ABSTRACT

Although mobile devices have been evolved enough to support complex mobile programs, performance of the mobile devices is lagging behind performance of servers. To bridge the performance gap, computation offloading allows a mobile device to remotely execute heavy tasks at servers. However, due to architectural differences between mobile devices and servers, most existing computation offloading systems rely on virtual machines, so they cannot offload native applications. Some offloading systems can offload native mobile applications, but their applicability is limited to well-analyzable simple applications. This work presents automatic cross-architecture computation offloading for general-purpose native applications with a prototype framework that is called Native Offloader. At compile-time, Native Offloader automatically finds heavy tasks without any annotation, and generates offloading-enabled native binaries with memory unification for a mobile device and a server. At run-time, Native Offloader efficiently supports seamless migration between the mobile device and the server with a unified virtual address space and communication optimization. Native Offloader automatically offloads 17 native C applications from SPEC CPU2000 and CPU2006 benchmark suites without a virtual machine, and achieves a geomean program speedup of 6.42× and battery saving of 82.0%.

Categories and Subject Descriptors
I.2.2 [Automatic Programming]: Program Transformation; D.4 [Operating Systems]: Process Management

Keywords
Native Computation Offloading, Mobile Cloud Computing

1. INTRODUCTION

Despite the advance of mobile devices, performance of mobile devices is lagging behind performance of desktops and servers. For example, Table 1 presents the execution time of the same chess game movement computation on a Samsung Galaxy S5 smartphone and a Dell XPS 8700 desktop. Though the smartphone is state of the art, the smartphone is more than 5 times slower than the desktop across all the different thinking depths. Thus, mobile users should suffer from more than 5 times longer waiting time for each turn or play the game with a stupider AI. Meanwhile, people want to use more and more complex applications such as office programs and 3D games on their mobile devices. Therefore, improving the mobile device performance becomes crucial to satisfy the mobile users.

Recent research has demonstrated that computation offloading systems can alleviate the performance overhead of the mobile devices by borrowing the high computing power from servers [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. The offloading systems send heavy and machine-independent tasks to servers, and receive their execution results from the servers. Since servers generally have more powerful computing resources than mobile devices, the systems can increase the performance of the mobile applications. Here, while most mobile platforms adopt ARM processors, most server platforms use x86 processors. To overcome the architectural difference, most existing computation offloading systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] rely on virtual machines (VMs) such as Dalvik VM and Microsoft .NET Common Language Runtime (CLR) to virtualize underlying architectures.

However, relying on VMs limits applicability of the offloading systems to Java or C# programs, so the systems cannot offload native C programs. To better understand how much codes are written and executed in native languages in real-world mobile applications, we investigated top 20 open source Android applications [17, 18]. Table 2 shows that around one third of the 20 applications include native codes more than 50% and spend more than 20% of the total execution time to execute them. Moreover, Mehrara et al. shows that Java and JavaScript programs are more than 6 times slower than the same C program due to the inter-
Native Offloader is a compiler-runtime cooperative system that automatically offloads machine-independent heavy tasks of a general-purpose native application from a mobile device to a server without any annotation and virtual machine. For Native Offloader to seamlessly and efficiently offload native applications across different architectures, there exist the following challenges.

- **ISA difference**: Since mobile devices and servers adopt different processors such as ARM and x86, Native Offloader should compile a mobile application into two different binaries with different ISAs. To support various combinations of architectures, Native Offloader...

### Table 2: Ratios of lines of C/C++ codes and their execution time in the top 20 open source Android applications. The execution time is measured under the described runtime behaviors.

