TR-IIS-12-002

A Dynamic Binary Translation System in a Client/Server Environment

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

Apr. 23, 2012 || Technical Report No. TR-IIS-12-002
http://www.iis.sinica.edu.tw/page/library/TechReport/tr2012/tr12.html

A Dynamic Binary Translation System in a Client/Server Environment

Ding-Yong Hong
Department of Computer Science
National Tsing Hua University
Institute of Information Science
Academia Sinica, Taiwan
dyhong@sslab.cs.nthu.edu.tw

Chun-Chen Hsu
Department of Computer Science
and Information Engineering
National Taiwan University, Taiwan
d95006@csie.ntu.edu.tw

Chao-Rui Chang
Department of Computer Science
and Information Engineering
National Taiwan University, Taiwan
crchang@iis.sinica.edu.tw

Jan-Jan Wu
Institute of Information Science
Academia Sinica, Taiwan
wuj@iis.sinica.edu.tw

Pen-Chung Yew
Department of Computer Science
University of Minnesota, USA
yew@cs.umn.edu

Wei-Chung Hsu
Department of Computer Science
National Chiao Tung University,
Taiwan
hsu@cs.nctu.edu.tw

Pangfeng Liu
Department of Computer Science
and Information Engineering
National Taiwan University, Taiwan
pangfeng@csie.ntu.edu.tw

Chien-Min Wang
Institute of Information Science
Academia Sinica, Taiwan
cmwang@iis.sinica.edu.tw

Yeh-Ching Chung
Department of Computer Science
National Tsing Hua University,
Taiwan
ychung@cs.nthu.edu.tw

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 enable 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 reachable to thin mobile clients. Those network servers are usually equipped with a substantial amount of resources. This opens 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 the client/server model. We proposed a DBT system that consists of two dynamic binary translators. An aggressive dynamic binary translator/optimizer to serve the translation/optimization requests from thin clients are run on the server. A thin DBT that executes light-weight binary translation and basic emulation functions is run on each thin client. With such a two-translator client/server approach, we successfully off-load the DBT overhead of the thin client to the server and achieve significant performance improvement over the non-client/server model. Experimental results show that the DBT of the client/server model could achieve 14% speedup over that of non-client/server model for x86-32 to ARM emulation using SPEC CINT2006 benchmarks with test inputs and are only 3.4X and 2.2X slower than the native execution with test and reference inputs, respectively, as opposed to 7.1X and 5.1X slow-down on QEMU.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; D.2.4 [**Distributed Systems**]: Client/server; D.3.4 [**Processors**]: Code generation; D.3.4 [**Processors**]: Run-time environments

General Terms

Design, Performance

Keywords

Systems, networks, and architectures for high-end computing, Virtualization of machines, networks, and storage, Resource management, energy and cost minimizations

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. 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 [6] and IA-32 EL [4] that have enabled IA-32 applications to be executed on Alpha and Itanium machines, respectively. With rapid advances in mobile computing, multi-core processors and expanded memory resources are being made available in new mobile devices. This trend will enable 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 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 increasing network accessibility and bandwidth in various environments, e.g. 4G networks or wireless LAN in public areas, an increasing number of network servers are becoming accessible to thin mobile 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 executes light-weight binary translation and basic emulation functions on each thin client.

In our DBT system, we use QEMU [5] 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 [15], also a popular compiler with sophisticated compiler optimizations, on the server side. With such a two-translator client/server approach, we successfully off-load the DBT overhead of the thin 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 a client/server-based DBT framework whose two-translator design can tolerate network disruption and outage for translation/optimization services on a server. Moreover, we showed that the translation overhead and network latency could be successfully hidden by an asynchronous communication scheme.
- Experimental results show that the DBT of the client/server model can achieve 14% speedup over that of non-client/server model for x86-to-ARM emulation using SPEC CINT2006 benchmarks with test inputs, and are only 3.4X and 2.2X slower than the native execution with test and reference inputs, respectively, as opposed to 7.1X and 5.1X slow-down on QEMU.

The rest of this paper is organized as follows: Section 2 provides the details of our two-translator client/server-based DBT system. Section 3 evaluates the effectiveness of our DBT framework. 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 such a distributed DBT system. Then we elaborate on the implementation details of our proposed client/server-based DBT framework.

