Enabling Cross-ISA Offloading for COTS Binaries

Wenwen Wang, Pen-Chung Yew,
Antonia Zhai, and Stephen McCamant
University of Minnesota, Twin Cities
{wenwang, yew, zhai,
mccamant}@cs.umn.edu

Youfeng Wu and Jayaram Bobba
Intel Labs
{youfeng.wu,
jayaram.bobba}@intel.com

ABSTRACT

Work offloading allows a mobile device, i.e., the client, to execute its computation-intensive code remotely on a more powerful server to improve its performance and to extend its battery life. However, the difference in instruction set architectures (ISAs) between the client and the server poses a great challenge to work offloading. Most of the existing solutions rely on language-level virtual machines to hide such differences. Therefore, they have to tie closely to the specific programming languages. Other approaches try to recompile the mobile applications to achieve the specific goal of offloading, so their applicability is limited to the availability of the source code.

To overcome the above limitations, we propose to extend the capability of dynamic binary translation across clients and servers to offload the identified computation-intensive binary code regions automatically to the server at runtime. With this approach, the native binaries on the client can be offloaded to the server seamlessly without the limitations mentioned above. A prototype has been implemented using an existing retargetable dynamic binary translator. Experimental results show that our system achieves 1.93X speedup with 48.66% reduction in energy consumption for six real-world applications, and 1.62X speedup with 42.4% reduction in energy consumption for SPEC CINT2006 benchmarks.

Keywords

Computation Offloading; Dynamic Binary Translation; Dynamic Binary Optimization; Offloading Target Selection

1. INTRODUCTION

Mobile devices, such as smart phones and tablets, have become ubiquitous nowadays, and the trend seems likely to continue in the foreseeable future. People increasingly rely on their smart phones and tablets to accomplish more and more heavy tasks, e.g., image and voice processing, 3D games and even virtual reality. However, most mobile devices suffer from low computing capability (due to their power constraints) and limited battery life compared to their desktop and server counterparts. A recent study [29] shows that a mobile device is more than 5 times slower than a desktop computer when used to play the same chess game. Therefore, users have to put up with longer waiting time while playing the same game on mobile devices.

Some recent survey [10] conducted on 1000 individuals showed that 89% rated long battery life was among the top features they considered when buying a new smart phone. This result indicates that battery life is crucial to the commercial success of mobile devices. However, increasingly heavy workloads on mobile devices could shorten battery life and curtail good user experiences.

Previous research work [22, 29, 41, 31, 38, 18, 43, 20, 30, 42, 27, 19, 35, 26] has shown that work offloading is an effective way to improve performance and to prolong battery life of mobile devices. In a work offloading system, the computation-intensive tasks on a mobile device, i.e., the *client*, are sent to a more powerful *server* and executed remotely. After the server completes the work, the results are sent back to the client. With much more powerful computing capability of the server, the performance of the offloaded application can be improved, and the battery energy is conserved because of the work offloading and the shortened execution time.

One of the biggest obstacles to an offloading system is the difference in instruction set architectures (ISAs) between the client and the server, e.g., ARM on a mobile device and Intel x86-64 on a server. To overcome such difficulties, some existing offloading systems [22, 18, 26, 27, 42] rely on language-level virtual machines (VMs) such as Dalvik JVM and the Microsoft .NET Common Language Runtime (CLR) to hide the ISA differences with strong runtime support. However, such approaches often tie closely to the specific programming languages and their runtime systems, and the trend for such VMs is to incorporate more native code, e.g., Java Native Interface (JNI) libraries, so that they can be more adaptive to the native environment and/or to meet the performance requirements, e.g., Android ART runtime [1]. Basically, many of the JNI libraries are developed by third party developers, and the source code is usually not available to VM developers to build the x86-64 version for the server. With more native binaries integrated into such VM systems, their portability across ISAs is significantly hampered. A recent study [29] shows that one third of the top 20 open source Android applications contain more than 50% native code and spend more than 20% of the execution time in native mode.

MobiSys'17, June 19 - 23, 2017, Niagara Falls, NY, USA

DOI: http://dx.doi.org/10.1145/3081333.3081337

Some other offloading systems [30, 38, 29, 41, 31] rebuild/recompile mobile applications with the specific goal of offloading at runtime. To make optimal partition of the code between the client and the server, these offloading systems statically analyze the source code using different static performance estimators and cost models. Due to the complexity of the source programs, some static analyzers [30, 38, 41, 31] are limited to applications with simple structures such as media encoding and decoding programs, or requiring annotations in the source code from the developers for assistance. A recent approach [29] allows more general-purpose native applications. However, its applicability is still limited to the availability of the source code, which is usually unavailable in practice due to commercial reasons.

To overcome the above limitations, we propose to extend the capability of dynamic binary translation across the client and server boundary, and automatically offload the identified computation-intensive binary code regions and their related memory data at runtime. An *offline profiler* using dynamic binary instrumentation is used to select *offloading targets*, i.e., hot code regions that can be offloaded to the server. At runtime, a *dynamic binary optimizer* is employed on the client to facilitate the migration of offloading targets to the server. In addition, an enhanced *dynamic binary translator/optimizer* is used to retarget, re-optimize and emulate the client binaries on the server.

Since most existing dynamic binary translation systems only run on a single physical machine, we need to address several critical issues in order to support the new client/server environment, e.g., the handling of required I/O operations and system calls across the client/server boundaries, managing the amount of data transferred between the client and the server to minize the communication overhead, and the maintenance of the necessary data/memory consistency between the two.

In our offloading system, client's virtual addresses are mapped to the server virtual address space by the translator. The I/O operations performed in the offloading targets are also supported in our system by sending them back to the client. Using this approach, native binary applications can be offloaded seamlessly to a server without the limitations mentioned above.

A prototype has been implemented using an existing retargetable dynamic binary translation system HQEMU [24], which is based on QEMU with an LLVM backend for optimization purpose. Experimental results show that our offloading system achieves 1.93X performance speedup with 48.66% reduction in energy consumption for six real-world applications, and 1.62X performance speedup with 42.4% reduction in energy consumption for SPEC CINT2006 benchmarks on an ARM-based client and an Intel Xeon Ivy Bridge server.

In summary, we made the following contributions:

- We propose a cross-ISA offloading system for COTS binaries on a mobile client to a remote server. This approach is not limited to specific programming languages, runtime systems, nor the availability of application source code. *To the best of our knowledge, this is the first attempt to allow native binary applications to be offloaded across ISAs.*

- A prototype for such an offloading system has been implemented. It includes three parts: an offline profiler and a dynamic optimizer using dynamic binary instrumentation/rewriting to support the migration of binary code regions from client to server, and an enhanced dynamic binary translator/optimization system based on HQEMU to work with the client on the server.

- A number of experiments are conducted to demonstrate the effectiveness of the proposed approach on an ARM-based client and an Intel Ivy Bridge server. On average, 1.93X speedup with 48.66% reduction in energy consumption are observed on six real-world applications that include games and a JavaScript engine, and 1.62X speedup with 42.4% reduction in energy consumption on industry standard benchmark SPEC CINT2006.

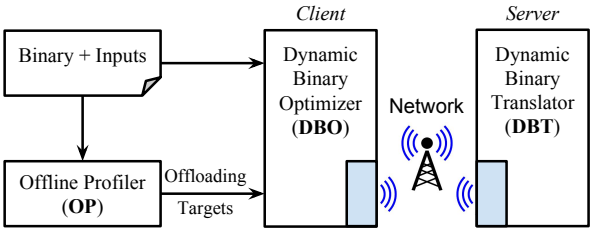

Figure 1: The framework of our offloading system.

