without incurring too much detection overhead, we set the profiling threshold to 50 and the maximum trace length to 16 in current implementation. Figure 2(a) illustrates a region formed by NETPlus.

The region formation algorithm used in our framework also performs trace expansion on the NET trace exits but does not confine potential paths to the cyclic paths back to the trace head as NETPlus does. Our approach searches for any path that would link back to any block of the trace body. Figure 2(b) illustrates an example of our region formation approach. In Figure 2(b), our approach can form the region of control flow graph (CFG) containing the paths of both `BB2-BB6-BB1` and `BB4-BB7-BB5` where the path `BB4-BB7-BB5` is not allowed with the NETPlus algorithm. With such extension, our approach can form a larger code region and can further reduce the amount of regions detected, and thus results in the elimination of the overhead of region transitions and network communication.

2.4. Asynchronous Translation

The two translators in our distributed DBT system work independently and concurrently. When a thin client sends an optimization request to the server, its lightweight translator will continue its own work without having to wait for the result from the server. Such asynchrony is enabled by an optimization manager running on another thread on the client (see Figure 1) . The advantage of such an asynchronous translation model is threefold, (1) the network latency can be mitigated, (2) the translation/optimization overhead incurred by the aggressive translator is offloaded to the server, and (3) the thin client can continue the emulation process while the server performs further optimization on its request.

2.5. Additional Optimization

To avoid frequent switching between the dispatcher and code cache, the block linking optimization is applied in the un-optimized translated code (default optimization by QEMU) as well as the optimized code to enhance execution performance in the thin clients. After the optimized code is emitted, the un-optimized code is patched and the execution is redirected from the un-optimized codes to the optimized codes. This jump patching is processed asynchronously by the optimization manager. To correctly support concurrent patching to the branch instruction, we use self-branch patching mechanism proposed in [12] to ensure the patching is completed correctly when a multi-thread application is emulated.

In additional to the block linking optimization, region formation and asynchronous translation, we also add optimization for the handling of *indirect* branch instructions, such as *indirect jump*, *indirect call* and *return* instructions. Instead of making the emulation thread go back to the emulation module for the branch target translation each time when an indirect branch is encountered, we build the *Indirect Branch Translation Cache* (IBTC) [13]. The IBTC in our framework is a big hash table shared by all indirect branches and the translation of branch targets looks up the same IBTC for all indirect branches. Upon an IBTC miss, the emulation thread saves the program contexts to the memory, goes back to the emulation module for resolving the address of branch target, and caches the result in the IBTC entry. Upon an IBTC hit, the execution jumps directly to the next translated code region so that context switch back to the emulation module is not required. With IBTC, the execution performance can be improved by increasing the chances to keep the execution staying in the code cache when executing an indirect branch, and the high overhead of saving/restoring program contexts when switching between the emulation module and the code cache is reduced.

Table 1: Experiment setup.

	Server	Client
Processor	Intel Core i7 3.3 GHz	ARMv7 1.0 GHz
# Cores	4	2
Memory	12 GBytes	1 GBytes
OS	Linux 2.6.30	Linux 2.6.39
Network	100Mbps Ethernet / WLAN 802.11g	

	Extra optimization flags
Native (ARM)	-O3 -ffast-math -mfpu=neon -mcpu=cortex-a8 -ftree-vectorize
X86/32	-O3 -ffast-math -msse2 -mfpmath=sse -ftree-vectorize

3. Performance Evaluation

In this section, we present a performance evaluation of the server/client-based DBT framework. We conduct the experiments with emulation of single-thread and multi-thread programs from short-running to long-running benchmarks. The comparison of asynchronous and synchronous translation and the impact of persistent code cache are evaluated. Detailed analysis of the overall performance is also provided to verify the effectiveness of the proposed scheme.

3.1. Experimental Setup

All performance evaluation is conducted using an Intel quad-core machine as the server, and an ARM PandaBoard embedded platform [14] as the thin client. The detailed hardware configuration is listed in Table 1. The evaluation is conducted with the x86/32-to-ARM emulation. In our client/server DBT system, The x86-32 guest binary is fed to the DBT running on the ARM board. The x86-64 server receives optimization requests from the client and sends the translated ARM binary back which are then executed on the ARM processors. We use TCG of QEMU version 0.13 as the thin translator on the client side and use the JIT runtime system of LLVM version 2.8 as the aggressive translator/optimizer on the server side. The default optimization level (-O2) is used for the LLVM

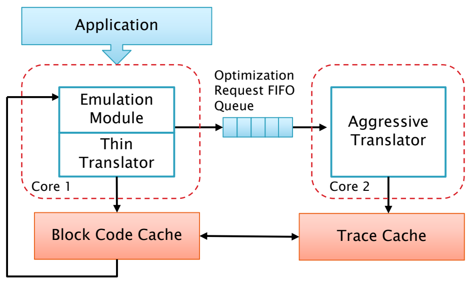


Figure 3: The architecture of the Client-Only configuration.

JIT compilation. The network communication between the client and server is through TCP/IP protocol.

We use MiBench [15] with large input sets, and SPEC CPU2006 integer benchmark suite with test and reference input sets as the short-, medium- and long-running test cases for our performance studies. A subset of the SPLASH-2 [16] benchmarks are also used for studying the emulation of multi-thread guest applications. Although SPEC and SPLASH-2 benchmarks are not widely applied embedded benchmarking sets, we chose to present the results of them for the purpose of cross-comparison because they are the standard benchmark suites which have been widely used to evaluate the DBT systems.