2.1 Design Issues

A typical DBT system consists of three main components: an emulation engine, a translator, and a code cache. As the 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. It translates the guest binary into server ISA binary, runs the translated code directly on 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 translated code between client and server, the system resources, such as heap and stack, must be carefully monitored so that the emulation can be migrated correctly between client and server. System resource monitoring at the binary level and the migration method between client and server are challenging issues that deserve further studies and are outside of the scope

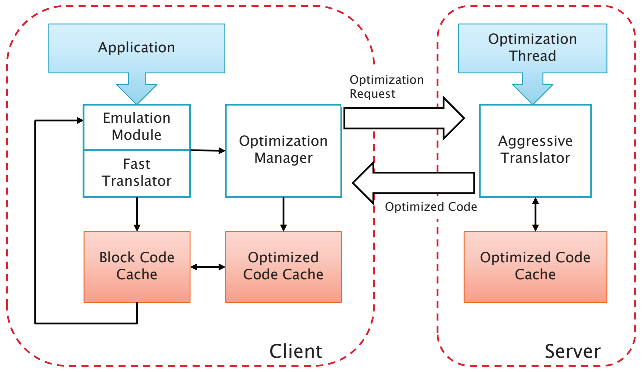


Figure 1: The architecture of the distributed DBT system.

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 standalone 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 considerations, we proposed a DBT system that consists of two dynamic binary translators: (1) a thin DBT that performs light-weight 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 Fig. 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 Fig. 1). In order to achieve low translation overhead, the thin client uses a light-weight 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 may be poor, the thin client can perform its own emulation without the help from the server if the network becomes unavailable.

When the light-weight DBT detects a section of code worthy of further optimization, the optimization manager issues a request to the server for such service. After it receives the optimized code back from the server, it needs to perform relocation for those relative addresses and commits the optimized code to the code cache. The entry of the old code is patched with a branch to the optimized code so that the subsequent execution will be on the optimized, better-quality host code.

Note that since the address to place the optimized code is not known to the server, the server cannot perform relocation of relative addresses (chaining code region or trampoline, for instance) for the client. To assist the client with relocating addresses, the server attaches the patching rules with the optimized code and wrap them in the message to the client so the client can easily patch the addresses.

In our implementation, we use the Tiny Code Generator (TCG) in QEMU, a well-known retargetable DBT system that supports both full-system virtualization and process-level emulation, as our light-weight 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 light-weight translator is to perform the translation as quickly as possible, so we could switch the execution to the translated code in the code cache 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.

The optimization manager running on another thread is responsible for network communication and the relocation of received codes. Since both require low overhead and the optimization manager is usually at the idle state, the execution thread is not interfered by the optimization manager.

Other implementation details for the client are described in the following.

Sending all sections of codes in a program to the server for optimization is impractical because this will result in substantial amount of network communication and not all optimized code can achieve performance gain. For example, optimizing the *cold* code regions might not be beneficial for the performance. To overcome this problem, we must ensure the performance gain brought by optimization can amortize the communication cost and the translation overhead on the server. Our strategy is that the translation of *cold* code regions is conducted by the light-weight DBT on the thin client. Only the *hot* code regions that form repeated execution are sent to the server for optimization. In this work, we target the cyclic execution path (e.g. loop or recursive function) as the optimization candidates since the optimized code is likely to be highly-utilized. The repetition detection scheme, *Next Executing Tail* (NET) [8], is applied to detect all possible cyclic execution paths which are formatted as the traces. Through this design, the number of network communication can be significantly reduced.

Unlike many existing approaches, the client in our framework does not attach any address information in the optimization request regarding where the optimized code will be placed. The reason is that, to provide such address information, the client needs to lock and reserve sufficient space for a region of memory in the optimized code cache from sending the request until receiving the optimized code. There are three drawbacks in such approach: (1) the client does not have information on how much space to reserve before optimization; (2) a long waiting time is incurred by the locking; (3) the next optimization request will be blocked until the previous optimization is completed. Instead, our approach allows contiguous sending of optimization requests without blocking, and the serialization only occurs when the optimized code is copied into the optimized code cache.

2.2.2 *Server*

The purpose of the server is to provide the translation/optimization service for the thin clients. The more powerful server allows the translator to perform more CPU-intensive optimizations and analyses that often require a large amount of memory resources, and thus 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 code cache on the server keeps all translated and optimized codes for the thin clients throughout their emulation lifetime. The thin client can take advantage of this persistent code cache as it flushes its code cache or 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 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.

For the more aggressive translator/optimizer on the server, we use an enhanced LLVM compiler because it consists of a rich set of aggressive optimization passes and a just-in-time 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 [14]. This approach simplifies the translation process on the server side tremendously because TCG IR consists of only about 142 different operation codes instead of a much larger set in 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.