The rest of this paper is organized as follows. Section 2 presents an overview of our offloading system. Section 3 describes some of its design issues. Section 4 presents the implementation details. Section 5 shows some experimental results. Section 6 discusses related work. And Section 7 concludes the paper.

2. SYSTEM OVERVIEW

Figure 1 shows the framework of our proposed offloading system. The system is composed of two subsystems, one on the client and the other on the server.

On the client side, the subsystem includes an *offline profiler* (OP) and a *dynamic binary optimizer* (DBO). The OP profiles the mobile application binaries offline on the client with some available inputs. It dynamically instruments the binary code to collect profiling information for offloading target selection. An *offloading target* (or *target* for short) is a binary code region offloaded to the server. The granularity of a target can be a basic block, a code trace, a code region that includes a loop, or a function. In our prototype system, the granularity is set at the *function* level. Determining the best granularity for an offloading target is beyond the scope of this paper.

For each profiled function, the OP records its execution time, the amount of live-in/live-out data accessed and I/O activities that can impact the communication overhead between the client and the server. Using a heuristic target selection algorithm, the OP performs target selection as described in Section 3.1. Generally, the most time-consuming functions are selected as targets in our prototype system.

After the offloading targets are identified, the DBO on the client can automatically offload the targets to the server at runtime. The DBO takes over the control of a mobile application immediately after the mobile application is launched.

The DBO tries to connect the client to the server via existing wireless network and checks the availability of the server. If the server is temporarily unavailable, no offloading is carried out and the execution proceeds as a native execution on the client. Otherwise, the DBO rewrites the beginning instructions of the target to redirect its execution to a *jump stub*, which is a code block that helps to offload the target to the server. After the target is offloaded, the DBO waits for the results to be returned from the server and resumes the execution of the mobile application.

In a jump stub, it saves the current execution state of the application including the content of its registers, and then sends the offloading request to the server. Each offloading request contains the saved execution state and the *beginning* and the *ending* addresses of the offloading target. After the request is sent, the DBO enters a wait mode for the results to come back from the server. In the wait mode, the DBO still needs to respond to the requests from the target running on the server, e.g., memory pages needed and/or the system calls from the target. More details are presented in Sections 3.2 and 3.3. When the server completes its emulation of the target, the DBO recovers the execution state of the application and resumes its execution until the next target is encountered.

Figure 1 also shows an enhanced *dynamic binary translator* (DBT) deployed on the server side, which receives and processes offloading requests from the client. Typically, a cross-ISA DBT system can dynamically translate *guest* binary code in one ISA to a different *host* ISA, and emulate the guest binary on the host machine by executing the generated host code [36]. In our offloading scenario, the client is the guest and the server is the host.

When an offloading request is received from the client via the wireless network, the DBT on the server initializes its internal emulation state according to the received execution state of the target. After the initialization, the DBT starts the emulation of the offloaded target. A memory page request is sent to the client if the page is not included in the initial execution state of the target. More details are presented in Section 3.2. The DBT terminates the emulation when the ending address of the target is encountered. Then it sends back the results together with the emulation state to the client, and then enters a wait mode for the next request from the client.

Figure 2 shows an example from a real-world application, GNU Chess [3], which is a chess-playing game. In this example, the OP first identifies that around 97.3% of the execution time is spent in the function `_ZN6engine6searchEv`, which is invoked at the address *0x2b778*. This particular function evaluates all potential moves and determines the best move among them. Thus, it is selected as an offloading target.

The DBO then replaces the beginning instruction of this target, i.e., the instruction at *0x2b778*, with a branch instruction and directs its execution to a jump stub as shown in the figure. When the execution reaches *0x2b778*, the jump stub is executed and an offloading request is sent to the server. Once the request is received, the DBT on the server starts its emulation from the address *0x2b778*. The emulation of the function `_ZN6engine6searchEv` terminates at the address *0x2b77c*, which is the return address of the function call. The DBT then sends back the emulation results to the client, and the client resumes its execution from *0x2b77c*. In this way, the function `_ZN6engine6searchEv` is offloaded to the server seamlessly.

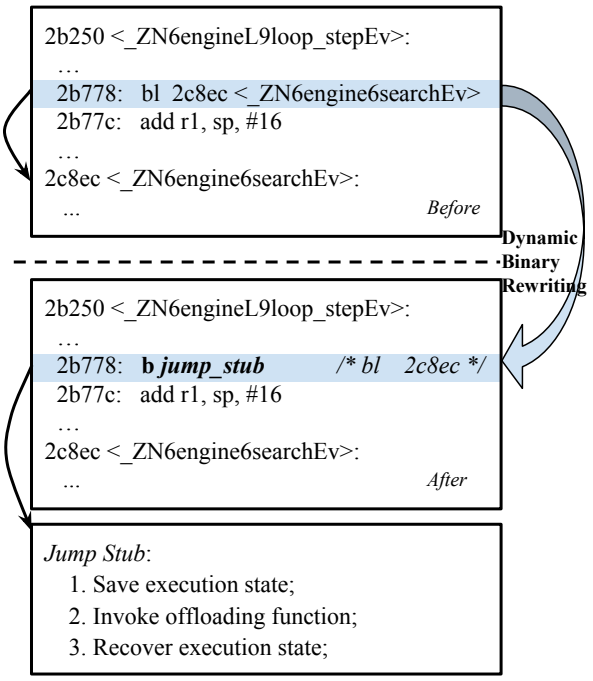

Figure 2: A real-world example from GNU Chess, with its binary code before and after dynamic binary rewriting, and the jump stub.

3. SYSTEM DESIGN ISSUES

As mentioned earlier, there are two main phases on the client, i.e., target selection (offline) and target offloading. The first phase is realized by the OP using a heuristic target selection algorithm. Target offloading in the second phase is achieved by the DBO using a dynamic binary rewriting technique.

The DBT on the server is responsible for receiving and processing the offloading requests from the client. During the DBT's emulation, data transfer may be required between the DBT (server) and the DBO (client). It is implemented in a way similar to demand paging. I/O operations and some system calls during the emulation also need to be handled between the DBT and the DBO. The following sections discuss these design issues in more details.

3.1 Target Selection for Offloading

We can avoid the target selection problem by sending the entire binary code that includes the application's binary and its shared libraries to the server, and asking the server to emulate the entire execution from the start. Although this simplifies the target selection problem, it has three fundamental limitations.

First, the communication overhead between the client and the server will increase substantially due to the large sizes of the application binaries and the shared libraries. Table 1 shows the sizes of the top 10 free applications from Google Play store [9] after their installation on a Google Nexus 6P smart phone. As shown in the table, up to 6 applications have more than 50 MB of executable binaries (including Java

Application	Category	Version	Size (MB)
Slither.io	Game	1.1.2	**43.35**
Messenger	Communication	69.0.0	**122**
Snapchat	Social	9.29.3	**109**
Facebook	Social	74.0.0	**249**
Pandora	Music & Audio	7.1	**46.16**
Instagram	Social	7.21.1	**48.15**
Clean Master	Tool	5.12.1	**70.42**
Color Switch	Game	2.4.0	**51.17**
Kika Emoji	Tool	3.4.148	**38.2**
Spotify Music	Music & Audio	5.2.0	**79.47**

Table 1: Application size of the top 10 free applications from Google Play store after their installation on a Google Nexus 6P smart phone.

bytecode). The largest application, i.e., Facebook, even approaches 250 MB.