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>Description</th>
<th>C/C++</th>
<th>Total</th>
<th>Ratio (Loc)</th>
<th>Runtime Description</th>
<th>Ratio (Exec. Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdAway</td>
<td>3.10</td>
<td>AD block</td>
<td>321,282</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Orbot</td>
<td>14.1.4-noPIE</td>
<td>Tor client</td>
<td>675,851</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Firefox</td>
<td>4.0</td>
<td>Web browser</td>
<td>8,094,678</td>
<td>3,584,526</td>
<td>33.12%</td>
<td>Read articles with ads</td>
<td>33.12%</td>
</tr>
<tr>
<td>VLC Player</td>
<td>1.5.1.1</td>
<td>Media player</td>
<td>6,433,726</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Open Camera</td>
<td>1.2</td>
<td>Camera</td>
<td>10,336</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>osmAnd</td>
<td>2.1.1</td>
<td>Map/Navigation</td>
<td>53,695</td>
<td>450,573</td>
<td>11.92%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Synching</td>
<td>0.5.0-beta5</td>
<td>File synchronizer</td>
<td>59,461</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>AFWall+</td>
<td>1.3.4.1</td>
<td>Network traffic controller</td>
<td>59,741</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>2048</td>
<td>1.95</td>
<td>Puzzle game</td>
<td>2,232</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td>4.804</td>
<td>Email client</td>
<td>96,588</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>PDF Reader</td>
<td>0.4.0</td>
<td>PDF viewer</td>
<td>594,434</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>ownCloud</td>
<td>1.5.8</td>
<td>File synchronizer</td>
<td>77,141</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>DAWdroid</td>
<td>0.6.2</td>
<td>Private data synchronizer</td>
<td>7,435</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Barcode Scanner</td>
<td>4.7.0</td>
<td>2D/QR code scanner</td>
<td>50,201</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>SatStat</td>
<td>2</td>
<td>Sensor status monitor</td>
<td>7,480</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Cool Reader</td>
<td>3.1.2.72</td>
<td>Ebook reader</td>
<td>681,001</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>OS Monitor</td>
<td>3.4.1.0</td>
<td>OS monitor</td>
<td>74,513</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Orweb</td>
<td>0.6.1</td>
<td>Web browser</td>
<td>14,124</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>PPSSPP</td>
<td>1.0.1.0</td>
<td>PSP emulator</td>
<td>1,438,322</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
<tr>
<td>Adblock Plus</td>
<td>1.1.3</td>
<td>AD block</td>
<td>63,779</td>
<td>0</td>
<td>0.00%</td>
<td>N/A</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

The contributions of this paper are:

- The first automatic computation offloading for general-purpose native applications across different architectures
- Architecture-aware memory unification and optimization schemes for efficient cross-architecture cooperative execution such as between ARM and x86
- An in-depth evaluation of the Native Offloader prototype on ARM and x86 platforms using 17 native C applications from SPEC CPU2000 and CPU2006 benchmark suites
The Native Offloader compiler automatically partitions the original IR codes into offloading-enabled IR codes in four steps; 1) target selection, 2) memory unification code generation, 3) partition, and 4) server specific optimization. In the target selection step, the compiler finds heavy tasks from profiling, filters out machine dependent tasks, and selects only profitable tasks through static performance estimation. In the memory unification code generation step, the compiler replaces all the memory allocation sites with UVA allocation, and realigns memory layouts of structures to provide the same virtual address space and data structures across mobile devices and servers. If the target architectures have different address sizes such as 32 bits and 64 bits, or different endianness, the compiler inserts translation codes. Here, the Native Offloader compiler achieves information about target architectures from back-end compilers. In the partition step, the compiler partitions the IR codes for mobile devices and servers, and inserts data communication codes only for objects that will be prefetched. The Native Offloader runtime will communicate the others if necessary at run-time. Finally, the compiler applies additional optimizations such as remote I/O and function pointer management that increase coverages of offloading candidates. Section 3 describes details of the compiler.

The runtime system seamlessly executes the offloading-enabled binaries on a mobile device and a server. Since the Native Offloader compiler and the back-end compilers generate native codes for each machine, the runtime executes the binaries without any virtual machine. To efficiently deliver live-in values of the offloaded tasks from the mobile device to the server, the runtime adopts copy-on-demand and communication optimization. Section 4 presents more details of the runtime.

3. NATIVE OFFLOADER COMPILER

Figure 2 illustrates how the Native Offloader compiler automatically transforms the original IR codes to be offloading-enabled. The compiler selects profitable code regions with profiling results (Section 3.1), and inserts memory unification codes for memory instructions and data structures (Section 3.2). The compiler partitions the original IR codes to offloading-enabled ones (Section 3.3), and applies additional optimization schemes to increase the coverage of offloading candidates (Section 3.4). Figure 3 shows code examples about how the Native Offloader compiler transforms a mobile chess game application into offloading-enabled applications for a mobile device and a server.

3.1 Target Selection

Hot function/loop profiler: The hot function/loop profiler measures execution time, invocation count, and memory usage of each function and loop in an application with a profiling input. The profiling results will be used for the performance estimator to predict execution time and select profitable targets. Table 3 shows profiling results for the chess game application in Figure 3(a).