2.3 Asynchronous Translation

The two translators in our distributed DBT system, one on the client and one on the server, are independent and working concurrently. When a thin client sends an optimization request to the server, its light-weight 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. The advantage of such an asynchronous translation model is threefold, (1) the network latency can be hidden, (2) the translation overhead incurred by the aggressive translator is also hidden, and (3) the thin client can continue the emulation process while the server performs further optimization on its request.

In summary, our two-translator design allows the thin client to perform the emulation independently and concurrently. It allows our distributed DBT system to tolerate network disruption or outage during the service by the server. Our asynchronous translation design can also hide network latency and offload the translation overhead from the client.

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	

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 the performance evaluation of the server/client-based DBT framework. Detailed analysis of overall performance is also provided to verify the effectiveness of the proposed framework.

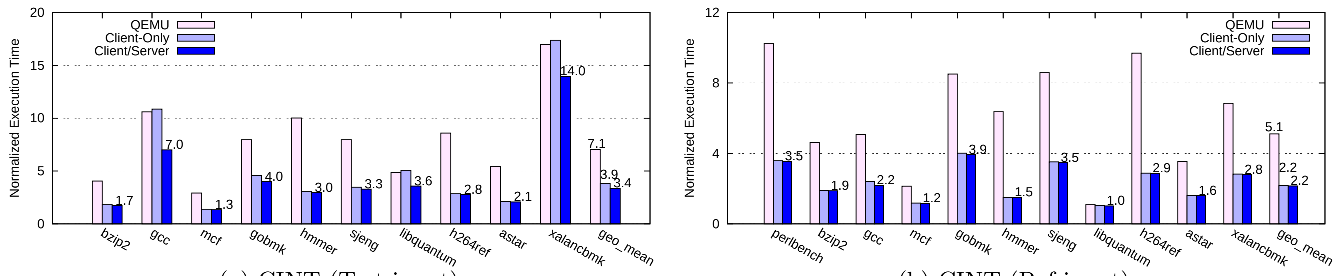
3.1 Experimental Setup

All performance evaluation is conducted using an Intel quad-core machine as the server and an ARM PandaBoard embedded platform [20] as the thin client. The detailed configuration is listed in Table 1. The experiment results on the SPEC CINT2006 benchmark suite are collected with both test and reference inputs. LLVM version 2.8 is used as the aggressive translator on the server side and the default optimization level (-O2) is used for the JIT compilation. The client/server communication is through TCP/IP protocol.

All benchmarks are compiled with GCC 4.4.2 for the emulated x86-32 executables. We compare the results to the native runs whose executables are compiled to GCC 4.5.2 on the ARM host. The compiler optimization flags used for each architecture are listed in Table 1. Three different configurations are used to evaluate the effectiveness of the framework:

- **QEMU**, which is the vanilla QEMU version 0.13 with the TCG fast translator.
- **Client-Only**, which runs both fast translator and aggressive translator on the thin client. Each translator runs on one thread.
- **Client/Server**, which runs fast translator on the thin client and aggressive translator on the server.

The QEMU configuration is used to demonstrate the performance of the thin translator itself, and is also the baseline performance when the network is unavailable. The evaluation in the following experiments is based on the 100Mbps Ethernet and WLAN of 802.11g is used in Section 3.4 for comparison.



(a) CINT (Test input)

(b) CINT (Ref input)

Figure 2: CINT2006 results of x86-32 to ARM emulation with test and reference input. Cache size: Unlimited.

Table 2: Measures of x86-32 to ARM emulation for CINT2006 benchmarks with test inputs. Unit of time: second. Cache size: Unlimited.

			Native	Client-Only			Client/Server			
Benchmark	# Blocks	# Traces	Total Time	Total Time	Trace Trans Time (C)	ICount (10^9)	Total Time	Trace Trans Time (S)	ICount (10^9)	Reduction Rate
bzip2	7170	1022	62	113	21	73	108	2.4	45	37%
gcc	64604	17680	8	90	87	67	58	26.6	25	62%
mcf	3156	319	28	39	5	17	37	.6	12	31%
gobmk	74477	23732	112	512	228	273	449	37.3	157	43%
hmmer	4633	303	20	60	6	66	59	1.0	57	13%
sjeng	4967	1284	29	100	16	63	97	2.4	40	36%
libquantum	2797	177	.5	2.4	2	4	1.8	.4	1	60%
h264ref	9046	1577	118	336	34	256	327	4.3	207	19%
astar	4812	605	42	89	8	65	87	1.1	45	31%
xalancbmk	30139	2562	.6	11	10	11	9	8.1	4	60%

3.2 Performance of the Server/Client Framework

Fig. 2 illustrates the overall performance results of x86-32 to ARM emulations over the native runs for SPEC CINT2006 benchmarks with test and reference inputs. The Y-axis is the normalized execution time over native execution time. The performance of `perlbench` and `omnetpp` in Fig. 2(a), and `omnetpp` in Fig. 2(b) are not listed because both QEMU and our framework failed to emulate these benchmarks. The size of code cache and optimized code cache on the client is set to unlimited in this experiment in order to exclude the overhead of translated code evictions.