Second, the performance benefit of offloading can be substantially reduced by the overhead of dynamic binary translation if there are too many infrequently executed code regions. Generally, a dynamic binary translator emulates a guest binary by translating it into host binary first [36]. For the *cold* code regions, i.e., the code regions that are rarely executed, the cost of translation often cannot be amortized during the application's lifetime. Previous work [34, 40] shows that almost 90% of the emulation time can be contributed to the translation overhead of cold code regions in graphical user interface (GUI) applications.

Third, it is hard for a client to recover from unexpected network failures if the entire execution is offloaded to the server. For example, suppose a chess playing game on the client is offloaded to the server. After several moves played, it encounters occasional connection disruptions, which means the client loses the connection to the server and cannot send/receive any data to/from the server. If the entire execution was offloaded to the server, the client would not be able to resume the execution because most of the intermediate data is on the server.

To overcome such limitations, our offloading system uses profiling to identify only *hot* code regions as offloading targets. The profiling information is gathered at the *function* level for simplicity. For each executed function, the following information is collected:

Execution Time The execution time is used to identify major time-consuming functions. The total execution time of a function is calculated by adding its own execution time and the time of all its callee functions.

Dynamic Memory Allocation and Deallocation Dynamic memory allocation and free operations are monitored to collect the starting address, the size, and other relevant information of the allocated memory region. The *time stamp* at the time of a dynamic memory allocation is assigned to the allocated memory region as its unique ID for future references.

Memory Access and I/O Operation Each memory access in a function is instrumented to collect the information such as its memory address, the size, and the ID of the accessed memory region. The I/O operations are also tracked to estimate the communication overhead.

Calling Context The calling context is the call path from the `main` function to the current function, which can be used by the OP to backtrack the location where the function is invoked and to distinguish different contexts of the same function.

Algorithm 1: Target Selection for Offloading

Input: (1) a ranked linked list F of hot functions based on hotness, i.e. execution time
(2) a DCG graph $G = (V, E)$
Output: a final set S of selected offloading targets

```
1  S ← ∅;
2  f ← HEAD(F);
3  while f ≠ NULL do
       // Step 1
4      if f ∉ S and f is not invoked during the execution of
         any function in S then
           // Step 2
5          T_f ← total execution time of f;
6          I_f ← total data size of I/O in f;
7          if I_f / T_f < M then
8              S ← S + {f};
               // Step 3
9              foreach v ∈ V do
10                 if e : v -n-> f ∈ E or e : f -n-> v ∈ E then
11                     if n > N and v ∉ S and v is not
                         invoked during the execution of any
                         function in S then
12                         T_v ← total execution time of v;
13                         I_v ← total data size of I/O in v;
14                         if I_v / T_v < M then
15                             S ← S + {v};
16     f ← f → next;
17 return S;
```

Based on the collected profiled information, the OP then proceeds with offloading target selection. Intuitively, the most time-consuming functions should be included in the *initial target set*. But, the execution time is not the only concern because some of these targets may have very large live-in and/or live-out data sets and extensive I/O activities, which can incur a large communication overhead and offset the benefit of work offloading.

Take GNU Chess [3] discussed in Section 2 as an example. There is a huge memory region (around 256 MB) that is frequently accessed in the hot function `_ZN6engine6searchEv`. This memory region is allocated and initialized dynamically in another function `_ZN6engine11trans_allocEPNS_5transE`, which is not a time-consuming function and, hence, will not be included in the initial target set. However, if this function is not offloaded and stays on the client, a huge amount of data has to be transferred through the network to the server when the function `_ZN6engine6searchEv` is being emulated on the server. To reduce this kind of excessive communication overhead, the OP heuristically extends the initial target set to include the functions that generate (or consume) large live-in/live-out data sets used (or produced) by the functions in the initial target set. It may also exclude the targets in the initial target set that have excessive I/O activities to avoid excessive communication overhead.

To simplify the target selection process, if a function is selected as an offloading target, all of its callee functions are also selected for offloading. A directed data communication

graph (DCG) $G = (V, E)$ is constructed to facilitate the selection process. Each node $v \in V$ represents a function, and an edge $e : v_1 \xrightarrow{n} v_2 \in E$ connects the functions v_1 and v_2 if there are n memory pages produced in v_1 and accessed in v_2 at runtime. Thus, if v_2 is offloaded to the server and v_1 stays on the client, these memory pages need to be transferred from the client to the server over the network.

The initial target set is determined based on the *hotness* of each function, i.e., the total execution time including its callee functions. Algorithm 1 shows the target selection process. The input F is the ranked list of hot functions in the initial target set based on the hotness. The output S is the final selected target set by OP. For each target $f \in F$, there are three steps in the algorithm.

In the first step, the OP checks if f has been included in S, or called during the execution of any function in S. If yes, the OP skips f and goes to next target in F. Otherwise, the OP moves to the second step.

In the second step, the OP examines the I/O operations in f (including its callee functions) to see if they are excessive in f. If they are not excessive, f is included into S and the OP moves to the third step.

In the third step, the OP traverses each node $v \in V$ in G that is connected with f, i.e., $e : v \xrightarrow{n} f \in E$ or $e : f \xrightarrow{n} v \in E$. If the value of n is large, it indicates that function v should be offloaded together with f to reduce the communication overhead. The OP also examines the I/O operations in v to further check if it is suitable for offloading. If yes, v can be included into S. The selection process iterates until all of the functions are examined.

For each target in S, we need to locate all of its call sites in the collected calling context and their corresponding return locations, i.e., the instructions that follow the call instructions, which are referred to as the *beginning* and the *ending* instructions of this offloading target, respectively. The DBO on the client replaces the beginning instruction by a branch instruction to a jump stub that enables the offloading of this target. Similarly, the DBT on the server uses the ending instruction to terminate the emulation of the offloaded target. Here, we did not use the entry and exit points of a function because there are often multiple exit points, i.e., return instructions, in a function and it is not easy to locate all of them in the binary. Besides, this choice of beginning and ending instructions can be extended easily to other types of offloading targets with different granularities, such as hot loops or traces, without additional work required.

3.2 Memory Mapping

To guarantee the correct execution of the offloaded targets, the data must be maintained coherent between the client and the server. To meet this requirement, the relevant memory regions in the virtual address space of an offloaded application on the client are mapped to their corresponding memory regions in the virtual address space of DBT on the server, as shown in Figure 3. This means the application and the DBT can see the same memory layout. Here, we only focus on offloading from a 32-bit or 64-bit client to a 64-bit server, where we can take advantage of a larger or equal address space available. Offloading from a 64-bit client to a 32-bit server is quite rare, and DBT on such systems is still an open research issue [16].

To simplify the memory mapping between the client and the server, it is done at the *page* level. We can take advantage of the access permission mechanism provided in modern operating systems. Figure 4 shows some details of the memory mapping between the mobile application on the client and the DBT on the server.