All benchmarks are compiled with GCC 4.4.2 for the emulated x86-32 executables. Except for the default optimization flags in each benchmark suite, we added extra optimization flags, which are listed in Table 1, when compiling the benchmarks. We compare the results to the native runs whose executables are compiled by GCC 4.5.2 on the ARM host also with the extra optimization flags in Table 1. The experiments are conducted in a clean laboratory situation. The performance results are measured as the median from 10 test runs for each benchmark. We do not provide the confidence intervals because there was no noticeable performance fluctuation among different runs.

To evaluate the effectiveness of the proposed framework, **Client/Server**, we compare it with the following two other configurations:

- **QEMU**, which is the vanilla QEMU version 0.13 with the TCG thin translator.
- **Client-Only**, which is similar to the Client/Server framework except that the fast translator and the aggressive translator are both run on the thin client. The thin translator runs the same as the execution thread and the aggressive translator runs on the other thread. The architecture of this configuration is shown in Figure 3. An FIFO queue is built in the DBT system for storing the optimization requests.

The QEMU configuration is used to demonstrate the performance of the thin translator itself, and also as the baseline performance when the network is unavailable. The evaluation in the following experiments is measured with unlimited code cache size and optimized code cache size on the thin client and based on the 100Mbps Ethernet unless changes of the cache size or network infrastructure are mentioned.

3.2. Emulation of Single-Thread Applications

3.2.1. MiBench

The single-thread MiBench is used as the small and short-running benchmark suite. Figure 4 illustrates the overall performance results for MiBench with large input sets. The Y-axis is the speedup over QEMU. As the figure shows, the performance of several benchmarks for Client-Only is slightly slower than QEMU, such as `tiff2rgba`, `tiffmedian` and `sha`, ... etc. For benchmark `jpeg`, `stringsearch` and `pgp`, the performance of Client-Only is even worse than QEMU. The reasons for such poor performance are: (1) The execution thread continues its execution while the aggressive optimizer running on another thread takes too much time to complete. Thus, the optimized code may miss the best timing to be utilized. (2) The instrumentation-based region detection approach (the extended NETPlus described in Section 2.3) incurs considerable overhead. (3) The interference by the optimization thread with the execution threads also incurs some penalty. The performance on the embedded platform is very sensitive to these three factors especially for such short-running benchmarks. We can observe from the three most short-running programs of all benchmarks, `jpeg`, `stringsearch` and `pgp`, their performance is poor with the Client-Only configuration. In contrast, the performance of the other benchmarks is significantly improved compared with QEMU because of the benefits of the optimized code from the aggressive translator. On average, Client-Only achieves 1.2X speedup over QEMU.

As for the Client/Server mode, all benchmarks outperform their Client-Only counterparts. This is because the translation/optimization of code region is much more efficient by the powerful server than by the ARM processors. Thus, the thin client can enjoy the optimized code earlier while the same optimization request may still be queued in the optimization request queue for the Client-Only mode. In addition, the execution thread incurs less interference by the optimization. Therefore, the benefit from optimized code can amortize the penalty from the hot code region detection and results in significant performance improvement over both Client-Only and QEMU. The performance of the benchmark, `jpeg`, `stringsearch` and `pgp`, are still slower than QEMU because the instrumentation overhead is too high to be amortized for these three very short-running programs. Our DBT framework, on average, is about 1.7X faster than QEMU and achieves 37% performance improvement over Client-Only with the client/server model for MiBench benchmarks.

3.2.2. SPEC CINT2006 Benchmarks

The emulation of medium-running and long-running programs are configured by running SPEC CINT2006 benchmarks with test and reference input sets, and their results

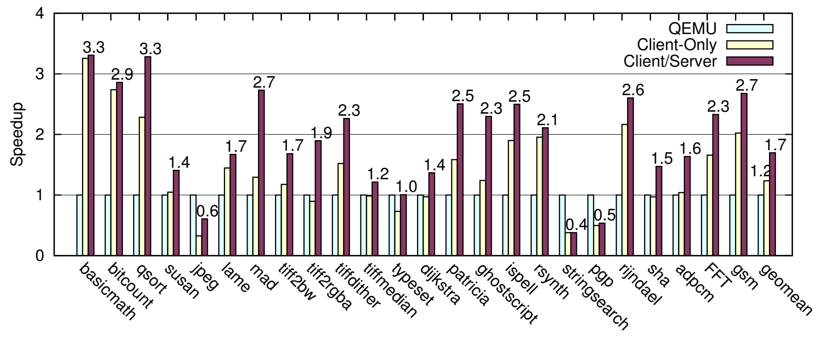
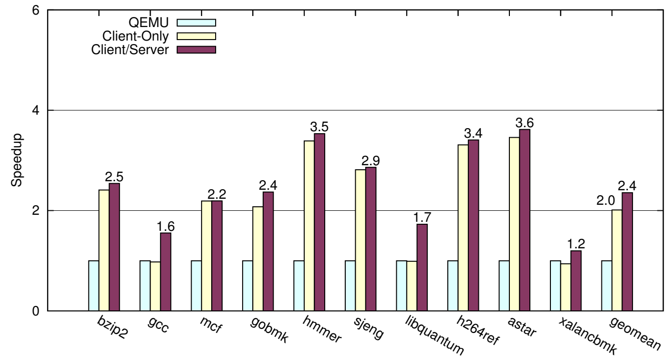
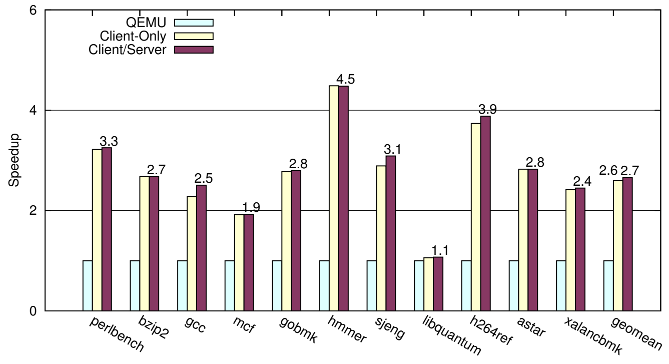


