Architecture-aware Automatic Computation Offload for Native Applications

Gwangmu Lee Hyunjoon Park Seonyeong Heo Kyung-Ah Chang† Hyogun Lee† Hanjun Kim
POSTECH
Pohang, Korea
{iss300, chun92, heosy, hanjun}@postech.ac.kr
†Samsung Electronics
Suwon, Korea
{kachang, hglee}@samsung.com

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.

Difficulty Level	7	8	9	10	11
Desktop (sec)	0.06	0.50	1.11	2.23	11.38
Smartphone (sec)	0.34	2.92	6.33	12.79	66.02
Performance Gap (×)	5.36	5.89	5.71	5.74	5.80

Table 1: Movement computation time of the same chess game application on a smartphone and a desktop

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, 11, 12, 13, 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-

DOI: http://dx.doi.org/10.1145/2830772.2830833

Application	Version	Description	C/C++	Total	Ratio (LoC)	Runtime Description	Ratio (Exec. Time)
AdAway	3.0.2	AD blocker	132,882	310,321	42.82%	Read articles with ads	21.54%
Orbot	14.1.4-noPIE	Tor client	675,851	969,243	69.73%	Web browsing with Tor	61.98%
Firefox	40.0	Web browser	8,094,678	15,509,820	52.19%	Web browsing 4 websites	88.27%
VLC Player	1.5.1.1	Media player	3,584,526	6,433,726	55.71%	Play a movie w/ HW decoder Play a movie w/o HW decoder	23.05% 92.34%
Open Camera	1.2	Camera	0	10,336	0.00%	N/A	0.00%
osmAnd	2.1.1	Map/Navigation	53,695	450,573	11.92%	Search nearby places	23.86%
Syncthing	0.5.0-beta5	File synchronizer	0	59,461	0.00%	N/A	0.00%
AFWall+	1.3.4.1	Network traffic controller	1,514	59,741	2.53%	Web browsing 4 websites	0.30%
2048	1.95	Puzzle game	0	2,232	0.00%	N/A	0.00%
K-9 Mail	4.804	Email client	0	96,588	0.00%	N/A	0.00%
PDF Reader	0.4.0	PDF viewer	334,489	594,434	56.27%	Read a book with zoom	28.30%
ownCloud	1.5.8	File synchronizer	0	77,141	0.00%	N/A	0.00%
DAVdroid	0.6.2	Private data synchronizer	0	7,435	0.00%	N/A	0.00%
Barcode Scanner	4.7.0	2D/QR code scanner	0	50,201	0.00%	N/A	0.00%
SatStat	2	Sensor status monitor	0	7,480	0.00%	N/A	0.00%
Cool Reader	3.1.2-72	Ebook reader	491,556	681,001	72.18%	Read a book	97.73%
OS Monitor	3.4.1.0	OS monitor	5,902	74,513	7.92%	Read network and process info.	4.38%
Orweb	0.6.1	Web browser	0	14,124	0.00%	N/A	0.00%
PPSSPP	1.0.1.0	PSP emulator	1,304,973	1,438,322	90.73%	Play a game for 1 minute	97.68%
Adblock Plus	1.1.3	AD blocker	2,102	63,779	3.30%	Read articles with ads	22.83%

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.

pretation overheads of their VMs [19]. Due to the performance overhead, many developers implement computation-intensive parts as native codes with NDK libraries. Therefore, as Gordon et al. [5] point out, existing VM-based offloading systems cannot support lots of real-world computation intensive applications.

There are computation offloading systems [10, 14, 20, 21, 22] for native applications. These systems statically analyze mobile programs and make optimal partitions for mobile devices and servers. However, their applicability is limited mostly to well-analyzable simple applications such as media encoding and decoding programs that only have regular data access patterns. Table 2 shows that nowadays smartphone users use various kinds of mobile applications ranging from media players and games to web browsers, navigation, cloud service applications and emulators. Therefore, computation offloading systems need to offload general-purpose applications that are characterized by irregular data access patterns and complex control flow.