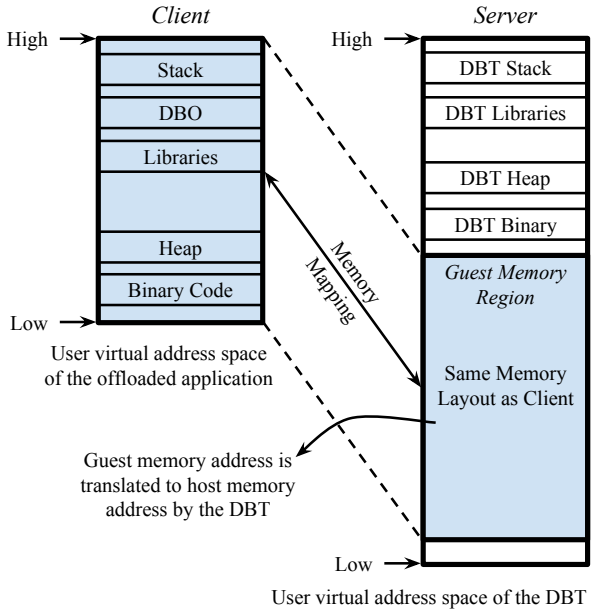

Figure 3: Memory address mapping between the offloaded application and the guest memory region in the DBT.

Initially, the DBT on the server allocates a guest memory region, but leaves it uninitialized. After receiving an offloading request from the client, the DBT initializes the guest memory region with the memory layout encapsulated in the offloading request. This initialization only organizes the guest memory region according to the memory layout in the offloading request. The actual data may still be on the client side, and is brought to the server at the page level when needed.

During the emulation of the offloaded target in the DBT, the first access to any memory page in the guest memory region will trigger a page access exception. The exception handler then sends a memory page request to the client, as shown in Figure 4. The DBO on the client responds to the request with the content of the requested page, and marks this page inaccessible on the client. Then the DBT loads the received page content to its corresponding page in the guest memory region, and marks it accessible. After the DBT finishes the emulation of the offloaded target, the DBO resumes the execution on the client with the received results as shown in Figure 4. A page transfer from the DBT to the DBO is triggered by an access to an inaccessible page on the client, and then this page is marked as inaccessible in the DBT.

To guarantee the same memory layout after the target is offloaded to the server, each time when a new memory region is dynamically allocated or an old memory region is deallocated on one side, the other side is informed to perform the same operation to maintain the same memory layout.

The memory mapping mechanism might be complicated by multi-threaded applications when some threads are offloaded to the server while others are remained on the client. The offloaded threads and the threads on the client can then

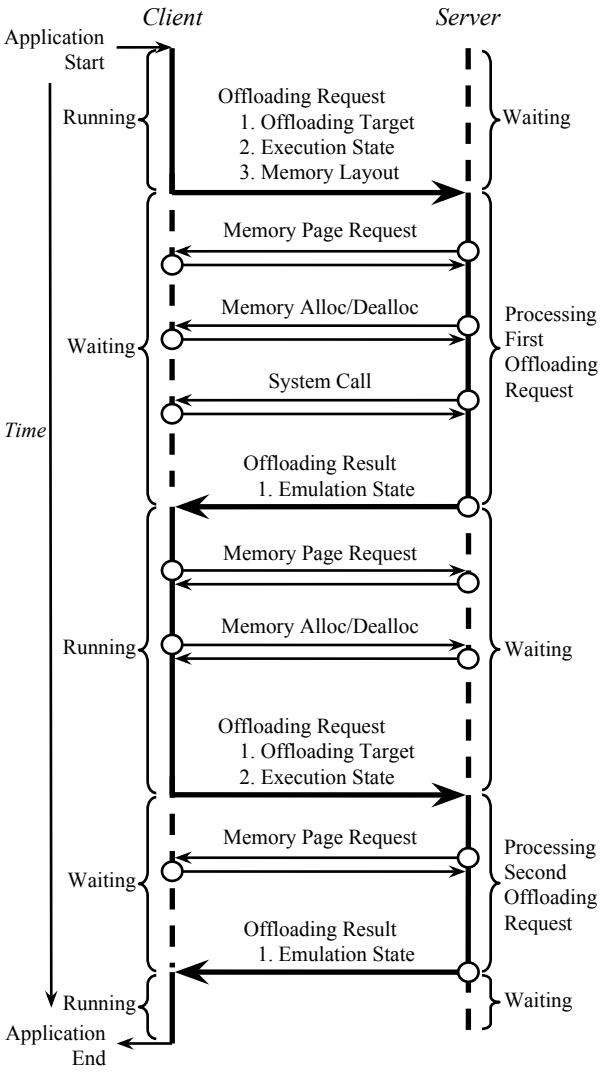

Figure 4: The detailed network communication between the client and the server.

synchronize via memory, e.g., through locks. In that case, our OP can be extended to collect synchronization information (e.g., `lock()/unlock()`) for each function and take them into consideration when selecting offloading targets.

3.3 I/O and System Calls

One significant limitation of most existing offloading systems is to avoid offloading code regions with I/O operations [22, 29]. However, it is very common in mobile applications to have I/O operations in hot code regions, e.g., reading or writing a file. In most mobile applications, the I/O operations are realized by invoking system calls provided by the operating system. To offload code regions with system calls, our offloading system classifies guest system calls into three categories, each of which is handled in a different way.

In the first category, the guest system calls can be emulated directly with similar system calls on the server, e.g., `gettimeofday`, `clock_gettime` and the like. System calls in this category are usually used to acquire generic information such as time and date. The execution results of such system calls are generally same/similar on both client and server[1]. The DBT can invoke these system calls on the server directly without going back to the client.

In the second category, the guest system calls need to be sent back to the client, e.g., `open`, `read`, `write` and `unlink`. These system calls are typically related to I/O operations, and thus the results of these system calls are system specific. For example, the `open` system call should open the file on the client, not on the server. The DBT sends the parameters of such system calls back to the client for execution, as shown in Figure 4. The DBO on the client executes received system call and sends the results back to the server.

In the last category, the guest system calls are required to be executed on both the server and the client, e.g., `brk`, `mmap`, `mremap` and `munmap`. These system calls are usually related to dynamic memory allocation and deallocation. Thus, these system calls might modify the memory layout of both virtual address spaces. To guarantee the same memory layout between the client and the server system calls in this category are executed in both DBT and DBO.

By classifying system calls into above three categories, our system can offload code regions that include I/O and other system calls. However, if the amount of I/O and system calls is too excessive, the network latency can significantly increase the execution time because each I/O operation requires two connections between the server and the client. Such targets may have to be excluded from the target set as mentioned earlier, and remain on client.

To reduce the communication overhead involved in I/O operations, the DBT will buffer the results of several output I/O operations on the server until they have to be sent back to the client for execution in one communication session. Similarly, for input I/O operations, the DBO will also try to buffer as much data as possible and send the data to the server in as few communication sessions as possible. In this way, the number of communication sessions between the server and the client required for I/O operations can be reduced substantially.

3.4 Network Failure Recovery

Typically, unexpected network disruptions and failures are caused by lost or closed network connections between the client and the server. To tolerate these occasional network failures, the client should be able to recover the execution when it loses response from the server during the offloaded execution. Our offloading system tries to support such recovery with minimal effort. Basically, the point of execution when the client sends an offloading request to the server is considered as a *nominal checkpoint*. If there is a network failure after the client sends an offloading request to the server, e.g., no response to this request from the server, the client can resume the execution from the nominal checkpoint by disabling the offloading function. Note that there is no additional work required at the nominal checkpoint (i.e., there is no need to save machine states) because the client halts its execution and waits for a response from the server at the nominal checkpoint.