Figure 4: MiBench results of x86-32 to ARM emulation with large data sets. Code cache size: Unlimited.



(a) CINT (Test input)



(b) CINT (Ref input)

Figure 5: CINT2006 results of x86-32 to ARM emulation with test and reference inputs. Code cache size: Unlimited.

are shown in Figure 5 (a) and 5(b), respectively. The performance of `perlbench` and `omnetpp` in Figure 5(a), and `omnetpp` in Figure 5(b) are not listed because both QEMU and our framework failed to emulate these two benchmarks. Figure 5 presents the speedup of Client-Only and Client/Server over QEMU. As Figure 5(a) shows, three benchmarks, `gcc`, `libquantum` and `xalancbmk`, have slight performance degradation with Client-Only over QEMU. The execution thread of them also miss the best timing to utilize the optimized code, especially for `gcc` which generates considerable requests for code region optimization.

As for the Client/Server mode, the performance of `gcc`, `gobmk`, `libquantum` and `xalancbmk`, has the most significant improvement over the Client-Only mode, while other benchmarks have only slight improvement. For `gcc` and `gobmk`, these two benchmarks have much more code regions to be optimized. Therefore, it is beneficial to process such heavy optimization load on the powerful server instead of on the thin client. Client/Server achieves about 60% and 15% performance improvement over Client-Only for `gcc` and `gobmk`, respectively. For `libquantum` and `xalancbmk`, the completion time of the translated regions on the server is earlier than that processed by the client. Client/Server can thus produce the optimized code earlier in time to benefit the overall execution time, which may not be possible with the Client-Only mode. The results show that Client/Server can further improve `libquantum` and `xalancbmk` whereas Client-Only causes performance degradation compared with QEMU. The performance of Client/Server is about 2.4X faster than QEMU on average, and achieves 17% improvement over Client-Only.

In Figure 5(b), the results show that both Client-Only and Client/Server mode outperform QEMU for all benchmarks with reference inputs. The reason is that the optimized code is heavily used with reference inputs. It makes the translation time to account for a smaller percentage of the total execution time. Once the highly optimized codes are emitted to the optimized code cache, the thin client can immediately benefit from the better code compared to the less optimized code executed by QEMU. Since the Client/Server mode tends to eliminate the optimization overhead which is insignificant to the total execution with reference inputs, no significant performance gain is observed with Client/Server over Client-Only in this case. The only exception is `gcc` where about 624 seconds are reduced (10% improvement) with Client/Server. On average, both Client-Only and Client/Server outperform QEMU with a speedup of 2.6X and 2.7X, respectively.

The performance of our client/server-based DBT system is only 3X slower than the native execution with test inputs on average, compared to 7.1X slowdown with QEMU. As for the reference inputs, the average slowdown over native run is only 1.9X, compared to 5.1X slowdown with QEMU. Note that we have only applied the traditional optimizations in LLVM for this experiment, however, with more aggressive optimizations for performance and power

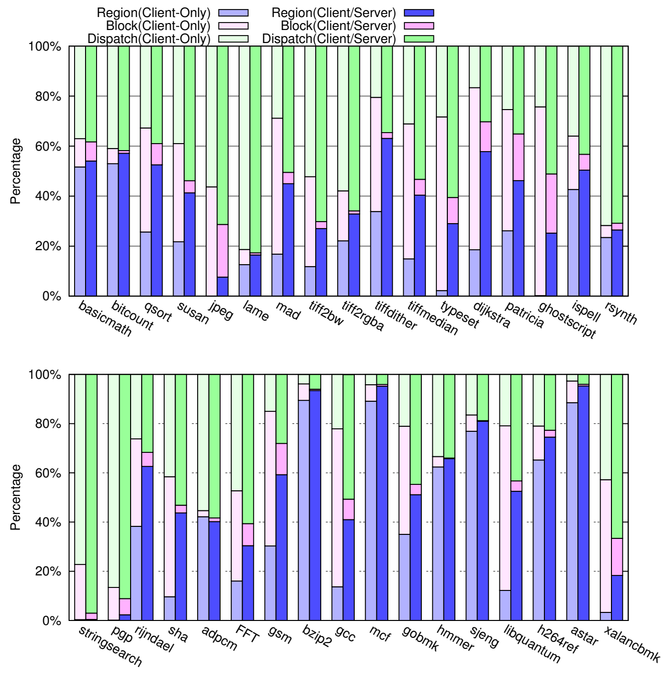


Figure 6: Breakdown of time of x86-32 to ARM emulation for MiBench with large inputs and CINT2006 with test inputs comparing Client-Only and Client/Server mode.

consumption, the benefit of the proposed Client/Server mode could be greater.