This paper is the first to demonstrate automatic computation offloading for general-purpose native applications, addressing the problems of different ISAs and distinct heterogeneous memory spaces across different architectures. This work has implemented a prototype framework for automatic computation offloading called Native Offloader, by combining an architecture-aware partitioning compiler and a seamless migration runtime. The Native Offloader compiler automatically finds machine independent heavy tasks from a native application without any annotation, inserts memory unification codes to overcome architectural differences such as memory layout, address size and endianness, and generates offloading-enabled native binaries for a mobile device and a server. The runtime provides a copy-on-demand sharing scheme between the mobile device and the server that allows data shared without an explicit communication instruction. With the memory unification codes and the copy-on-demand sharing scheme, Native Offloader provides the unified virtual address (UVA) space on distinct heterogeneous memory spaces of different architectures. For 17 native C applications from SPEC CPU2000 and CPU2006 benchmark suites, the Native Offloader framework achieves a geomean speedup of 6.42× and a geomean battery saving of 82.0% on an ARM mobile device with a x86 desktop server. This demonstrates that the architecture-aware memory unification makes automatic computation offloading for general-purpose native applications possible with low overheads.

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

2. DESIGN OF NATIVE OFFLOADER

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

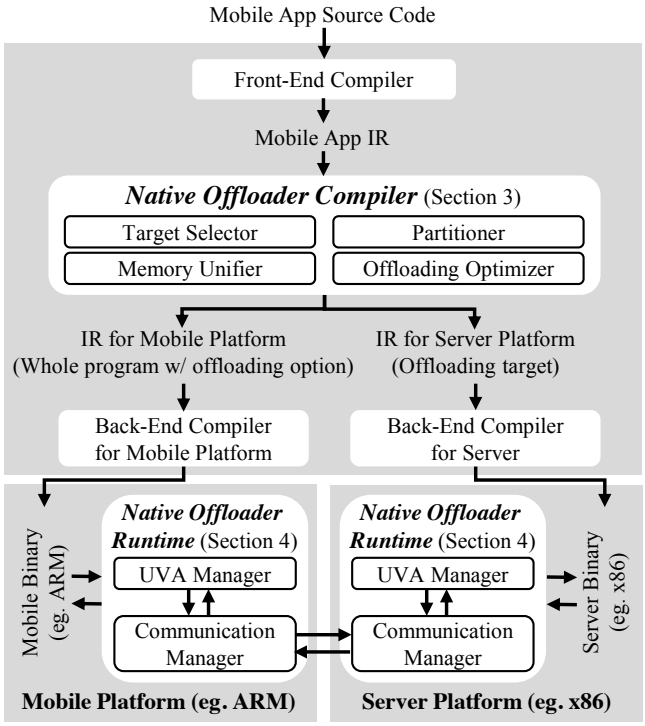


Figure 1: Structure of the Native Offloader framework

partitions the original application at IR level, and generates binaries for each target machine with various back-end compilers.

- Distinct memories: Due to distinct memory spaces, mobile devices and servers need to explicitly communicate shared objects. To efficiently share the objects without sophisticated static analysis and time-consuming address translation, Native Offloader provides a unified virtual address space across mobile devices and servers.

- Different memory layouts, address sizes and endiannesses: Though memory spaces are shared across mobile devices and servers, mobile devices and servers may not read the same value from the same memory address because they may have different memory layouts, address sizes and endiannesses. To overcome the architectural differences, Native Offloader unifies memory structures and inserts translation codes for memory operations.

Figure 1 illustrates the overall structure of Native Offloader. The Native Offloader compiler analyzes and transforms mobile applications at IR level. Front-end compilers transform various mobile applications to IR codes, the Native Offloader compiler partitions the IR codes into offloading-enabled IR codes for mobile devices and servers, and back-end compilers of each target machine compile the offloading-enabled IR codes to machine codes. Since IR codes are independent from source code languages and target machines, the IR level partitioning allows Native Offloader to easily enlarge its source language and target machine applicability.