To support a network failure recovery, the server needs to track modified guest memory pages when processing an

[1]Here, we assume the server has the same system time as the client. This is reasonable in practice because two connected computers can synchronize their system times with each other.

offloading request from the client. This can be easily done by marking the guest memory region in the DBT as read-only before the server starts to process each offloading request. After an offloading request is finished, the modified guest memory pages are sent back to the client. The client can then update corresponding memory pages before resume its execution. In addition, the client needs to record the results of each system call executed during the offloaded execution. These recorded system call results could be reused during a recovery execution when there is a network failure. With these extensions, the client can recover the execution from an unexpected network failure with minimal overhead.

4. IMPLEMENTATION

A prototype of the proposed offloading system has been implemented with three major components: OP, DBO and DBT. In our implementation, the client is an ARM-32 development board and the server is an Intel Xeon Ivy Bridge. Also, both the client and the server have a memory page size of 4KB.

The OP component is implemented using DynamoRIO [14], which is a dynamic binary instrumentation framework. DynamoRIO exports an interface for users to build user-defined program instrumentation tools. In our implementation, several DynamoRIO tools are used to gather profiling information. Each function call/return is instrumented to collect execution time information at runtime. Memory loads, stores, and system calls are also instrumented to gather data accesses to memory and I/O operations. In addition, the information on the calling context of each function is collected based on the calling context encoding method [37].

The DBO is implemented as a shared library on the client. It intercepts the `__libc_start_main` function via the environment variable `LD_PRELOAD`. In this way, the DBO can perform binary rewriting on the mobile application before its `main` function is executed. Alternatively, the DBO can be implemented as a lightweight dynamic binary instrumentation system to support statically linked applications.

A jump stub is generated for each offloading target. To accommodate these jump stubs, a memory page is created on the application heap. The page is marked as read-only and executable after the generation of the jump stubs. The original binary code is saved for future reference before it is rewritten and replaced by a branch instruction to the jump stub. If the offset from the original binary code to the jump stub is too large, e.g., more than 2^{25} in ARM-32, a load instruction to load the program counter: `ldr pc, [pc, -4]` is employed to perform the branch operation instead, and the absolute address of the jump stub is stored immediately after the load instruction.

In the first offloading request, the current memory layout on the client is included and sent to the server. This is done in the DBO by reading the file located at `/proc/self/maps`, which contains the detailed information about the available memory regions in the current process. In our implementation, the system call `mprotect` from Linux is used to set different access permissions of a virtual memory page. A signal handler is registered for the signal SIGSEGV, which is the signal received from the operating system when a protected page is accessed. Some applications might have registered their signal handlers as well. In that case, the original signal handler is saved in the DBO. If the received SIGSEGV signal is not caused by our offloading system, the original signal handler is invoked from our signal handler in the DBO. In addition, to track dynamic memory allocation and free operations on the client, the DBO also wraps related library functions, e.g., `mmap` and `munmap`, in `libc.so.6`. The original library functions are invoked from our wrapper functions.

The DBT is implemented within an existing retargetable dynamic binary translation system, HQEMU [24], which is based on QEMU [11]. HQEMU leverages LLVM [28] to generate more optimized code for the hot code regions. To amortize additional performance overhead introduced by LLVM, HQEMU spawns helper threads to perform the translation and optimization. In our implementation, helper threads are also enabled for performance consideration.

To implement memory mapping in HQEMU, `mprotect` is used to change the access permission of guest memory pages. HQEMU has its own signal handler for the signal SIGSEGV. Thus, we keep the original signal handler and insert our signal processing function at the beginning of the original handler. Each time when a SIGSEGV signal is received, our signal processing function is invoked first, and it checks the fault memory address to see if it is in the guest memory space or not. If yes, a memory page fetching request is triggered and sent to the client for the real content of the page. Otherwise, our signal processing function returns to the original signal handler and no request will be generated. The SIGSEGV signal is then handled by the original signal handler in HQEMU.

One special case in this signal handling process is from the dynamically generated binary in the offloaded target, e.g., the offloaded target is a JavaScript engine that includes a JIT compiler, which can dynamically generate and optimize guest binary at runtime. To maintain the consistency between the original guest binary code generated by the JavaScript engine and the translated host code, HQEMU (also QEMU) marks each guest binary page generated by the JavaScript engine read-only when the binary code in this page is being translated [11]. Thus HQEMU can detect the code changes on this page when a write event to this page occurs, e.g., by the JavaScript engine itself. This kind of page exceptions should be handled by the original signal handler in HQEMU. Our signal processing function ignores such page exceptions by checking if the page is a read-only page marked by HQEMU before sending a memory page fetching request to the client.

5. EVALUATION

Table 2 shows some characteristics of the applications evaluated in our experiments. We did not use most of the mobile applications listed in Table 1 because our implemented prototype system is constrained by the capability of HQEMU, which does not support those mobile applications but only desktop applications. HQEMU could be extended to emulate mobile applications, e.g., Android apps, which is left as part of our future work. We instead used two different types of applications to evaluate the effectiveness of the system. They include six real-world applications and the industry standard benchmark SPEC CINT2006. Among the six real-world applications, GNU Go [4] is a go game engine using the territorial effect values to find the best move. Free Awale [2] is a hole-and-seed game based on a min-max Von Neuman's algorithm. GNU Chess [3] is a chess engine using the classical principal variation search (PVS) algorithm

Application	Lang.	LoC (K)	Description	Size (MB)	Time (S)	#MemPage		#SystemCall		Comm. (KB)	
						C2S	S2C	OnC	OnS	C2S	S2C
GNU Go	C	229.5	Go Game	8.51	629.06	963	232	0	837	4223.92	979.26
Free Awale	C	9.7	Hole and Seed Game	0.05	243.54	13	4	0	0	53.04	40.69
GNU Chess	C++	48.1	Chess Game	1.27	325.88	216	91	1292	27594	890.81	481.71
Natch	C	21	Game Solver	0.32	250.63	116	12	738	1162	468.43	89.70
Sudoku	C	0.8	Puzzle Game	0.03	238.91	30	6	0	0	157.22	32.80
SpiderMonkey	C/C++	336	JavaScript Engine	108.97	677.62	1016	19	507	6986	5479.75	113.81
400.perlbench	C	128	Perl Interpreter	3.67	177.57	1564	50	8310	2318	20167.62	639.3
401.bzip2	C	5.7	Compression	0.21	340.85	307	21	77	83	62312.82	117.06
403.gcc	C	385.7	C Compiler	11.04	7.75	840	9	52	172	3444.21	230.11
429.mcf	C	1.6	Vehicle Scheduling	0.07	244.62	90	7	613	12	2605.42	123.8
445.gobmk	C	157.7	Go Game	5.85	554.4	7891	80	1729	292	35650.42	554.82
456.hmmer	C	40.7	Gene Sequence	1.01	322.06	184	10	63	39	815.63	90.47
458.sjeng	C	10.5	Chess Game	0.45	806.14	725	7	1515	6	2938.73	92.96
462.libquantum	C	2.6	Quantum Computing	0.15	15	121	9	3	54	486.94	46.29
464.h264ref	C	36	Video Compression	1.77	562.4	266	9	904	757	5539.08	82.71
471.omnetpp	C++	26.7	Event Simulation	4.17	413.26	302	202	517	701	1229.15	1014.02
473.astar	C++	4.3	Path Finding	0.23	558.19	314	26	104	2284	5105.09	128.73
483.xalancbmk	C++	266.9	XML Processing	47.2	442.68	621	16	80543	932	42316.02	40959.39