3.2.3. Analysis of Emulation of Single-Thread Programs

Figure 6 shows the execution-time breakdown in the Client-Only and Client/Server mode according to the emulation components: *Region* (time spent in the optimized code cache), *Block* (time spent in the code cache) and *Dispatch* (time spent in the emulation module) for the MiBench and SPEC CINT2006 benchmarks. We use hardware performance counters and Linux Perf Event toolkit [17] with event `cpu-cycles` to estimate the time spent in these three components. As the result shows, the percentage of time spent in the optimized code cache increases and the time spent in the code cache reduced significantly with Client/Server over Client-Only mode for all benchmarks. That is, the execution thread of Client/Server spends more time running in the optimized code regions. Column 4 and 6 in Table 2 lists the number of regions generated with Client-Only and Client/Server modes during the whole emulation time period, respectively. The results show that much more regions are generated in the Client/Server mode. This is because the powerful server can perform more efficient LLVM JIT compilation than the ARM thin client. Take benchmark `gcc` as an example, only 4678 regions are optimized by Client-Only, but 16868 regions can be optimized by the powerful server even with less emulation time. That means there are many optimization requests pending in the optimization request queue with Client-Only. The results of the number of regions generated for Client-Only and Client/Server mode also imply that the optimized code can catch the best utilization timing for the execution thread earlier with Client/Server than Client-Only mode because its translation time for each code region is shorter. These results also match the results from the breakdown of time in Figure 6 that the execution thread spends more percentage of time within the optimzed code with Client/Server mode.

We also use hardware performance counters with event `instructions` to count the total number of instructions executed on the thin client for MiBench and SPEC CINT2006 benchmarks. The results for Client-Only and Client/Server are listed in column 5 and 7 in Table 2, respectively. For Client-Only, the instruction count includes the instructions for emulating guest programs and for the aggressive optimization. Since the aggressive optimization is conducted by the server with Client/Server, only the instructions for emulation of guest programs are counted in this mode. The instruction reduction rate of Client/Server is presented in the last column, where on average about 46% and 31% of the total instructions can be reduced for MiBench and SPEC CINT2006 benchmark, respectively. This reduction of instructions results mainly from off-loading the aggressive translation/optimization to the server and also from earlier execution of better quality code from the server.

Column 2 and 3 in Table 2 compares the quality of the translated host code generated by QEMU TCG fast translator and by our DBT system respectively in terms of the expansion rate. The expansion rate is measured as the number of host instructions translated per guest instruction (i.e. the total number of ARM instructions generated divided by that of the x86-32 instructions). As shown in the results, QEMU translates, on average, one guest binary instruction to 8 host instructions. With the LLVM aggressive translator/optimizer, we can reduce that number to 4, and thus achieve the goal of generating higher quality code.

3.3. Emulation of Multi-Thread Applications

3.3.1. SPLASH-2

Figure 7 illustrates the performance results of three configurations for SPLASH-2 benchmarks running one and two emulation thread(s). The Y-axis is the speedup over QEMU with one emulation thread. For QEMU, the performance scales well when increasing the number of threads to two for most benchmarks. However, the performance scalability of Client-Only is poor and even degrades for benchmarks `Cholesky` with two emulation threads, whose scalability ranges from -12%(`Cholesky`) to 24%(`FFT`). For benchmark such as `Barnes`, `Cholesky`, `Ocean`, `Water-Nsq` and `Water-Sp`, the performance results of them are slower than those of QEMU. The poor scalability is due to two reasons: (1) the emulation threads miss the timing to utilize the optimized code, and (2) the aggressive optimization thread would compete the processor resources and interfere with the execution of the emulation threads because the testing ARM platform has only two cores (i.e.

Table 2: Measures of x86-32 to ARM emulation for MiBench with large input sets and SPEC CINT2006 benchmarks with test input sets. *Expansion Rate* represents the average number of ARM instructions translated per x86-32 instruction by QEMU TCG and LLVM translator; *#Region* represents the total amount of regions generated by the optimization thread (Client-Only) and optimization service (Client/Server) during the execution period; *ICount* represents the number of retired instructions on the client platform; *Reduction Rate* represents the ratio of reduced instruction counts.