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

(a) Original code

```
typedef struct {
 char from, to; double score;
} Move;
typedef struct {
 char loc, owner, type;
} Piece;
typedef double (*EVALFUNC) (Piece);

int maxDepth;
Piece *board;
EVALFUNC evals[7]={Pawn, ..., King};

int main () {


 ...
 scanf ("%d", &maxDepth);

 board = malloc(sizeof(Piece)*64);
 runGame ();
 ...
 return 0;
}

void runGame () {
 bool gameover = false;
 Move mv;
 while (!gameover) {
  mv = getPlayerTurn ();
  updateBoard (mv);









  mv = getAITurn ();

  updateBoard (mv);
  ...
 }
}

Move getAITurn () {
 Move mv;

 for (i=0; i < maxDepth; i++) {
  for (j=0; j < 64; j++) {
   ...
   char pieceType = board[j].type;
   EVALFUNC eval = evals[pieceType];


   mv.score += eval (board[j]);
   ...
  }

  printf ("%lf\n", mv.score);
 }
 return mv;
}


Move getPlayerTurn () {
 Move mv;
 scanf ("%d, %d", &mv.from, &mv.to);
 return mv;
}
```

(b) Partitioned code for mobile

```
// Mem. layout realignment (Sec. 3.2)
typedef struct {
 char from, to; char[6]; double score;
} Move_t;
typedef struct {
 char loc, owner, type;
} Piece;
typedef double (*EVALFUNC) (Piece);

// Global var realloc. (Sec. 3.2)
int *maxDepth_re;
Piece *board;
EVALFUNC evals[7]={Pawn, ..., King};

int main () {
 // Global var realloc. (Sec. 3.2)
 maxDepth_re = u_malloc(sizeof(int));
 ...
 scanf ("%d", maxDepth_re);
 // Unified heap management (Sec. 3.2)
 board = u_malloc(sizeof(Piece)*64);
 runGame ();
 ...
 return 0;
}

void runGame () {
 bool gameover = false;
 Move_t mv;
 while (!gameover) {
  mv = getPlayerTurn ();
  updateBoard (move);
  // Client partitioning (Sec. 3.3)
  if (isProfitable (getAITurn_id)) {
   requestOffload (getAITurn_id);
   sendData ();
   mv = receiveReturn ();
   receiveData ();
  } else {
   mv = getAITurn ();
  }
  updateBoard (mv);
  ...
 }
}

Move_t getAITurn () {
 Move_t mv;
 // Global var realloc. (Sec. 3.2)
 for (i=0; i < *maxDepth_re; i++) {
  for (j=0; j < 64; j++) {
   ...
   char pieceType = board[j].type;
   EVALFUNC eval = evals[pieceType];


   mv.score += eval (board[j]);
   ...
  }

  printf ("%lf\n", mv.score);
 }
 return mv;
}


Move_t getPlayerTurn () {
 Move_t mv;
 scanf ("%d,%d", &mv.from, &mv.to);
 return mv;
}
```

(c) Partitioned code for server

```
// Mem. layout realignment (Sec. 3.2)
typedef struct {
 char from, to; char[6]; double score;
} Move_t;
typedef struct {
 char loc, owner, type;
} Piece;
typedef double (*EVALFUNC) (Piece);

// Global var realloc. (Sec. 3.2)
int *maxDepth_re;
Piece *board;
EVALFUNC evals[7]={Pawn, ..., King};

int main () {




 // Stack change (Sec. 3.2)
 executeAtNewStack (listenClient);
}


// Server partitioning (Sec. 3.3)
void listenClient () {
 FcnID offID;
 while (true) {
  offID = acceptOffload ();
  if (!offID) break;
  receiveData ();
  switch (offID) {
  case getAITurn_id:
    Move_t ret = getAITurn ();
    sendReturn (ret);
    break;
  }
  sendData ();
 }
}



Move_t getAITurn () {
 Move_t mv;
 // Global var realloc. (Sec. 3.2)
 for (i=0; i < *maxDepth_re; i++) {
  for (j=0; j < 64; j++) {
   ...
   char pieceType = board[j].type;
   EVALFUNC eval = evals[piecetype];
   // Fcn ptr. converting (Sec. 3.4)
   eval = s2mFcnMap (eval);
   mv.score += eval (board[j]);
   ...
  }
  // Remote output (Sec. 3.4)
  r_printf ("%lf\n", mv.score);
 }
 return mv;
}

// Unused function removal (Sec. 3.3)
// Move getPlayerTurn ();
```