Table 2: Applications evaluated in our experiments. **Lang.**: programming language, **LoC (K)**: thousand lines of source code, **Size (MB)**: size of the application binary, **Time (S)**: execution time on the client in seconds, **#MemPage**: number of memory pages transferred, **C2S**: from the client to the server, **S2C**: from the server to the client, **#SystemCall**: number of system calls invoked during the emulation, **OnC**: executed on the client, **OnS**: emulated on the server, **Comm. (KB)**: the amount of data transferred between the client and the server.

with iterative deepening. Natch [7] is a proof game solver used to find the shortest moves to a given position. Sudoku [8] is a puzzle game generator with a difficulty level estimator. SpiderMonkey [6] is a JavaScript engine from Mozilla, which has been deployed in various Mozilla products including Firefox. These applications are quite representative and their stripped-down versions could also be found on smart mobile devices. For SPEC CINT2006, the *train* input set (instead of the larger *reference* input set) is purposely used to reflect the fact that most mobile applications are predominantly smaller and shorter applications with short execution times, which make them more sensitive to DBT overheads. Benchmarks with more than one input under the train input set are tested with all inputs covered.

To evaluate those real-world applications, we played 5 steps for the games against a computer, i.e., GNU Go, Free Awale, and GNU Chess. For Natch, a given position is solved for 16 moves. Sudoku is evaluated to generate and solve 6 puzzles. SpiderMonkey is evaluated on Google Octane [5], which is a benchmark suite used to measure a JavaScript engine's performance by running a series of test applications. Our evaluation covers 11 of 15 applications in Octane because HQEMU failed to emulate 3 of the applications and produced random emulation errors for the remaining application.

Our evaluation platform includes a client and a server. The client is an ARM development board equipped with a Quad-Core Cortex-A7 running at 1.3 GHz with a 2-GB main memory and the operating system is Linaro-14.04 with Linux-3.4.39. The server is equipped with a 16-Core Intel Xeon E5-2650 Ivy Bridge running at 2.60 GHz with Hyper-threading enabled. It also has a 64-GB main memory running Ubuntu-14.04 with Linux-3.13. Both the client and the server are connected to a public 802.11g WiFi network.

To collect the energy consumption data, the ARM development board is powered with an external 4V battery. During the measurements, there is no other task running on the board. The Linux command line tool `powerstat` is used to measure the power consumption on the board. `powerstat` can collect the whole-system power statistics in a user-specified sampling time interval. In our measurements, the sampling interval is set at 1 second, which is the minimum sampling interval supported by `powerstat`. In addition, to reduce the influence of random factors, each application is run three times, and the arithmetic mean of the three runs is used as the final result.

5.1 Performance Improvement

The first benefit of work offloading is the performance improvement because of the significant performance gap between the client and the server. Assume P_{Server} and P_{Client} are the performance of a mobile application running on the server and the client, respectively. The overhead introduced by the network communication is $O_{Network}$. Considering that the offloaded binary code is emulated by HQEMU instead of running natively on the server because of the need of DBT, we assume the overhead introduced by HQEMU is O_{HQEMU}. The performance improvement, I, that can be

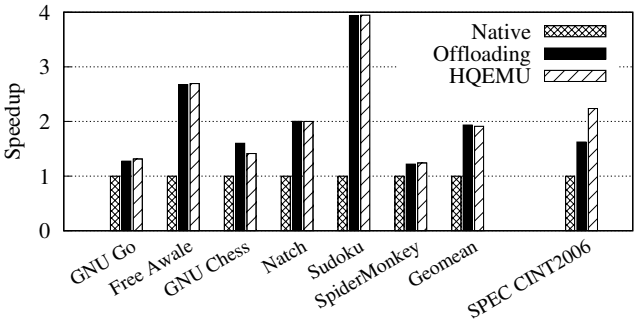

Figure 5: Performance speedup. The baseline is the native execution on the client without offloading. The HQEMU performance is from the emulation of a benchmark on original HQEMU (running entirely on server). SPEC CINT2006 is aggregated using the geometric mean.

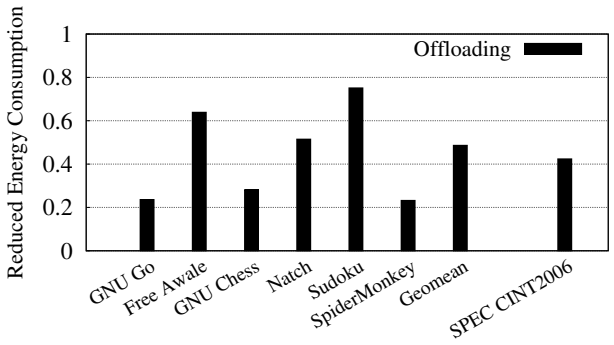

Figure 6: Reduced energy consumption of all benchmarks. The baseline is the amount of energy consumed during the native execution on the client without offloading. The results of SPEC CINT2006 are aggregated using the geometric mean.

achieved on our offloading system is shown in the following equation:

$$I \approx P_{Server} - P_{Client} - O_{Network} - O_{HQEMU} \quad (1)$$

Figure 5 shows the speedups on our offloading system. The baseline is the native execution on the client without offloading. The performance of original HQEMU running only on the server is also presented for comparison. The performance results of SPEC CINT2006 is aggregated using the geometric mean, and presented separately to avoid skewing the overall speedup because of its large number of programs in the benchmark suite and most of them are computation intensive.

As shown in Figure 5, our system can achieve good performance improvement for all of the applications. For most of the evaluated applications, our offloading system achieves similar performance as HQEMU emulating them on server, i.e., they are approaching the server performance with DBT.

An interesting anomaly is observed on GNU Chess. It achieves a 1.6X speedup with offloading, while the emulation using HQEMU on the server only gains 1.41X speedup. After studying this application, we found it is a multi-threaded application and the best move searching function is only invoked by one thread. Thus, only this thread is offloaded to the server in our offloading system with good speedup. However, HQEMU emulates multi-threaded guest applications sequentially, which degrades its overall performance. On average, 1.93X performance speedup can be observed for the real-world applications with our offloading system, compared with 1.91X performance speedup of original HQEMU.

For SPEC CINT2006 benchmark suite, our offloading system gains an overall speedup of 1.62X, while the emulation on HQEMU can achieve a 2.24X speedup. The performance difference comes from two sources. The first is from some extremely short running applications, such as 403.gcc. These applications complete their execution in only a few seconds. Due to the short running time, the network communication becomes a dominant overhead even if the communication volume is small. The second is from the large amount of data transferred from client to server in some benchmarks. The last two columns in Table 2 show the amount of data transfer between the client and the server. For 483.xalancbmk, it requires 41.32 MB of data from the client to the server, and 40 MB of data from the server to the client because this application processes a huge XML file and writes the processed result to another file. Similarly, 60.85 MB of data needs to be transferred from the client to the server for 401.bzip2 because it applies compression on this file.