Fig. 2(a) presents the results with test inputs. In Fig. 2(a), the slowdown factors of QEMU over native execution range from 2.9X to 17X and the geometric mean is 7.1X. With the optimization from the aggressive translator, significant improvement can be observed from the results of Client-Only whose geometric mean is 3.9X. Three benchmarks, `gcc`, `libquantum`, and `xalancbmk`, have slight performance degradation with Client-Only over QEMU. The reason is that the execution thread continues its execution while the aggressive optimizer running on another thread consumes much time to complete. Thus, the optimized code may miss the best timing to be utilized and the interference incurred by the optimization thread causes the performance degradation.

As for the Client/Server mode, the performance of benchmarks, `gcc`, `gobmk`, `libquantum` and `xalancbmk`, has the most significant improvement over the Client-Only mode. The other benchmarks have slight improvement. Table 2 lists the detailed information of total execution time, number of traces translated, and the translation time of traces at the client side and the server side. For `gcc` and `gobmk`, these two benchmarks have relatively much more code regions to be optimized to traces (column 3 in Table 2). Therefore, it is beneficial to process such large amount of optimizations on the powerful server instead of the thin client. Client/Server achieves about 34% and 12% performance improvement over Client-Only for `gcc` and `gobmk`, respectively. For `libquantum` and `xalancbmk`, the completion time of the translated traces with powerful server is earlier than that processed by the client. Furthermore, Client/Server can catch the timing for the execution thread to utilize the optimized code, 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 3.4X slower than the native execution in average.

In Fig. 2(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 translated codes are highly utilized with reference inputs and the translation time only accounts for a small portion of the total execution time. Once the optimized codes are emitted in the optimized code cache, the thin client can benefit from the better codes compared with the poor code executed by QEMU. Since the Client/Server mode tends to eliminate the optimization overhead which is insignificant to the total execution, no significant performance gain is observed with Client/Server over Client-Only. The only exception is `gcc` where about 630 seconds are reduced (10% improvement)

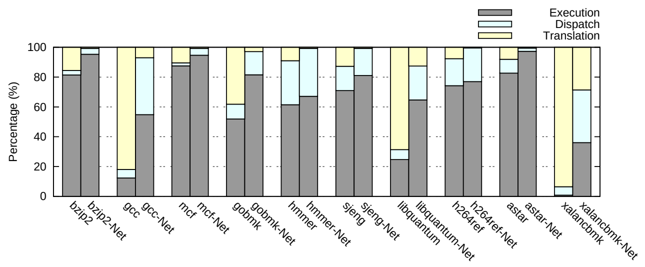


Figure 3: Comparison of breakdown of time of x86-32 to ARM emulation for CINT2006 with test inputs in the Client-Only and Client/Server mode.

Table 3: Measures of translation quality between QEMU and our DBT system. Unit of size: KBytes.

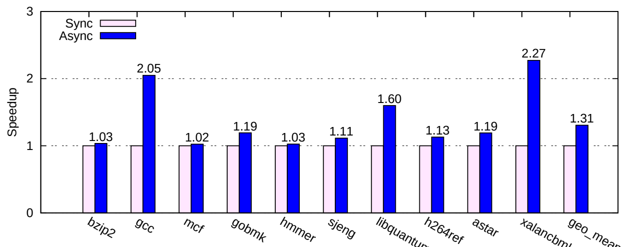
	QEMU			Client/Server		
Benchmark	Guest	Host	Expa.	Guest	Host	Expa.
bzip2	181	2050	11.36	117	439	3.74
gcc	1038	14586	14.15	856	5429	6.34
mcf	59	783	13.19	23	124	5.46
gobmk	1450	18396	12.69	1212	6699	5.53
hmmer	90	1149	12.80	23	149	6.41
sjeng	102	1232	12.07	78	393	5.02
libquantum	56	726	12.98	11	66	6.21
h264ref	306	2940	9.59	292	855	2.93
astar	104	1233	11.92	42	216	5.13
xalancbmk	550	7087	12.90	213	1247	5.87
average			12.36			5.26

with Client/Server and the average slowdown over native run is 2.2X.