Function filter: The function filter checks whether a function or a loop includes a machine specific instruction. If so, the filter marks the function or loop as a machine specific task, and rules out the task from offloading candidates. The
void runGame () {
    char from, to; double score;
}

int main () {
    int maxDepth;
    Piece *board;
    EVALFUNC evals[7] = {Pawn, ..., King};
    int numPieces = 0;
    EVALEVALFUNC evals[7] = {Pawn, ..., King};
    for (int i = 0; i < 64; i++) {
        Piece *piece = &board[i];
        if (piece->type == Pawn) {
            numPieces++;
        }
    }
    if (numPieces == 8) {
        // Game is over.
        return 0;
    }
    while (!gameover) {
        // Client partitioning (Sec. 3.3)
        board = malloc(sizeof(*piece) * 64);
        // Global var realloc. (Sec. 3.2)
        maxDepth_re = u_malloc(sizeof(int));
        // Server partitioning (Sec. 3.3)
        if (isProfitable (getAITurn_id)()) {
            requestOffload (getAITurn_id);
            sendData ();
        } else {
            mv = getAITurn ();
            updateBoard (mv);
        }
        // Move getAITurn ()
        Move_t mv = receiveReturn ();
        sendData ();
        // Move getPlayerTurn ()
        return mv;
    }
}

Move_t getPlayerTurn () {
    FcnID offID;
    while (true) {
        offID = acceptOffload ();
        if (offID == acceptOffload ()) {
            receiveData ();
        } else {
            sendData ();
        }
    }
}

Move_t getAITurn () {
    EVALFUNC evals[7] = {Pawn, ..., King};
    int maxDepth = 5;
    while (!gameover) {
        // Unified heap management (Sec. 3.2)
        board = malloc(sizeof(*piece) * 64);
        if (isProfitable (getAITurn_id)()) {
            requestOffload (getAITurn_id);
            sendData ();
        } else {
            mv = getAITurn ();
            updateBoard (mv);
        }
        // Move getAITurn ()
        Move_t mv = receiveReturn ();
        sendData ();
        // Move getPlayerTurn ()
        return mv;
    }
}

Figure 3: Simplified chess AI game code example
remote I/O functions [23] in Section 3.4, the filter excludes the I/O instructions from the machine specific instructions because the remote I/O functions execute the original I/O functions at the mobile device. For example, in the example code in Figure 3(a), the filter rules out getPlayerTurn and its callers such as runGame and main from offloading candidates because getPlayerTurn includes a user interactive I/O function call, scanf. However, although getPlayerTurn includes an output function call, printf that is one of the remote output functions, the filter classifies getPlayerTurn as an offloading-enabled function.

Static performance estimator: The static performance estimator calculates expected performance gains for the offloading candidates, and decides the final offloading targets. Here, the static performance estimation is only used for code generation. The Native Offloader runtime dynamically makes offloading decisions for the targets at run-time through dynamic performance estimation with run-time values. Therefore, the selected targets may not be offloaded at run-time.

Ideally, the performance gain is the difference between the server execution time ($T_s$) and the mobile execution time ($T_m$) on the same task. If the server is $R$ times faster than the mobile device on average, the ideal gain is $T_m \times (1 - \frac{1}{R})$. However, there always exist communication overheads in offloading execution, so the actual gain is the difference between the ideal gain and the communication overhead ($T_c$). If an offloading task uses $MB$ memory and its network bandwidth is $BW$, the task requires $\frac{MB}{BW}$ seconds to send the shared data in the memory. Since the shared data are communicated twice from a mobile device to a server and from a server to a mobile device, the network cost should be doubled. Moreover, if the task is invoked $N_{invoc}$ times, the cost should be multiplied due to repeated communication. As a result, the performance estimator calculates the performance gain ($T_g$) according to Equation 1.

$$T_g = (T_m - T_s) - T_c$$

$$= T_m \times (1 - \frac{1}{R}) - 2 \times \frac{M}{BW} \times N_{invoc} \quad (1)$$

Finally, the target selector chooses offloading targets if their predicted performance gains are positive. For example, Table 3 shows the performance estimation results based on the profiling results for the chess game example in Figure 3(a). The estimator assumes that $R$ is 5 and $BW$ is 80Mbps, and calculates their performance gains. Although all the candidates show positive ideal performance gains,
some of the candidates show negative numbers if the communication costs are considered. Especially, although for_i and for_j have similar execution times and memory usages, for_j shows the negative performance gain because it is invoked 12 times more than for_i causing huge expected communication costs. Since getPlayerTurn and runGame are filtered due to the interactive I/O function call, the target selector chooses getPlayerTurn and for_i as offloading targets. In Figure 3, the Native Offloader compiler offloads only getPlayerTurn to simplify the example.