	QEMU	LLVM	Client-Only		Client/Server		
Benchmark	Expa. Rate	Expa. Rate	#Region	ICount (10^9)	#Region	ICount (10^9)	Reduc. Rate
MiBench (large input sets)							
basicmath	8.0	4.6	195	15.0	196	7.0	54 %
bitcount	8.0	3.3	22	2.2	23	1.9	16 %
qsort	8.1	4.9	59	2.7	73	1.2	56 %
susan	7.9	3.2	45	2.9	117	1.7	43 %
jpeg	7.8	4.5	71	2.5	324	1.6	36 %
lame	7.4	6.0	678	33.2	707	23.1	30 %
mad	8.2	4.1	211	5.4	233	1.4	73 %
tiff2bw	8.0	4.7	68	1.9	65	.79	59 %
tiff2rgba	7.9	4.5	76	2.9	78	1.3	53 %
tiffdither	8.0	4.4	255	6.6	255	2.0	69 %
tiffmedian	8.0	4.6	174	4.9	202	1.6	68 %
typeset	7.5	4.1	167	7.2	1601	2.4	67 %
dijkstra	8.2	4.9	38	1.8	65	.63	65 %
patricia	8.1	4.5	213	5.5	220	2.0	62 %
ghostscript	7.5	4.5	456	13.9	1272	4.3	69 %
ispell	8.1	4.9	374	7.0	365	3.3	53 %
rsynth	8.0	5.1	242	22.1	243	9.3	58 %
stringsearch	8.0	2.5	5	.31	24	.27	12 %
pgp	7.9	3.6	43	1.8	393	1.2	34 %
rijndael	7.6	3.0	55	3.1	52	1.5	52 %
sha	7.7	3.1	43	1.6	48	.52	67 %
adpcm	8.0	3.3	21	3.7	25	3.2	14 %
FFT	7.9	4.6	119	5.8	187	2.7	52 %
gsm	7.7	3.5	147	7.0	383	3.0	57 %
Average	7.9	4.1					46 %
CINT2006 (test input sets)							
bzip2	8.0	3.4	924	71	928	56	21 %
gcc	8.5	5.1	4678	58	16868	24	58 %
mcf	7.9	4.2	275	16	301	11	29 %
gobmk	8.7	4.9	14106	258	19993	153	41 %
hmmer	8.1	4.5	237	56	241	51	9 %
sjeng	8.3	4.8	988	54	1137	36	33 %
libquantum	7.8	4.1	112	3.5	163	1.1	68 %
h264ref	7.5	2.9	1535	210	1503	176	16 %
astar	7.7	4.0	518	56	551	40	29 %
xalancbmk	7.4	4.4	551	11	2676	4.2	63 %
Average	8.0	4.2					31 %

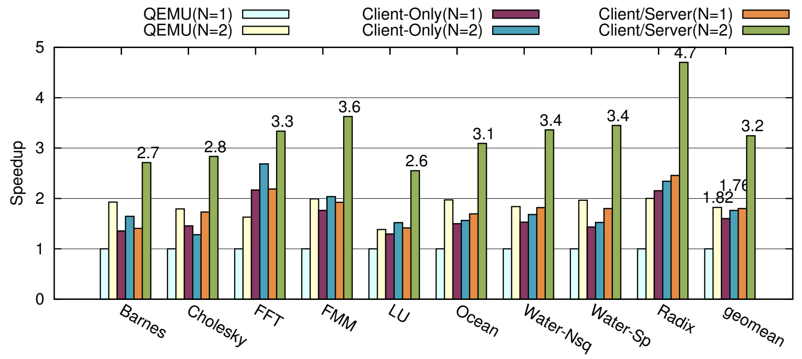


Figure 7: SPLASH-2 results of x86-32 to ARM emulation with large data sets. N represents the number of emulation thread. Code cache size: Unlimited.

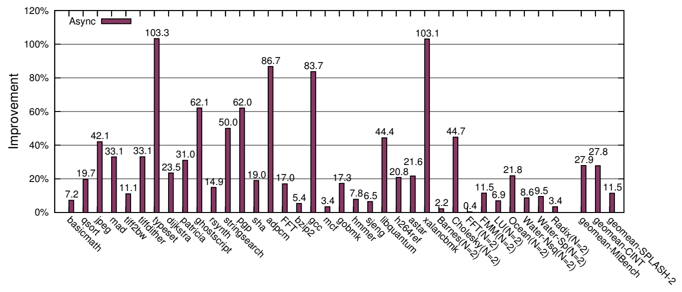


Figure 8: Comparison of asynchronous and synchronous translation. Code cache size: Unlimited. (*baseline = Synchronous translation*)

three threads are running at the same time). On average, QEMU achieves 1.82X speedup with two emulation threads, where only 10% improvement is achieved with Client-Only.

The performance of Client/Server also scales well with two emulation threads. Thanks to the powerful server for conducting the aggressive optimization, the emulation threads can fully utilize the processor resources without incurring any interference. The performance of the Client/Server mode is 1.78X and 1.84X faster than QEMU and Client-Only, respectively, when two guest threads are emulated. From the study of this result, emulating single-thread application on a single-core embedded platform can also achieve low overhead and high performance with our client/server DBT framework.

3.4. *Comparison of Asynchronous and Synchronous Translation*

To evaluate the effectiveness of asynchronous translation, we compare it with synchronous translation. Figure 8 illustrates the performance gain of the asynchronous translation mode compared to the synchronous translation mode for a subset of MiBench, SPEC CINT2006 and SPLASH-2 benchmarks. As opposed to that in the asynchronous translation, when the thin client in the synchronous mode sends an optimization request to the remote server, it will suspend and resume its execution until the optimized code is sent back from the server and emitted to the optimized code cache.