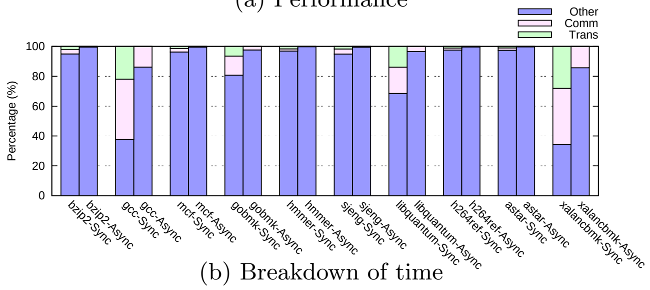
We use hardware performance counter and Linux Perf Event toolkit [17] with event instructions to calculate the total instruction executed on the thin client with test inputs. The values are listed in column 7 and 10 in Table 2 for the Client-Only and the Client/Server mode, respectively. The instruction count is derived from execution, light-weight and aggressive translations with Client-Only and from execution, light-weight translation and network processing with Client/Server. The instruction reduction rate is presented in column 11 where about 13% (`hmmer`) to 62% (`gcc`) instruction counts are reduced. The results show that the Client/Server mode achieves both performance gain and saving in power consumption because fewer instructions are executed with the Client/Server mode.

Fig. 3 compares the breakdown of time in the Client-Only and Client/Server mode in terms of the emulation components: *Execution*, *Dispatch* and *Translation*. We also use the hardware performance counter to sample the *Instruction Pointer* (IP) during execution to estimate the time spent in these three components. The benchmarks with '-Net' as the postfix in X-axis refer to the results of Client/Server. For Client-Only, *Translation* denotes the time for generating host code (from both light-weight and aggressive translation). For Client/Server, it denotes the time from light-weight translation plus the time for network transferring of optimization request/optimized code to/from the server, and the time for post-processing the received code. *Dispatch* is the time for QEMU core components, including block/trace table look up, and execution flow control between QEMU and the code cache. *Execution* is the time for executing translated host codes. As Fig. 3 shows, with the help of server to perform aggressive translation, the ratio of *Translation* is reduced dramatically with Client/Server over Client-Only. In particular, for `xalancbmk`, the one with the largest percentage of translation time, its ratio reduces by 77.8%; `gcc`, which generates the largest number of traces, benefits most from the client/server model achieving 91% reduction in translation time.

Table 3 compares the quality of the translated host codes generated by QEMU and our DBT system. *Guest* denotes the total size of original guest binary code disassembled; *Host* represents the total size of the translated host code. The expansion rate, *Host* divided by *Guest*, is also listed in the table. As shown in Table 3, QEMU translates each guest binary instruction to 10˜14 host instructions. With LLVM aggressive translator, our system can reduce the number of translated host instructions to 4˜6 and thus achieves the goal of generating better quality codes.



(a) Performance



(b) Breakdown of time

Figure 4: Comparison of asynchronous and synchronous translation for SPEC CINT2006 with test inputs. Cache size: Unlimited.

3.3 Comparison of Asynchronous and Synchronous Translation

To evaluate the effectiveness of asynchronous translation, we compared it with synchronous translation. We also set the size of caches on the client to unlimited in this experiment to eliminate the impact of code eviction.

Fig. 4(a) illustrates the performance speedup of asynchronous mode compared with synchronous mode for SPEC CINT2006 benchmarks with test inputs. The breakdown of time in terms of the communication time, *Comm*, the translation time at server, *Trans*, and the rest of time, *Other*, is presented in Fig. 4(b). Different to asynchronous mode, when the client in synchronous mode sends the optimization request to remote server, it will suspend and resume its execu-

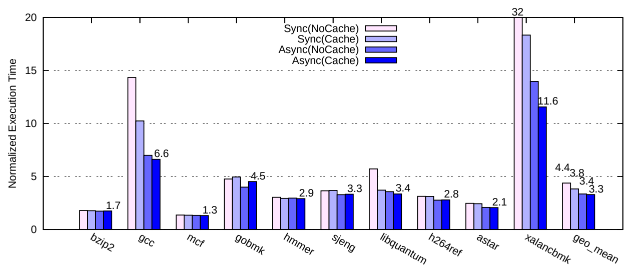


Figure 5: Impact of persistent code cache with continuous emulation with the same program.