### 3.2 Memory Unification Code Generation

To execute offloading tasks across different architectures without a virtual machine, Native Offloader provides the unified virtual address (UVA) space. Unlike distributed shared memory systems [24, 25, 26, 27] that provide only a shared memory view to different platforms, Native Offloader does not only provide a shared memory view, but also unifies memory layouts for an object across different architectures because different architectures may allocate the same object as different memory layouts. Before partitioning the offloading targets, the memory unification code generator transforms the whole IR codes to allocate objects as the same memory layout on the same UVA space.

**Heap allocation replacement**: The Native Offloader compiler replaces memory allocation/deallocation call sites with UVA allocation/deallocation function calls to allocate memory objects on the UVA space. For example, in Figure 3(b), the compiler changes malloc at line 21 to u_malloc. The compiler replaces all the allocation/deallocation sites because a server may access an object not on the UVA space due to imprecise static alias analysis.

**Referenced global variable allocation**: Since the Native Offloader compiler transforms offloading targets at IR level, back-end compilers may allocate global variables at different addresses. As a result, if a global variable is referenced at a mobile device and its pointer is dereferenced at a server, the pointer may point a different object. To solve this problem, the Native Offloader compiler allocates all the referenced global variables on the UVA space using u_malloc, and transforms their uses to dereferenced references. For example, since maxDepth is dereferenced at Line 19 in Figure 3(a), the compiler transforms its declaration to int *maxDepth_re at Line 11 with an allocation site at Line 17, and changes all the its uses like Line 19 in Figure 3(b).

**Memory layout realignment**: Because there is no unified rule about a memory layout for an object in C language across different platforms, the offloaded task may access different data with the same address on the UVA space. Figure 4 shows that a mobile device and a server use different memory layouts for the same object Move with different memory layouts. If a server accesses score in Move, the server will read a garbage value from its memory although the virtual memory space is unified. To overcome the memory layout difference, the Native Offloader compiler statically realigns the server memory layout to the mobile memory layout. Native Offloader chooses the mobile one as a standard layout because the mobile device is the default one in the computation offloading.

**Address size conversion**: If a mobile device and a server use different address sizes such as 32 bits and 64 bits, the Na-

![Figure 4: Memory layout realignment for type Move in Figure 3. Gray, dark gray, and slashed boxes represent meaningful data, paddings for memory layout realignment, and outside of the structure respectively.](image-url)
sults, the function filter excludes most of the IR codes from offloading targets, and Native Offloader cannot generate profitable offloading codes. To increase the offloading coverage, the Native Offloader compiler replaces well-known output function call sites with remote I/O function calls [23]. The remote I/O function sends I/O requests from the server to the mobile device, so it allows the mobile device to remotely execute the I/O operations at the local environment. For example, the compiler replaces `printf` with `r_printf` (Line 61 in Figure 3(c)). Here, most remote I/O functions are output functions because a remote input operation requires round-trip communication. For file streams, Native Offloader supports remote input operations because it can prefetch data and amortize the communication overheads.

**Function pointer mapping:** Like global variables, the Native Offloader compiler cannot manipulate the addresses of functions that the back-end compilers decide. As a result, if a code region includes a function pointer, the compiler cannot offload the code region. To increase the offloading coverage, the compiler creates a function address table that maps function addresses between a mobile device and a server, and inserts an address conversion code before the function pointer uses like `eval` at Line 56 in Figure 3(c).

4. **NATIVE OFFLOADER RUNTIME**

The Native Offloader runtime seamlessly and cooperatively executes the offloading-enabled tasks on a mobile device and a server. Figure 5 illustrates a life cycle of the runtime: local execution, initialization, offloading execution, and finalization.

**Local execution:** Before executing an offloading-enabled task, a mobile device locally executes the native application, and a server waits for the task. Since only the mobile device executes the application, the server memory is empty.