Figure 3: Simplified chess AI game code example

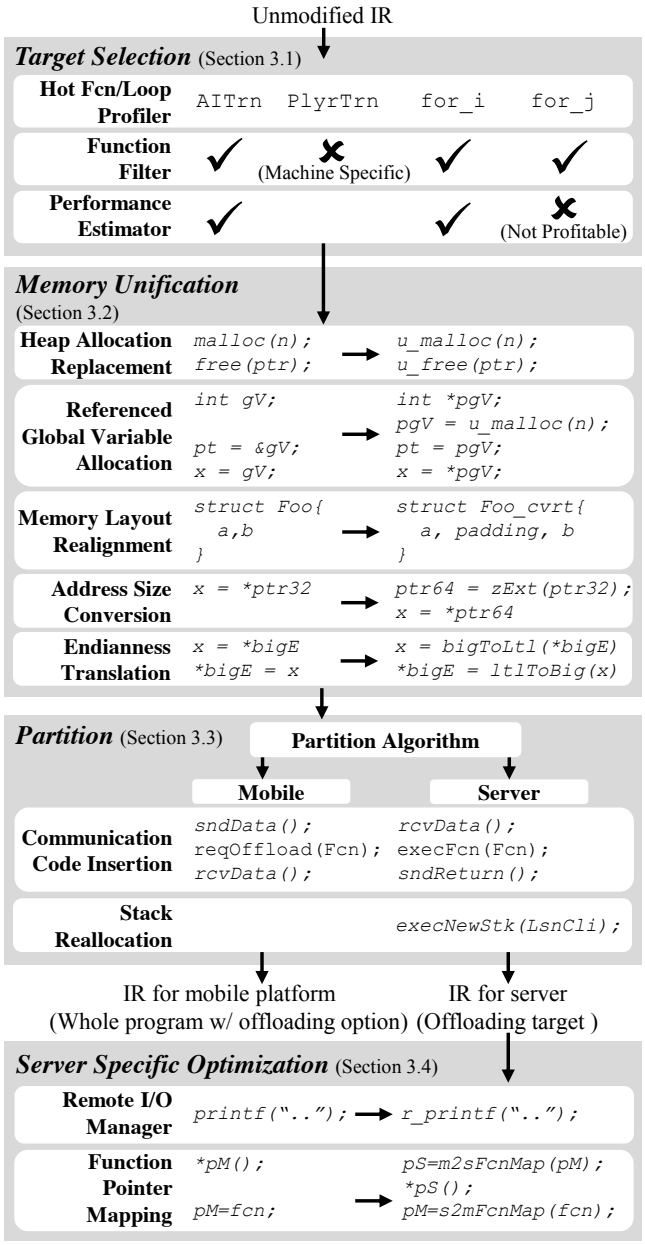


Figure 2: Structure of the Native Offloader compiler

filter considers an instruction machine specific if the instruction is one of the following instructions.

- Assembly instruction
- System call
- Unknown external library call
- I/O instruction

Assembly instructions are machine specific because they are written only for the target mobile device. Since system calls and unknown external library calls may cause side effects, the filter categorizes the instructions as machine specific instructions. I/O instructions use peripheral devices of a mobile device, so the I/O instructions are machine specific. Here, if the I/O functions are remotely executable through 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 `getAITurn` includes an output function call, `printf` that is one of the remote output functions, the filter classifies `getAITurn` as an offloading-enabled function.

Candidate	Profiling Results			Performance Estimation		
	Exec. Time	Invo. Cnt.	Mem. Size	T_{ideal} (sec)	T_c (sec)	T_g (sec)
runGame	27.0	1	20 MB	21.6	4.0	17.6
getAITurn	26.0	3	12 MB	20.8	7.2	13.6
for_i	26.0	3	12 MB	20.8	7.2	13.6
for_j	25.0	36	12 MB	20.0	86.4	-66.4
getPlayerTurn	1.5	3	10 MB	1.2	6.0	-4.8

Table 3: Profiling and performance estimation results of the chess game in Figure 3. The estimator assumes that the performance ratio (R) is 5 and the network bandwidth (BW) is 80Mbps.