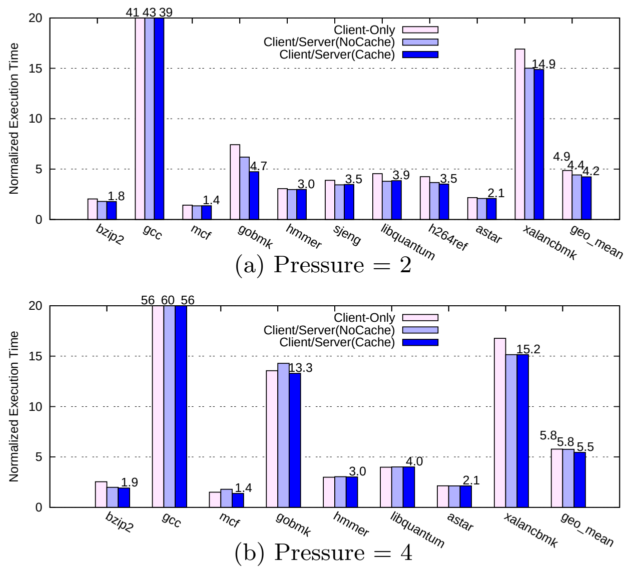


Figure 6: Comparison of CINT2006 benchmarks with test inputs with different code cache pressure.

tion until the optimized codes are sent back from the server and emitted in the optimized code cache. The communication time, *Comm*, in synchronous mode includes the time to send the optimization request message, network latency, and the time to receive and post-process the received codes. For asynchronous mode, *Comm* only consists of message send and receive, and the time to post-process the received codes. Since asynchronous translation does not wait for the aggressive translation on the server, *Trans* in this mode is zero.

In Fig. 4(b), the result shows that a significant proportion of the emulation in synchronous mode is for communication and translation, especially for the benchmarks, `gcc`, `gobmk`, `libquantum` and `xalancbmk`. By using asynchronous translation, we can see that the ratio of communication and translation time is reduced significantly, and thus results in about 31% improvement in average (shown in Fig. 4(a)) against synchronous translation for test inputs. As for reference inputs, no obvious improvement is observed because the translation time only constitutes a small portion of the overall emulation time.

3.4 Impact of Persistent Code Cache

In this experiment, we evaluate two scenarios: first, one client emulates a program and another client coming later emulates the same program; second, we add pressure to the client and force it to flush its local code caches. These two cases are used to evaluate the impact of the server's persistent code cache as the translated code can be re-used.

Fig. 5 presents the performance results of the first case. As the first client starts the emulation, no pre-compiled code is cached and the server needs to perform translation. The result of this first client is marked with *NoCache* in the figure. For the second client requesting the same program which has been compiled for previous client, the optimized code is reused and the result is marked with *Cache*. We also compare this case with synchronous and asynchronous mode. As the result shows, the performance is improved with the server-side caching because the response time is shortened for the optimization request. The impact of caching is more noticeable with synchronous mode because the waiting time for one of the clients can be reduced because the code is already cached on the server. Benchmark `gcc`, `libquantum` and `xalancbmk`, are improved by about 40%, 54% and 73% in synchronous mode, respectively, where in average 14% performance gain is achieved. As for the asynchronous mode, the client does no waiting for optimization. Furthermore, it continues the execution with the help from the light-weight translator. Therefore, the benefit from caching is not significant for the client: only 2% improvement in average.

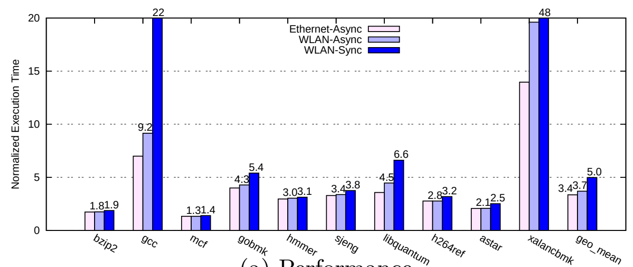
`gobmk` is the only benchmark that has negative impact with caching. The reason is that there are 7 different input data in this benchmark. As the emulation starts with the first input data, the server keeps some optimized code in its persistent code cache. As the emulation restarts with the succeeding inputs, the server will send the cached code back to the client if available. However, this code for previous input data may not be suitable for current input data, which in turn results in early exit of trace [11], and thus causes performance loss.