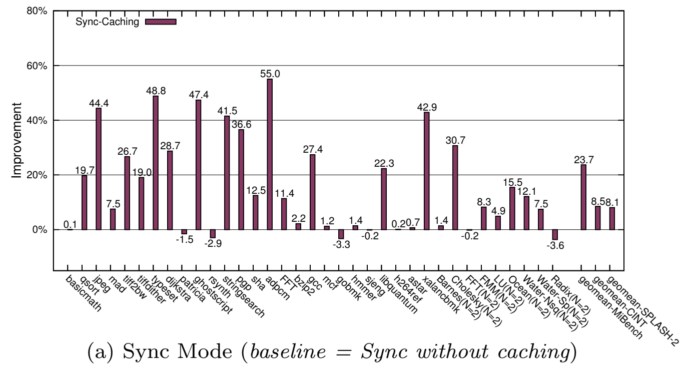
In Figure 8, the result shows that the many benchmarks of MiBench are sensitive to synchronous translation. Even though the optimization conducted by the powerful server is efficient, the waiting time is still not tolerable for such short-running programs. For several benchmarks in MiBench, the performance with asynchronous translation can reach at least 50% improvement. The benchmarks, `libquantuam` and `xalancbmk` in SPEC CINT2006 and `Cholesky` in SPLASH-2, also have similar behavior because they are all short-running programs. As for `gcc`, which has a lot of code regions for optimization, asynchronous translation also shows significant performance gain (about 84%) compared to synchronous translation. The performance of the asynchronous translation achieves the geomean of 27.2%, 27.8% and 11.5% improvement over synchronous translation for MiBench, SPEC CINT2006 and SPLASH-2 benchmark suite, respectively.

3.5. *Impact of Persistent Code Cache*

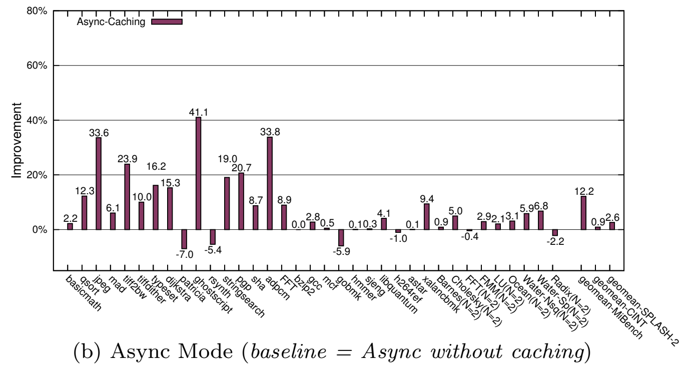
In this section, we evaluate two scenarios. First, one client emulates a program and another client emulates the same program a while later. Second, we force the client to flush its local code cache after the code cache is full. These two cases are used to evaluate the impact of the server's persistent code cache if the translated code can be re-used.

3.5.1. *Continuous Emulation of the Same Program*

Figure 9 presents the performance results of the first case. We also evaluate this case using both asynchronous and synchronous translation modes whose results are shown in Figure 9(a) and 9(b). As the first client starts the emulation, no pre-translated/optimized code is cached, and the server needs to perform translation/optimization for the first client. The result of this first client is set as the baseline performance. When the second client requests the same program that has been translated/optimized by the first client, the optimized code is reused and the improvement is illustrated in Figure 9. As the results show, the performance is improved with the server-side caching because the response time is shortened for all optimization requests. The impact of caching is more significant



(a) Sync Mode (*baseline = Sync without caching*)



(b) Async Mode (*baseline = Async without caching*)

Figure 9: Impact of persistent code cache with continuous emulation of the same program.

Table 3: Measures of code cache flushing. Unit of code size: KBytes.

Benchmark	Code Size	#Flush	Benchmark	Code Size	#Flush
CINT2006			SPLASH-2		
bzip2	1050	12	Barnes	918	2
gcc	14984	59	Cholesky	1671	22
mcf	783	2	FFT	735	2
gobmk	5637	30	FMM	1227	6
hmmer	1155	2	LU	751	4
sjeng	1246	7	Ocean	1314	5
libquantum	724	2	Water-Nsq	1076	2
h264ref	2978	49	Water-Sp	1098	2
astar	1242	2	Radix	749	2
xalancbmk	7253	2			

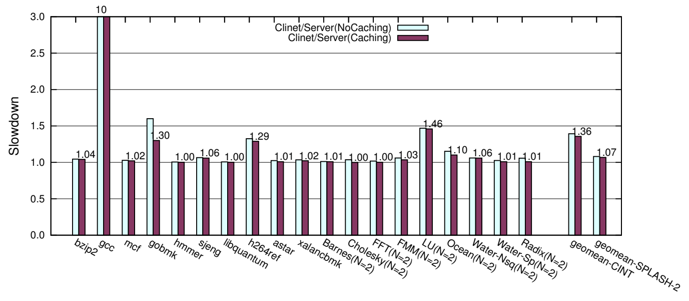


Figure 10: Impact of persistent code cache with code cache size as half translated code size. (*baseline = Client/Server with unlimited cache size*)

for the short-running benchmarks such as MiBench and `libquantum` and `xalancbmk`.

The impact of caching is even more noticeable with the synchronous translation mode (Figure 9(a)) because the waiting time for translation/optimization can be reduced with the persistent code already cached on the server. Many benchmarks of MiBench are improved by at least 30%, and benchmark `gcc`, `libquantum` and `xalancbmk`, are improved by about 27%, 22% and 43% in the synchronous mode, respectively, where an average of 24%, 8.5% and 8.1% of performance gain is achieved for MiBench, SPEC CINT2006 and SPLASH-2.

As for the asynchronous mode, the client does not wait for the server-side optimization. It also continues the execution with the help of its lightweight translator. Therefore, the benefit from persistent code caching is not significant for the client. An average of 12.2% improvement is achieved for MiBench and only 0.9% and 2.6% of improvement on average is observed for SPEC CINT2006 and SPLASH-2 benchmarks.