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 $T_m * (1 - \frac{1}{R})$. 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 MMB memory and its network bandwidth is BW, the task requires $\frac{M}{BW}$ 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_{invo} 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.

$$\begin{aligned} T_g &= (T_m - T_s) - T_c \\ &= T_m * (1 - \frac{1}{R}) - 2 * \frac{M}{BW} * N_{invo} \end{aligned} \quad (1)$$

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 `getAITurn` and `for_i` as offloading targets. In Figure 3, the Native Offloader compiler offloads only `getAITurn` 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 at the UVA space using `u_malloc`, and transforms their uses to dereferenced instructions. 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 allocate 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 Native Offloader compiler inserts address size conversion codes that extend 32-bit pointers to 64-bit pointers for every memory access. Since the compiler inserts the conversion codes only when the target devices use different address sizes, the compiler does not apply the address size conversion if the targets use the same address size.

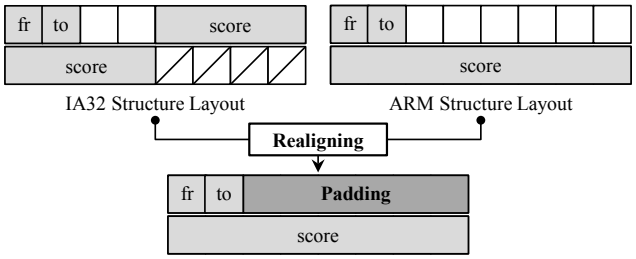


Figure 4: Memory layout realignment for type `Move` in Figure 3. Gray, dark gray, and slashed boxes represent meaningful data, paddings for memory layout realignment, and outside of the structure respectively.

Endianness translation: Though memory spaces and layouts are unified, a mobile device and a server may not read the same value from the same address due to different endianness. Like the address size conversion, the compiler inserts endianness translation codes for each memory access if the mobile device and the server have different endianness.

3.3 Partition

For the offloading targets, the Native Offloader compiler generates offloading-enabled IR codes for a mobile device and a server separately.

Partition for mobile device: To allow the mobile device dynamic offloading decision, the compiler inserts dynamic performance estimation codes, and generates target codes for both cases such as offloading execution and local execution. For the offloading execution, the compiler inserts communication codes to exchange shared data and target information. For the local execution, the compiler just calls the target as before. Figure 3(b) shows how the compiler transforms the original code (Line 33-41).

Partition for server: The compiler generates the server application codes that listen the offloading requests from the mobile device and execute the requested target. To manipulate different targets, the compiler inserts target function calls in `switch-case` statements with the target ID. Figure 3(c) shows the generated server application that manages offloading requests (Line 26-41). Here, the compiler finds and removes unused functions at server-side with a call graph. Figure 3(c) shows an unused function elimination example on the `getPlayerTurn` function (Line 66-67).

Stack reallocation: Since the mobile device and the server have the same virtual memory space, the server may corrupt the mobile stack memory if their stack areas are overlapped. To avoid this problem, the compiler changes the stack area of the server to be far from the mobile stack area before executing the offloading tasks (Line 22 in Figure 3(c)).

3.4 Server Specific Optimization

Remote I/O manager: Since most hot code regions include I/O operations such as reading files and printing re-

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.