5.2 Energy Efficiency

In general, the energy consumption of an application should be consistent with its performance improvement because the battery usage is proportionate to the execution time. Suppose W_{Native} is the average power consumption of the native execution on the client without offloading, and T_{Native} is the execution time. Correspondingly, $W_{Offloading}$ and $T_{Offloading}$ are the average power consumption and the execution time with offloading, respectively. The energy consumption reduced by offloading S can be represented by the following equation:

$$\begin{aligned} S &= W_{Native} \times T_{Native} - W_{Offloading} \times T_{Offloading} \\ &= W_{Native} \times (T_{Native} - T_{Offloading}) + \\ &\quad (W_{Native} - W_{Offloading}) \times T_{Offloading} \\ &= W_{Native} \times I + W_{\Delta} \times T_{Offloading} \end{aligned} \quad (2)$$

The reduced energy consumption comes mainly from two sources, i.e., $W_{Native} \times I$ and $W_{\Delta} \times T_{Offloading}$, where W_{Δ} is the reduced power consumption due to offloading.

Figure 6 illustrates the energy consumption on the client reduced by our offloading system. Here the baseline is the energy consumed in the native execution on the client without offloading. As shown in the figure, our offloading system successfully reduces the energy consumption for all of the evaluated applications. Even when the performance improvement of GNU Go and SpiderMonkey is not as much as other applications after offloading to server, they can still achieve 26.62% and 23.26% energy reduction, respectively. On average, 48.66% and 42.4% energy consumption is reduced, respectively, for the real-world applications and SPEC CINT2006.

Figure 7 shows some detailed power consumption behavior in the first 500 seconds on our offloading system compared to the native execution without offloading on the client. As shown in the figure, our offloading system exhibits peak power consumption in the initial stage due to the network activity between the client and the server for the required

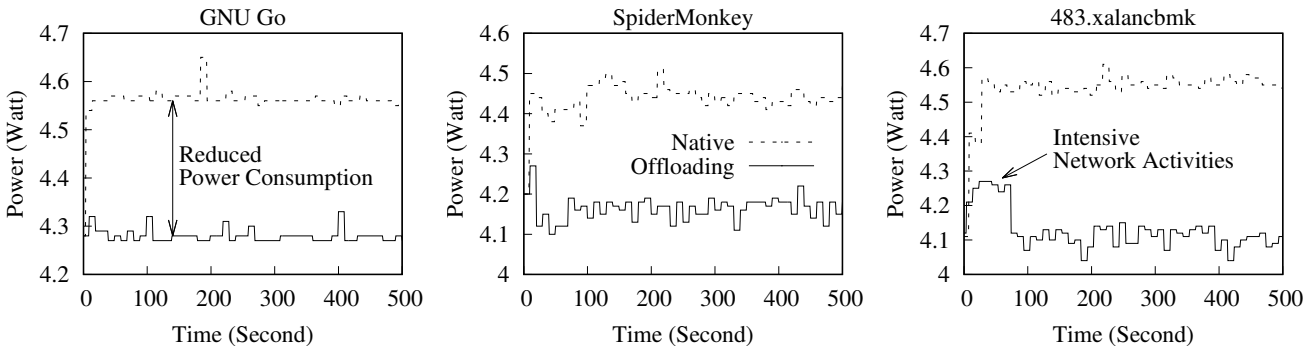

Figure 7: Power consumption in the first 500 seconds of the execution with and without offloading.

data transfer. The heavy power consumption is observed for 483.xalancbmk in the first 74 seconds due to the large input file transferred from the client to the server. However, in most cases, the client then settles into the wait mode, and the power consumption reduced to a level similar to the system standby state.

Even though we aim to reduce the power consumption of an application by offloading the computation-intensive code regions to the server side, it is probable that the power consumption might actually increase due to the potential fluctuation in the network communication overheads between the client and the server. It is generally hard to detect such scenarios [18, 22]. In that case, the benefit of computation offloading might come only from the performance improvement.

5.3 Network Sensitivity

To study the performance sensitivity to the network latency, we also tried to simulate real-world network environments by inserting randomized delays to each connection between the server and the client. The delay is implemented using `usleep` system call from Linux, which suspends the execution of the calling thread for an interval in microseconds specified by the parameter. To emulate different network environments, the parameter of `usleep` is generated using the expression: `random()` mod $(m + 1)$, where m is the maximum network delay in microseconds. Since our target client/server environments are those with short network latencies such as fog computing, the maximum network delay for each connection is set to 10^5 microseconds in this network sensitivity study.

Figure 8 shows the performance results of the 6 real-world applications with different values of m. As expected, the performance speedup achieved by our offloading system decreases as the network delay between the client and the server increases. The reason is obvious, because the increased network latency lowers the performance benefit of work offloading.

However, one interesting phenomenon that can be observed from this figure is that the decrease in speedup is rather slow relative to the increase in network latency for most of applications, i.e., the speedup is quite insensitive to the network delay until the maximum delay approaches 10^5 microseconds, at which point the performance drops suddenly for some applications, e.g., GNU Chess and SpiderMonkey. This demonstrates the capability of our offloading system to tolerate reasonable network latency, which makes the work offloading quite robust in practice.

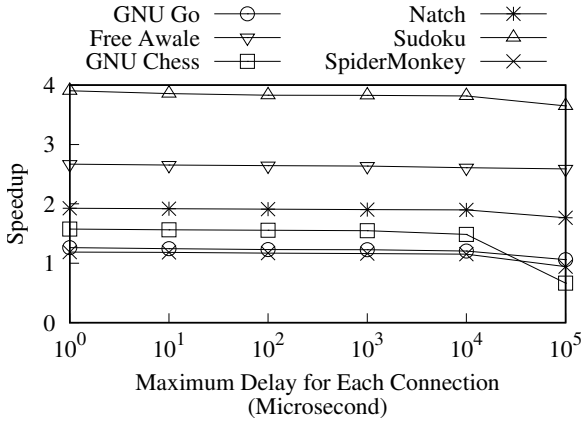

Figure 8: Network sensitivity with randomized delay for each connection between the client and the server.

6. RELATED WORK

A significant amount of previous research work [17, 30, 18, 43, 22, 29, 33, 31, 20, 42, 19, 41, 23, 38, 35, 27, 26, 21] has demonstrated that computation offloading is an effective approach to improve the performance of mobile devices and prolong their battery life.

MAUI [18], ThinkAir [27], and COMET [22] are representative offloading systems using language-level virtual machines such as JVM and the Microsoft .NET CLR. MAUI and ThinkAir recompile mobile applications to enable their offloading features, i.e., they rely on the availability of the source code. Furthermore, they require specific annotations in mobile applications from expert programmers to enable their offloading targets. COMET offloads mobile applications in Java bytecode without the recompilation requirement, but it needs Dalvik JVM support to hide the differences in the underlying ISAs with strong runtime support. Tango [21] replicates a mobile application and executes it on both the client and the server. It allows either replica to lead the execution because either of the client or the server execution may be faster during different phases of the application. Besides, it also leverages techniques from deterministic re-

play to synchronize the two replicas. However, Tango still relies on Dalvik JVM to tackle the cross-ISA issue between the client and the server and cannot offload mobile applications using native libraries. Additionally, similar offloading systems include CloneCloud [17], Cuckoo [26], HELVM [31], JESSCIA2 [43] and the system described in [42].

Compared with these offloading systems, the main difference of our system is that our system works directly on native binary code across different ISAs. Thus there is no requirement on the availability of the source code, which is usually unavailable due to commercial reasons in practice, and there is no required support from the programming languages or runtime systems. The OP in our system automatically selects offloading targets based on the collected profiling information. So no programmer annotation is required. Besides, there is also no specific constraint on the native code regions to have I/O and system calls, which is usually avoided in most offloading systems mentioned above.