In the second case, the code cache size on the client is set to be fewer than the total size of the translated host code for the emulated benchmarks. As the code cache is full, the client is forced to flush both the code cache and the optimized code cache. The total size of the translated host code from the light-weight and aggressive translator is listed in column 2 and 5 of Table 3, respectively. In this case, we set the pressure to 2 where the code cache size is half of the total translated code size and 4 for a quarter. For CINT2006 benchmarks with test inputs, the light-weight translator in our DBT system always full-fills the code cache earlier than the optimized code cache, thus, changing the pressure of the code cache is enough to cause the optimized code cache to flush at the same time. The amount of code cache flushes is listed in Table 4. Since we failed to emulate benchmark `sjeng` and `h264ref` under pressure 4, the result of them is not shown.

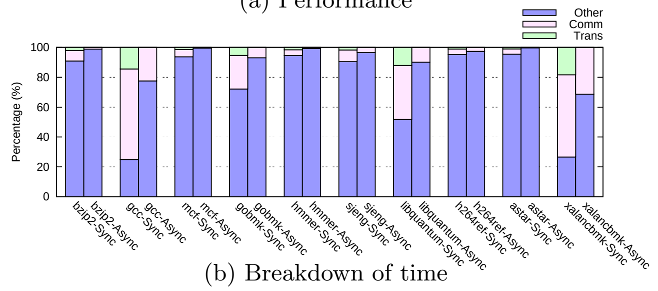
We compare the performance of Client-Only and Client/Server with or without server-side caching. The performance result is shown in Fig. 6. Under pressure 2, most benchmarks, e.g. `mcf`, `hmmer` etc., only incur a few flushes (row 2 in Table 4) and not much performance loss is observed with these benchmarks for both Client-Only and Client/Server modes. For `bzip2` and `gobmk`, Client/Server with caching outperforms Client-Only with a factor of 15% and 57%, re-

Table 4: The number of code cache flush under different pressures.

Benchmark	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquantum	h264ref	astar	xalancbmk
Pressure=2, # Flush	12	59	2	29	3	7	2	49	2	2
Pressure=4, # Flush	59	201	6	709	7	–	6	–	6	5



(a) Performance



(b) Breakdown of time

Figure 7: Comparison of Ethernet and WLAN for SPEC CINT2006 with test inputs. Cache size: Unlimited.

spectively, compared to only 4% and 14% with unlimited code cache size. The reason is that the aggressive translation will be redone by the client itself as the local caches are flushed with Client-Only mode. For Client/Server with caching, it can benefit from the cached codes on the server. Thus, the performance gap between Client-Only and Client/Server increases as code caches are flushed. For Client/Server without caching, although it cannot benefit from server-side caching, it can still get help from the server for retranslation of code and therefore the performance is better than Client-Only.

Under pressure 4, most benchmarks still have few code cache flushes and the performance results are similar to those of pressure 2. For `bzip2`, Client/Server with caching outperforms Client-Only with a factor of 32%. For `gcc` and `gobmk`, the performance gap between Client-Only and Client/Server is not as significant compared with pressure 2. The reason is that: first, these two benchmarks tend to generate large number of optimization requests and second, the code cache size is too small to accomodate the benchmark's working sets, so huge amount of code cache flushes is incurred (201 and 709 for `gcc` and `gobmk`, respectively). Therefore, the optimized code is not fully utilized during two flush intervals and the frequent sending of optimization requests to the server cause no gain but a lot of penalty on the client even if the server can cache the optimized code.

3.5 Comparison of Different Network Infrastructure

Fig. 7(a) presents the performance of Ethernet and WLAN with synchronous and asynchronous translation. Compared with Ethernet, the performance with WLAN and synchronous mode is worse than Ethernet because of longer network latency. The long latency can cause significant performance degradation on programs that generate large number of optimization requests because it may cause the client to miss the best timing to utilize the optimized code generated by the server. Such result caused by long network latency can be observed from the benchmarks: `gcc`, `gobmk`, `libquantum` and `xalancbmk`. With asynchronous translation, the long latency is hidden and the performance of these four benchmarks has dramatic improvement. Especially for `gcc`, the slowdown factor over native execution drops from 22X to 9.2X, and it drops from 48X to 19X for `xalancbmk`. The geometric mean is only 3.7X compared with 3.4X for 100Mbps Ethernet.

Fig. 7(b) illustrates the breakdown of time of WLAN with synchronous and asynchronous mode. We can see that, compared with 100Mbps Ethernet (Fig. fig:async(b)), wireless communication results in longer communication time (the *Comm* component) for all benchmarks. Similar to the result in Fig. 4(b), the benchmarks, `gcc`, `gobmk`, `libquantum` and `xalancbmk` benefits most from asynchronous translation, resulting in about 35% improvement in average against synchronous translation for test inputs.