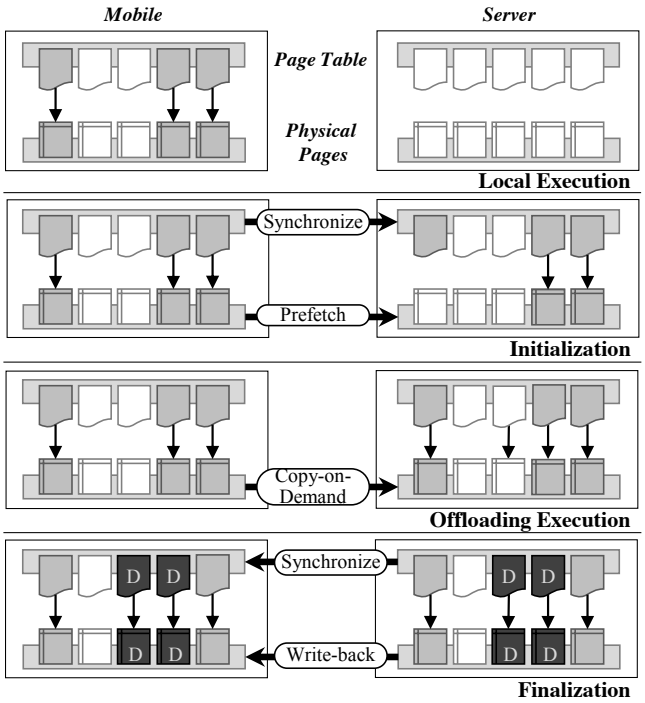


Figure 5: Life cycle of an offloaded task

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.

Program	Description	LoC	Exec. Time	Offloaded Function	Referenced GV.	Fcn. Ptr.	Target Function	Cover.	Inv.	Com. Traf.
164.gzip	Compression	5.5k	15.3	20 / 89	141 / 241	9	spec_compress	98.90	1	151.5
175.vpr	FPGA Simultion	11.3k	26.9	9 / 272	672 / 760	3	try_place_while.cond	99.07	1	0.8
177.mesa	3-D Graphic	42.2k	120.2	11 / 1105	608 / 627	1169	Render	99.02	1	20.3
179.art	Image Recognition	5.7k	325.5	7 / 26	52 / 79	0	scan_recognize	85.44	1	16.4
183.equake	Seismic Wave Propagation	1.0k	334.0	5 / 28	83 / 104	0	main_for.cond548	99.44	1	16.5
188.ammp	Computational Chemistry	9.8k	878.0	17 / 179	324 / 333	66	AMMPmonitor	13.53	2	17.0
							tpac	85.60	1	17.6
300.twolf	Place/Route Simulator	17.8k	157.8	3 / 191	566 / 838	0	utemp	99.84	1	3.3
401.bzip2	Compression	5.7k	27.0	58 / 100	95 / 120	24	spec_compress	98.79	1	134.3
429.mcf	Vehicle Scheduling	1.6k	104.8	19 / 24	39 / 43	0	global_opt	99.55	1	47.9
433.milc	Quantum Chromodynamics	9.6k	365.8	61 / 235	445 / 493	6	update	96.21	2	13.4
445.gobmk	Go Game	156.3k	361.8	6 / 2679	21844 / 22090	77	gtp_main_loop	99.96	1	25.7
456.hmmer	Gene Sequence	20.6k	31.3	36 / 538	995 / 1050	36	main_loop_serial	99.99	1	0.3
458.sjeng	Chess Game	10.5k	950.8	91 / 144	495 / 624	1	think	99.95	3	240.2
462.libquantum	Quantum Computing	2.6k	71.0	62 / 116	0 / 44	0	quantum_exp_mod_n	92.56	1	6.3
464.h264ref	Video Encoder	59.5k	78.2	48 / 1333	2012 / 2822	457	encode_sequence	99.79	1	17.1
470.lbm	Fluid Dynamics	0.9k	1444.9	1 / 19	16 / 20	0	main_for.cond	99.70	1	643.6
482.sphinx3	Speech Recognition	13.1k	375.2	124 / 370	1265 / 1329	14	main_for.cond	98.39	1	34.0

Table 4: Details of offloaded programs, including lines of code; the total execution time (sec) on the smartphone with the evaluation input; numbers of offloaded functions among all the functions, referenced global variables among all the global variables, and function pointer uses; and offloaded targets, their coverage (%), the number of invocations and communication traffic per invocation (MB).

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 `197.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 multithreading 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`.