When the mobile device meets the offloading-enabled task, the Native Offloader runtime dynamically estimates local execution time and offloading execution time for the task. Unlike the static performance estimation of the Native Offloader compiler, the dynamic performance estimation reflects the current network bandwidth, memory usage, and target execution time information, so the Native Offloader runtime can avoid offloading under unfavorable situation such as slow network connection.

**Initialization:** If the dynamic performance estimation decides to offload the task, the Native Offloader runtime initializes the server to execute the offloaded task. First, the mobile device sends offloading information such as offloaded task ID, current stack pointer, and page table to the server. Second, the server creates a new process for the offloaded task with a different stack space from the mobile stack. This stack reallocation allows stacks of the server and the mobile device not to be overlapped on the UVA space. Then, the server updates its page table, so the server can have the same UVA space with the mobile device. To reduce communication costs, the mobile device prefetches parts of mobile heap memory to the server that are most likely used in the server.

**Offloading execution:** Although the server updates its page table, there are physical pages not yet copied. During offloading execution, if the server accesses data in one of the physical pages, a page fault occurs for the page. The Native Offloader runtime hooks the page fault, and copies the physical page from the mobile device to the server (copy-on-demand). Once the page is copied, the server can access data in the page again and again without a page fault. Here, the mobile device and the server can access the shared data without any address translation because they have the same UVA space. To reduce communication costs, the Native Offloader runtime also checks dirty pages and sends only the dirty pages to the mobile device after the offloaded task.

**Finalization:** After finishing the offloading execution, the server sends a termination signal to the mobile device with a return value, dirty pages, and updated page table. To terminate the offloading process without keeping the offloading data, the server sends all the dirty pages to the mobile device instead of using the copy-on-demand on the mobile device. Since the Native Offloader runtime sends only the dirty pages, the amount of communication is not huge in practice. After updating the modified program state, the mobile device resumes its local execution after the offloaded task.

While communicating data between the mobile device and the server, the Native Offloader runtime batches and compresses the communicated data to reduce the communication overheads. The batching reduces the number of communication operations by keeping the communicated data in a buffer and sending the buffer once. This batching process amortizes the overheads from the communication function calls. The runtime also compresses the communicated data before sending it to overcome the limited network bandwidth. Here, since compression requires much more time than decompression, the Native Offloader runtime applies the compression only to the server-to-mobile communication to avoid performance slowdown due to the compression overhead on the mobile device.

![Figure 5: Life cycle of an offloaded task](image)
The Native Offloader compiler finds more than 85% of the whole program execution time for offloading statistics. Offloaded functions and loops cover threading supports of the Native Offloader framework.

A Monsoon Power Monitor [28] is used to measure battery consumption at the smartphone. The Native Offloader component is evaluated with 17 native C programs from SPEC CPU2000 and CPU2006 [30]. Among all the C programs in SPEC CPU2000 and CPU2006, we exclude 400.perlbench and 403.gcc because the LLVM compiler cannot compile the programs for the smartphone. Also, Native Offloader cannot find any profitable offloading target for g++ parser, 254.gap and 255.vortex. The evaluation cannot include the mobile applications in Table 2 due to lack of a bridge between LLVM and Android NDK and lack of multi-threading supports of the Native Offloader framework.

Table 4 illustrates features of the evaluated programs and offloading statistics. Offloaded functions and loops cover more than 85% of the whole program execution time for all the programs. The Native Offloader compiler finds more than one offloading target like the 188.ammp case, and executes the same target multiple times if the target is invoked multiple times like AMMPmonitor, update and think.

Figure 6 presents the whole program execution time and battery consumption of the offloaded applications normalized to local execution time and battery consumption on the smartphone. In each graph, the x-axis shows evaluated applications and the y-axis shows the normalized execution time and battery consumption. All the execution times and battery consumption were averaged over five runs. We use different inputs for profiling and evaluation.

### 5. EVALUATION

Native Offloader is evaluated on a Samsung Galaxy S5 smartphone with a 2.5GHz quad-core Krait 400 CPU and a Dell XPS 8700 desktop server with an Intel 3.60GHz quad-core i7-4790 processor. The smartphone runs the Android 4.4.2 (KitKat) operating system, and the desktop server runs Ubuntu 14.04. To analyze how the network environments affect the performance, Native Offloader is evaluated under two different wireless environments such as slow connection (802.11n, maximum bandwidth is 144Mbps) and fast connection (802.11ac, maximum bandwidth is 844Mbps). A Monsoon Power Monitor [28] is used to measure battery consumption at the smartphone. The Native Offloader compiler builds on the LLVM compiler infrastructure [29].