Several benchmarks, such as `gobmk`, show a negative impact from persistent code caching. The reason is that there are 7 different input data sets for `gobmk`. When the emulation is done with the first input data set, the server keeps some optimized code in its persistent code cache. As the emulation continues with the rest of the input sets, the server sends the cached code back to the client. However, the cached code that is good for the previous input data sets may not be suitable for the current input data. It causes many early exits from the traces [9], and results in some performance loss.

3.5.2. Code Cache Flushing

In the second case, the code cache size on the client is set to be smaller than the size needed for the emulated benchmarks. When the code cache is full, the client is forced to flush both the code cache and the optimized code cache. In this experiment, we use the single-thread SPEC CINT2006 and the multi-thread SPLASH-2 benchmarks for our performance studies. We set the code cache size to half of the total translated code size. For the testing benchmarks, the lightweight translator in our DBT system always fills up the code cache earlier than the optimized code cache. Thus, as the code cache is full, both code cache and the optimized code cache are flushed at the same time. The total size of the translated host code from the thin translator and the number of code cache flushes is listed in Table 3.

We compare the performance of Client/Server with and without server-side persistent code caching. Figure 10 shows the slowdown factor compared with Client/Server mode with unlimited code cache size (i.e. no flushing is required).

From column 3 and 6 in Table 3, most benchmarks, such as `mcf`, `hmmer` etc. in SPEC CINT2006 and `Barnes` etc. in SPLASH-2, only incur a few flushes and not much performance loss is observed in Figure 10 for these benchmarks with or without server-side caching. As for `gobmk` and `h264ref`, which have more code cache flushing frequency, performance degradation for these two benchmarks

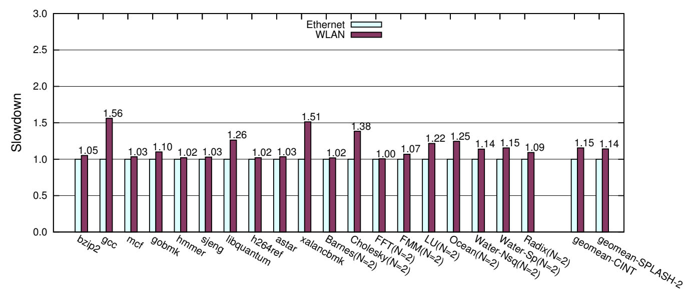


Figure 11: Comparison of x86-32 to ARM emulation for SPEC CINT2006 and SPLASH-2 benchmarks using Ethernet and WLAN. Code cache size: Unlimited.

are observed. Without server-side caching, although it can still get help from the powerful server to re-translate the code, about 60% and 33% performance are lost compared with Client/Server mode of unlimited code cache size, respectively. With server-side caching, the slowdown of these two benchmarks reduces to 30% and 28%, respectively. As for `gcc`, its code cache size is too small to accommodate its working sets, so frequent code cache flushes would occur. The optimized code is not effectively utilized between two flushes, and sending frequent optimization requests to the server incurs a lot of overhead on the client even if the server has cached the optimized code. Thus, about 10X slowdown is observed.

3.6. Comparison of Different Network Infrastructures

Figure 11 presents the performance comparison of Client/Server mode with asynchronous translation using Ethernet and WLAN of 802.11g for SPEC CINT2006 and SPLASH-2 benchmarks. For the WLAN settings, the thin client connects to a wireless access point which connects to the server through the local area network. For SPEC CINT2006 benchmarks, the longer latency of WLAN only cause slight performance degradation over Ethernet, such as `bzip2`, `mcf` and `hmmer`, etc. `libquantum` and `xalancbmk` incur more performance loss because they are relatively short-running programs which are more sensitive to the network latency. `gcc` incurs about 56% performance degradation. This is because it has a large amount of code regions for optimization and the longer latency of WLAN causes the execution thread to miss the best timing to utilize the optimized code. The results also show that the benchmarks of SPLASH-2 running two emulation threads are also sensitive to the communication latency of WLAN. On average, the performance degrades about 15% and 14% with WLAN over Ethernet for SPEC CINT2006 and SPLASH-2 benchmark suite, respectively.

4. Related Work

Dynamic binary translation is widely used in many applications: transparent performance optimization [18, 2, 19], runtime analysis [20, 21, 22] and cross-ISA emulation [3, 1]. With the advances in mobile devices, many have explored the technology of virtualization on embedded systems.

Xen on ARM [23] and KVM for ARM [24] provided software solutions to enable full system virtualization before hardware virtualization technology becomes available on ARM platforms. Those software approaches use specialized VMM systems and need to modify the guest VMs in order to run on the ARM processors. Such approaches lose transparency. They also require the guest VMs and the host machine be the same architecture. In contrast, we focus on user-mode emulation. Our approach is based on dynamic binary translation that allows cross-ISA emulation without any modification to the emulated programs.

Guha et al. [25] and Baiocchi et al. [26] attempted to mitigate the memory pressure for embedded system by reducing the memory footprint in the code cache. [25] discovered that many codes in exit stubs are used to keep track of the branches of a trace. [26] also found that DBT introduces a large amount of meta-code in the code cache for its own purposes. They proposed techniques to re-arrange the meta-codes so that more space can be reclaimed for the application codes.

Baiocchi et al. [27] studied the issues of code cache management for dynamic binary translators targeting on embedded devices with a three-level cache hierarchy, ScratchPad memory, SDRAM and external Flash memories, the authors proposed policies to fetch or evict translated code among these three storage levels based on their size and access latency. Instead of using external memory, our work uses remote server's memory space as the extension to the thin client's code cache.