4. RELATED WORK

Dynamic binary translation is widely used for many purposes: transparent performance optimization [3, 12, 22], runtime analysis [18, 19, 21] and cross-ISA emulation [4, 6]. With the advances of mobile hardware, many researches have started to exploit the technology of virtualization on embedded systems.

Xen on ARM [13] and KVM for ARM [7] proposed software solutions to enable full system virtualization before the hardware virtualization technology comes to the ARM platforms. Their approaches have specialized VMM system and need to modify the source codes of the guest VMs in order to directly run the guest VMs on the ARM processors. Such approaches lose transparency and require the guest VMs and the host machines to be of the same architecture. In contrast, we focus on user-mode emulation. Our approach is based on dynamic binary translation and allows cross-ISA emulation without any modification to the emulated programs.

Guha et al. [10] and Baiocchi et al. [1] attempted to mitigate the memory pressure for embedded system by reducing the memory footprint in the code cache. [10] discovered that many codes in exit stubs are consumed to keep track of the branches of a trace. [1] also found that DBT introduces large amount of meta-codes 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 actual application codes.

Baiocchi et al. [2] study the issue of code cache management for dynamic binary translators. Targeting on embedded devices with three-level cache storages, ScratchPad memory, SDRAM and external Flash memories. The authors proposed policies to fetch or evict translated codes 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 [9, 16] also proposed the dynamic binary translation system in a client/server environment. The thin client in their system only consists of the execution cache and engine, and does not have a translator. The whole translation process is shifted to the server as a remote service. The authors also proposed a cache management scheme in which the cache management decisions of the client are totally guided by the server in order to reduce the management overhead of 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 for every translation 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 loss or outage of service on the server and still keep low overhead. Moreover, the performance of emulation can be improved by asynchronous translation with the two-translator approach where the communication latency can be hidden.

Zhou et al. [23] proposed the framework of code server and the thin client downloads 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 service, and the client sends optimization requests to the server at runtime.

5. CONCLUSION

In this work, we 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 executes light-weight binary translation and basic emulation functions on 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.

With such a two-translator client/server approach, we also successfully off-load the DBT overhead of the thin client to the server and achieve significant performance improvement over the non-client/server mode. Experimental results show that the DBT of the client/server model can achieve 14% speedup over that of non-client/server model for x86-to-ARM emulation using SPEC CINT2006 benchmarks with test inputs, and are only 3.4X and 2.2X slower than the native execution with test and reference inputs, respectively, as opposed to 7.1X and 5.1X slow-down on QEMU. The results also show that the Client/Server mode achieves saving in power consumption because fewer instructions are executed with the Client/Server mode.

6. ACKNOWLEDGMENT

This work is supported in part by NSC grants NSC99-2221-E-001-003-MY3 and NSC99-2221-E-002-111-MY3, and is also supported in part by NSF grant CNS-0834599.

7. REFERENCES

[1] J. A. Baiocchi, B. R. Childers, J. W. Davidson, and 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.

[2] J. A. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser, and 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.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. PLDI*, pages 1–12, 2000.

[4] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on ItaniumÂő-based systems. In *Proc. Annual Microarchitecture Symposium*, 2003.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, pages 41–46, 2005.

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

[7] C. Dall and J. Nieh. Kvm for arm. In *12th Annual Linux Symposium*, 2010.

[8] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proc. ASPLOS*, pages 202–211, 2000.

[9] H. Guan, Y. Yang, K. Chen, Y. Ge, L. Liu, and 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.

[10] A. Guha, K. Hazelwood, and 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.

[11] D. Hiniker, K. Hazelwood, and M. D. Smith.

Improving region selection in dynamic optimization systems. In *Proc. Annual Microarchitecture Symposium*, pages 141–154, 2005.

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

[13] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and 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.

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

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

[16] L. Lin, C. Chu, T. Sun, A. Liang, and H. Guan. Distribit: A distributed dynamic binary execution engine. In *Proceedings of the third Asia International Conference on Modelling & Simulation*, 2009.

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

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

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI*, pages 89–100, 2007.

[20] http://pandaboard.org/node/300/#Panda.

[21] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. Annual Microarchitecture Symposium*, pages 135–148, 2006.

[22] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proc. VEE*, pages 175–185, 2006.

[23] S. Zhou, B. R. Childers, and 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.