The Native Offloader prototype is evaluated with 17 native C programs from SPEC CPU2000 and CPU2006 [30] as listed in Table 4. Among all the C programs in SPEC CPU2000 and CPU2006, we exclude 400.perlbench and 403.gcc because the LLVM compiler cannot compile the programs for the smartphone. Also, Native Offloader cannot find any profitable offloading target for g++ parser, 254.gap and 255.vortex. The evaluation cannot include the mobile applications in Table 2 due to lack of a bridge between LLVM and Android NDK and lack of multi-threading supports of the Native Offloader framework.

Table 4 illustrates features of the evaluated programs and offloading statistics. Offloaded functions and loops cover more than 85% of the whole program execution time for all the programs. The Native Offloader compiler finds more than one offloading target like the 188.ammp case, and executes the same target multiple times if the target is invoked multiple times like AMMPmonitor, update and think.

Figure 6 presents the whole program execution time and battery consumption of the offloaded applications normalized to local execution time and battery consumption on the smartphone. In each graph, the x-axis shows evaluated applications and the y-axis shows the normalized execution time and battery consumption. All the execution times and battery consumption were averaged over five runs. We use different inputs for profiling and evaluation.

### 5.1 Execution Time

Figure 6(a) shows that Native Offloader achieves performance speedups for all the evaluated programs in slow and fast wireless environments, and reduces 82.0% and 84.4% of program execution time on geomean of program execution time. Ideal offloading means execution time without any overhead such as data communication and translation. For 175.vpr, 179.art, 183.eqake, 188.ammp, 433.milc, 456.hmmer and 482.sphinx3, Native Offloader achieves almost ideal performance speedups. These programs require little communication compared to computation. For example, the offloaded function of 456.hmmer that searches against a gene sequence DB takes only the initialized parameters as its inputs. Therefore, 456.hmmer communicates only a small amount of data such as the input parameters and printed results.

Native Offloader achieves performance improvement for
458.sjeng that invokes think multiple times even on
the slow network environment. Considering that 458.sjeng,
a chess game, is one of the representative user-interactive
applications, the speedup shows that Native Offloader can
successfully offload user-interactive applications.

Figure 7 presents overheads of Native Offloader for all
the evaluated programs in different network environments.
To deeply analyze the performance of Native Offloader, we
break the total execution time into computation, function
pointer translation, remote I/O operation and communication.
The computation time is equal to the ideal execution
time. The function pointer translation overhead is spent for
Native Offloader to find a correct address of a function pointer.
The remote I/O operation overhead is the execution time of
remote I/O functions. The communication overhead is spent
for Native Offloader to transfer memory. Network environ-
ments such as bandwidth and latency affect the communica-
tion overhead. Here, Figure 7 does not illustrate the address
size conversion that changes 32-bit pointers to 64-bit ones
due to its negligible overhead. Moreover, Native Offloader
does not suffer from endianness translation overheads be-
cause the mobile device and the server use the same endi-
anness, little-endian.

164.gzip, 401.bzip2, 429.mcf, 458.sjeng and
470.ibm have a huge amount of communication compared
to their execution time. Since the communication overhead
increases in the slow network, the programs are very sensi-
tive to the network bandwidth. Therefore, for the slow net-
work connection, though the Native Offloader compiler gen-
erates offloading-enabled codes, the dynamic performance
estimator in the Native Offloader runtime decides not to of-
load the offloading target. For example, Native Offloader
does not offload spec_compress from 164.gzip ac-
cording to Equation 1. The dynamic performance estima-
tion allows Native Offloader not to suffer from performance
slowdown in an unexpected slow network environment.

300.twolf, 445.gobmk and 464.h264ref suffer
from high remote I/O operation overheads. During the of-
foading execution, 300.twolf reads a file about cell in-
formation to optimally place cells, 445.gobmk reads files
about previous play records, and 464.h264ref reads a
video file to encode. Unlike the other programs, these pro-
grams execute remote input operations that require expen-
sive round-trip communication. Therefore, the programs have
higher remote I/O operation overheads than the others.