Paranoid Android [33] offloads computation-intensive security checks from mobile devices to the cloud. In Paranoid Android, a tracer is placed on the client to dynamically record the execution trace of each executed mobile application. Once the execution gets completed, the generated trace is transmitted to the server over the network. Then the server can replay the execution using a system level virtual machine and the computation-intensive security checks are applied during this process. Different with Paranoid Android, our system offloads the computation from the client to the server on the fly, which avoids the requirement of the record-replay support. In addition, our system uses a lightweight process level dynamic binary translator on the server instead of a system level virtual machine, which can substantially reduce the resource usage on the server and improve the offloading throughput.

Some new programming language features have been designed to facilitate offloading. Network Objects [12] specify an object whose methods can be invoked over a network. Obliq [15] allows computation roaming over the network. Emerald [25] supports fine-grained object mobility such as arrays, records, and integers. Instead of extending a programming language, our system offloads mobile applications written in existing languages with no need to rewrite the applications.

Some schemes [30, 38, 41, 31] use static analysis techniques to partition mobile applications into tasks that can run on mobile devices and on servers separately. To minimize the communication overhead, various partitioning algorithms and cost analysis are proposed. However, most of those algorithms are limited to mobile applications with simple structures, e.g., media encoding and decoding applications, due to conservative pointer alias analysis. To offload general-purpose applications, Native Offloader [29] leverages offline profiling and performance estimator to select offloading targets, which is similar to our offloading system. But, code regions that include assembly code, system calls, or unknown external library calls are not allowed to be offloaded in Native Offloader, while there is no such limitation in our approach. The applicability of Native Offloader is also dependent on the availability of the source code, which is often unavailable in practice.

Some other offloading systems optimize task offloading for cloud environments, e.g., managing cloud resources to support offloading requests from various mobile devices [35], making offloading decisions with the consideration of cloud parameters and related rules [20, 19], and offloading with an intelligent selection of partitioning policy adaptively [23]. Our offloading system can work effectively with such systems. In addition, our offloading system can also benefit from various optimization techniques for DBT systems [34, 13, 32, 39, 40].

7. CONCLUSION

In this paper, we present a cross-ISA offloading system for native binary applications on mobile devices. The system offloads the identified computation-intensive binary code regions and their related data from the client to the server at runtime. The client leverages a dynamic binary optimizer to rewrite the binaries and to send offloading requests. An enhanced dynamic binary translator is employed on the server to retarget and re-optimize the client binaries to server ISA. With this approach, the native binary applications on the client can be offloaded to the server with a different ISA seamlessly. A prototype has been implemented using an existing retargetable dynamic binary translation system HQEMU. Experimental results show that our offloading system can achieve 1.93X performance speedup with 48.66% reduction in energy consumption for six real-world applications and 1.62X performance speedup with 42.4% reduction in energy consumption for SPEC CINT2006 benchmarks on an ARM-based client and an Intel Xeon Ivy Bridge server.

8. ACKNOWLEDGEMENTS

We would like to thank Gang Shi and Minjun Wu for their help in setting up the experimental environment and collecting the experimental results. We are also very grateful to Geoffrey Challen and the anonymous reviewers for their valuable comments and feedback. This work is supported in part by the National Science Foundation under the grant number CNS-1514444.

9. REFERENCES

[1] ART and Dalvik. https://source.android.com/devices/tech/dalvik/index.html.
[2] Free Awale. http://www.nongnu.org/awale.
[3] GNU Chess. https://www.gnu.org/software/chess.
[4] GNU Go. https://www.gnu.org/software/gnugo.
[5] Google Octane. https://developers.google.com/octane.
[6] Mozilla SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[7] Natch. http://natch.free.fr/Natch.html.
[8] Sudoku. http://dlbeer.co.nz/articles/sudoku.html.
[9] Top 10 free Android applications from Google Play store. https://play.google.com/store/apps/collection/topselling_free.
[10] Your smartphone's best app? Battery life, say 89% of Britons. https://www.theguardian.com/technology/2014/may/21/your-smartphones-best-app-battery-life-say-89-of-britons.
[11] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[12] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 217–230, New York, NY, USA, 1993. ACM.

[13] D. Bruening and V. Kiriansky. Process-shared and Persistent Code Caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 61–70, New York, NY, USA, 2008. ACM.

[14] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[15] L. Cardelli. A language with distributed scope. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 286–297, New York, NY, USA, 1995. ACM.

[16] C.-J. Chang, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew. Efficient memory virtualization for cross-isa system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 117–128, New York, NY, USA, 2014. ACM.

[17] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[19] H. Flores and S. Srirama. Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning. In *Proceeding of the Fourth ACM Workshop on Mobile Cloud Computing and Services*, MCS '13, pages 9–16, New York, NY, USA, 2013. ACM.

[20] H. Flores and S. Srirama. Mobile code offloading: Should it be a local decision or global inference? In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 539–540, New York, NY, USA, 2013. ACM.

[21] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 137–150, New York, NY, USA, 2015. ACM.

[22] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[23] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, PERCOM '03, pages 107–114, Washington, DC, USA, 2003. IEEE Computer Society.

[24] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. `http://itanium.iis.sinica.edu.tw/hqemu/`.

[25] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988.

[26] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. *Mobile Computing, Applications, and Services: Second International ICST Conference, MobiCASE 2010, Santa Clara, CA, USA, October 25-28, 2010, Revised Selected Papers*, chapter Cuckoo: A Computation Offloading Framework for Smartphones, pages 59–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[27] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the 31st Annual IEEE International Conference on Computer Communications*, INFOCOM '12, pages 945–953, Washington, DC, USA, 2012. IEEE Computer Society.

[28] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. `http://llvm.org`.

[29] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 521–532, New York, NY, USA, 2015. ACM.

[30] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 238–246, New York, NY, USA, 2001. ACM.

[31] S. Ou, K. Yang, and A. 'Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, PERCOM '06, pages 116–125, Washington, DC, USA, 2006. IEEE Computer Society.

[32] J. a. Porto, G. Araujo, E. Borin, and Y. Wu. Trace Execution Automata in Dynamic Binary Translation. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pages 99–116, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual*

Computer Security Applications Conference, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[34] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 74–88, Washington, DC, USA, 2007. IEEE Computer Society.

[35] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura. Cosmos: Computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '14, pages 287–296, New York, NY, USA, 2014. ACM.

[36] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[37] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 525–534, New York, NY, USA, 2010. ACM.

[38] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 119–130, New York, NY, USA, 2004. ACM.

[39] W. Wang, C. Wu, T. Bai, Z. Wang, X. Yuan, and H. Cui. A pattern translation method for flags in binary translation. *Journal of Computer Research and Development*, 51(10):2336–2347, 2014.

[40] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant. A general persistent code caching framework for dynamic binary translation (dbt). In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 591–603, Denver, CO, June 2016. USENIX Association.

[41] C. Xian, Y.-H. Lu, and Z. Li. Adaptive computation offloading for energy conservation on battery-powered systems. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 01*, ICPADS '07, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[42] K. Yang, S. Ou, and H.-H. Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *Comm. Mag.*, 46(1):56–63, Jan. 2008.

[43] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 381–388, Washington, DC, USA, 2002. IEEE Computer Society.