DistriBit [28, 29] also proposed a dynamic binary translation system in a client/server environment. The thin client in their system only consists of an execution cache and the engine. It does not have a translator. The whole translation process is sent to a remote server. The authors proposed a cache management scheme in which the decisions of a client are totally guided by the server in order to reduce the overhead on the thin client. Since only the server has the translation ability in their DBT system, the system would stop functioning if the translation service is not available. The emulation also needs to be suspended until the client receives the response from the server. Instead of using only one translator, our DBT system keeps a thin translator on the client. Thus, our system can tolerate network disruption or outage and still keep low overhead. Moreover, the performance of an emulation can be improved by asynchronous translation with our proposed two-translator approach where the communication latency can be hidden.

Zhou et al. [30] proposed a framework with a code server. Thin clients download the execution code to its code cache on-demand. In their work, the application code has to be pre-installed on the server before the clients can connect to it. In contrast, our DBT system does not require such pre-installation of application code. The server only needs to provide the translation/optimization service, and the client sends requests to the server at runtime.

5. Conclusion

In this work, we developed a distributed DBT system based on a client/server model. We proposed a DBT system that consists of two dynamic binary translators: an *aggressive dynamic binary translator/optimizer* on the servervto service the translation/optimization requests from thin clients, andva *thin DBT* on each thin client that executes lightweight binary translation and basic emulation functions for each thin client.

Such a two-translator client/server approach allows the process of translation and code optimization in a distributed DBT system to tolerate network disruptions. The client can perform stand-alone translation and emulation when network connection or translation service on a remote server becomes unavailable. While in a normal operation mode, the DBT system can take advantage of the compute resources available on the server to perform more aggressive dynamic translation and optimization there.

With such a two-translator client/server approach, we also successfully off-load the DBT overhead of thin clients to the server, and achieve a significant performance improvement. Experimental results show that the DBT in client/server model could achieve 37% and 17% improvement over that of non-client/server model for x86/32-to-ARM emulation using MiBench and SPEC CINT2006 benchmarks with test inputs, respectively, and 84% improvement using SPLASH-2 benchmarks running two emulation threads.

6. Acknowledgment

This work is supported in part by the Ministry of Science and Technology of Taiwan under grant number NSC102-2221-E-001-034-MY3.

[1] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, J. Yates, FX!32: A profile-directed binary translator, IEEE Micro 18 (2) (1998) 56–64.

[2] R. J. Hookway, M. A. Herdeg, DIGITAL FX!32: Combining emulation and binary translation, Digital Technical Journal 9 (1) (1997) 3–12.

[3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, Y. Zemach, IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003.

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

[5] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004, pp. 75–88.

[6] Tiny code generator, http://wiki.qemu.org/Documentation/TCG.

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

[8] E. Duesterwald, V. Bala, Software profiling for hot path prediction: Less is more, in: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000, pp. 202–211.

[9] D. Hiniker, K. Hazelwood, M. D. Smith, Improving region selection in dynamic optimization systems, in: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005, pp. 141–154.

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

[11] D. M. Davis, K. Hazelwood, Improving region selection through loop completion, in: ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments, 2011.

[12] K. Hazelwood, G. Lueck, R. Cohn, Scalable support for multithreaded applications on dynamic binary instrumentation systems, in: Proceedings of the 2009 International Symposium on Memory Management, 2009, pp. 20–29.

[13] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, M. L. Soffa, Overhead reduction techniques for software dynamic translation, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, pp. 200–207.

[14] Http://pandaboard.org/node/300/#Panda.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization, 2001, pp. 3–14.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995, pp. 24–36.

[17] The performance monitoring interface for linux, https://perf.wiki.kernel.org/.

[18] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, 2000, pp. 1–12.

[19] S. Sridhar, J. S. Shapiro, E. Northup, P. P. Bungale, HDTrans: an open source, low-level dynamic instrumentation system, in: Proceedings of the 2nd International Conference on Virtual Execution Environments, 2006, pp. 175–185.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005.

[21] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 89–100.

[22] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, Y. Wu, LIFT: A low-overhead practical information flow tracking system for detecting security attacks, in: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006, pp. 135–148.

[23] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, C.-R. Kim, Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones, in: 5th IEEE Consumer Communications and Networking Conference, 2008.

[24] C. Dall, J. Nieh, Kvm for arm, in: Proceedings of the 12th Annual Linux Symposium, 2010.

[25] A. Guha, K. Hazelwood, M. L. Soffa, Reducing exit stub memory consumption in code caches, in: Proceedings of the 2nd international conference on High performance embedded architectures and compilers, 2007, pp. 87–101.

[26] J. A. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser, Reducing pressure in bounded dbt code caches, in: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems, 2008.

[27] J. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser, J. Misurda, Fragment cache management for dynamic binary translators in embedded systems with scratchpad, in: Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems, 2007, pp. 75–84.

[28] H. Guan, Y. Yang, K. Chen, Y. Ge, L. Liu, Y. Chen, Distribit: a distributed dynamic binary translator system for thin client computing, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 684–691.

[29] L. Ling, C. Chao, S. Tingtao, L. Alei, G. Haibing, Distribit: A distributed dynamic binary execution engine, in: Proceedings of the third Asia International Conference on Modelling & Simulation, 2009.

[30] S. Zhou, B. R. Childers, M. L. Soffa, Planning for code buffer management in distributed virtual execution environments, in: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, 2005, pp. 100–109.