The analysis results show that 445.gobmk, 458.sjeng
and 464.h264ref spend lots of time to translate func-
tion pointers. 445.gobmk and 458.sjeng have function
pointer arrays such as commands and evalRoutines to
manage next commands and piece movements respectively.
464.h264ref has function pointers about various SAD (Sum of Absolute Differences) computations for video quality metrics. Since the programs refer the function pointers every time when they execute commands, simulate a piece movement and encode each frame, the function pointers are dereferenced a huge number of times causing high function pointer translation overheads.

Though frequent remote I/O operations and function pointer translations cause high overheads, the optimizations play a key role in increasing offloading coverage and performance because many applications include I/O operations and function pointers in their hot functions and loops.

### 5.2 Battery Consumption

Figure 6(b) presents that Native Offloader saves geomeans of 77.2% and 82.0% battery consumption in the slow and fast wireless environments compared to the local execution. Native Offloader reduces battery consumption for all the programs except 164.gzip that requires huge power for communicating an input file and its compressed result. Since the performance estimator focuses on the execution time reduction, the dynamic performance estimator cannot catch the additional battery consumption of 164.gzip.

Generally, battery consumption results are very similar to the execution time results because the battery usage is proportional to the execution time. However, 300.twolf, 445.gobmk, 464.h264ref, and 482.sphinx3 consume relatively more battery than the ideal execution compared to the other programs. For detail analysis about the battery consumption, Figure 8 illustrates required power over time for two similar programs such as 458.sjeng and 445.gobmk. In the fast network connection, the smartphone consumes about 300mW for idle state, 1350mW for waiting signals, 2000mW for data reception, and 2000mW to 5000mW for data transmission. During three invocations of the offloaded function, 458.sjeng spends more power than 2000mW only at the beginning and the end of each invocation to communicate the shared data and results. However, 445.gobmk does not only spend huge power at the beginning and the end of the offloading task, but also continuously spends 2000mW to manage remote I/O requests. As a result, 300.twolf, 445.gobmk, 464.h264ref, and 482.sphinx3 consume relatively more battery than the others due to many remote I/O operations.

Especially for 300.twolf and 445.gobmk, Native Offloader spends more battery on the fast network environment than the slow one unlike the others. Figure 8(b) and Figure 8(c) show power consumption over time in different network environments for 445.gobmk. The fast network connection requires 2000mW to handle remote I/O requests while the slow one requires 1700mW. As a result, for 300.twolf and 445.gobmk that frequently request remote I/O operations more than the other programs, Native Offloader consumes more battery in the fast network environment than in the slow environment despite shorter execution time.
Native Offloader automatically finds heavy and machine independent tasks from general-purpose native mobile applications without any annotation, and achieves performance speedups by offloading the tasks without a virtual machine. Table 5 summarizes related works of this paper.

Static partitioning algorithms [10, 14, 21, 22] represent a program as a graph in which vertices are computation tasks and edges are data flows between the tasks. The algorithms partition the vertices into mobile device tasks and server tasks, and insert computation codes for the edges between mobile device tasks and server tasks. However, the algorithms work well only for well-analyzable applications such as media encoding and decoding programs because of conservative static alias analysis. If an application has irregular data access patterns and control flows, the algorithms should conservatively send all the data that the offloaded tasks may touch, and pay unnecessary communication costs. Since the Native Offloader runtime delivers only accessed data via the copy-on-demand on the unified virtual address space (UVA), Native Offloader offloads general-purpose applications without suffering from the huge communication overheads.

Roam [2] and Cuckoo [7] propose programming models for computation offloading, and offload complex general-purpose applications. However, they require programmers to manually analyze and transform the applications. MAUI [4] and ThinkAir [8] automatically transform the applications to offloading-enabled ones, but they still require programmer annotations to find the offloading targets. With the profiler and the performance estimator, Native Offloader automatically finds and transforms offloading targets.

To alleviate programmers’ efforts, OLIE [6, 11], DiET [13], HELVM [12, 15], and CloneCloud [3] also automatically find and partition offloading tasks without any programmer annotation. However, since these computation offloading systems rely on virtual machines such as Java VM and Microsoft .Net CLR, the systems cannot offload native applications. Cooperating with front-end and back-end compilers, Native Offloader automatically generates offloading-enabled native binaries for each platform, and executes the binaries without a virtual machine.

Like COMET [5] that provides distributed shared memory for computation offloading, Native Offloader provides a shared memory view for a mobile device and a server via the UVA space. Unlike COMET [5], Native Offloader additionally inserts translation codes that make the native applications have the same memory layout for the same object on different architectures.

To reduce communication overheads, Cloudlet [32] proposes the use of a nearby server instead of a cloud server that has higher latency and lower bandwidth. With Cloudlet, Native Offloader can reduce the communication latency. In addition, Rio [23] suggests a device driver for I/O sharing between mobile devices and optimizes remote I/O performance close to the local one. With Rio, Native Offloader can alleviate the remote I/O operation overheads.

Native Offloader uses static and dynamic performance estimation results for the compiler and the runtime to make offloading decisions. Narayanan et al. [33] and CMcloud [9, 31] use logging data and machine learning methods to predict the performance of mobile applications. Wolski et al. [34] and NWLSlite [35] propose bandwidth-aware performance prediction to count network costs. With these prediction algorithms, the Native Offloader compiler and runtime can predict the performance more precisely.

Native Offloader provides the UVA space that enables offloading tasks to share data across different architectures without a virtual machine. Distributed shared memory (DSM) systems [24, 25, 26, 27] provide the shared memory view across different platforms, but they cannot unify different memory layouts of the different architectures. With memory layout realignment, address size conversion, and endianness translation, Native Offloader does not only provide a shared memory view, but also unifies memory layouts across different architectures.

### Table 5: Comparison of computation offload systems

<table>
<thead>
<tr>
<th>System</th>
<th>Fully-Automatic</th>
<th>Offloading Decision</th>
<th>Requires VM Support</th>
<th>Target Language</th>
<th>Complexity of Target App.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo [7]</td>
<td>No (Manual)</td>
<td>Static</td>
<td>Yes</td>
<td>Java</td>
<td>Complex</td>
</tr>
<tr>
<td>Li et al. [10]</td>
<td>No (Manual)</td>
<td>Static</td>
<td>No</td>
<td>C</td>
<td>Simple</td>
</tr>
<tr>
<td>MAUI [4]</td>
<td>No (Annotation)</td>
<td>Dynamic</td>
<td>Yes</td>
<td>C#</td>
<td>Complex</td>
</tr>
<tr>
<td>Wang and Li [14]</td>
<td>No (Annotation)</td>
<td>Dynamic</td>
<td>No</td>
<td>C</td>
<td>Simple</td>
</tr>
<tr>
<td>DiET [13]</td>
<td>Yes</td>
<td>Static</td>
<td>Yes</td>
<td>Java</td>
<td>Simple</td>
</tr>
<tr>
<td>Chen et al. [1]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Simple</td>
</tr>
<tr>
<td>HELVM [12, 15]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Simple</td>
</tr>
<tr>
<td>OLIE [6, 11]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Complex</td>
</tr>
<tr>
<td>CloneCloud [3]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Complex</td>
</tr>
<tr>
<td>COMET [5]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Complex</td>
</tr>
<tr>
<td>CMcloud [9, 31]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Java</td>
<td>Complex</td>
</tr>
<tr>
<td>Native Offloader [This paper]</td>
<td>Yes</td>
<td>Dynamic</td>
<td>No</td>
<td>C</td>
<td>Complex</td>
</tr>
</tbody>
</table>

### 6. RELATED WORKS

Native Offloader is the first prototype framework for automatic cross-architecture computation offloading for general-purpose native application. With automatic architecture-aware partitioning and memory unification across different architectures such as ARM and x86, Native Offloader automatically transforms 17 native C applications from SPEC CPU2000 and CPU2006, and achieves a geometric mean whole-program speedup of 6.42× and battery saving of 82.0%.

### 7. CONCLUSION

Native Offloader is the first prototype framework for automatic cross-architecture computation offloading for general-purpose native application. With automatic architecture-aware partitioning and memory unification across different architectures such as ARM and x86, Native Offloader automatically transforms 17 native C applications from SPEC CPU2000 and CPU2006, and achieves a geometric mean whole-program speedup of 6.42× and battery saving of 82.0%.
8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and shepherd for their valuable feedback. We thank Bongjun Kim for helping our various evaluation. This material is partly supported by the Korean Government (MSIP) under the “ICT Consilience Creative Program” (IITP-2015-R0346-15-1007) and the "Basic Science Research Program" (NRF-2014R1A1A1002171). Gwangmu Lee is supported by Korea Foundation for Advanced Studies under the "Graduate Student Scholarship Program".

9. REFERENCES