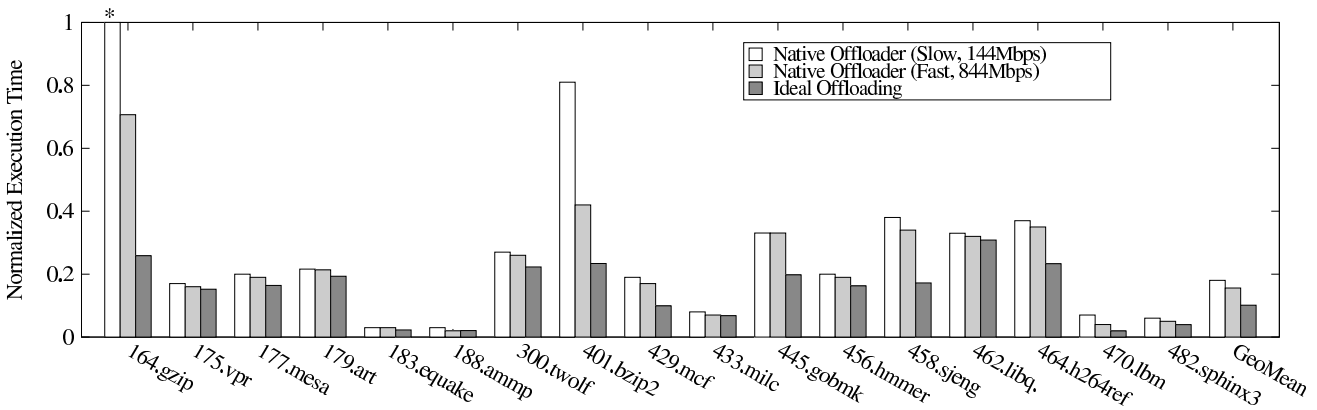
Figure 6 presents the whole program execution time and battery consumption of the offloaded applications normalized to local execution time and battery consumption on the smartphone. In each graph, the x-axis shows evaluated applications and the y-axis shows the normalized execution time and battery 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

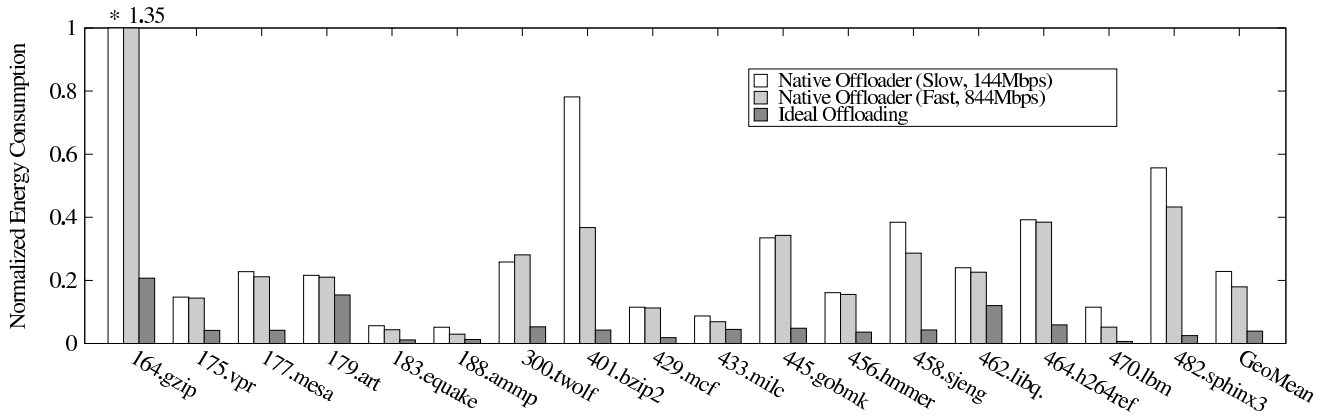
Figure 6(a) shows that Native Offloader achieves performance speedups for all the evaluated programs in slow and fast wireless environments, and reduces 82.0% and 84.4% of program execution time on geomean of program execution time. Ideal offloading means execution time without any overhead such as data communication and translation.

For `175.vpr`, `179.art`, `183.equake`, `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



(a) Normalized execution time



(b) Normalized battery consumption

Figure 6: Execution time and battery consumption normalized to local execution in different network environments. * means non-offloaded by the dynamic performance estimation.

`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 environments such as bandwidth and latency affect the communication 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 because the mobile device and the server use the same endianness, little-endian.

`164.gzip`, `401.bzip2`, `429.mcf`, `458.sjeng` and `470.lbm` have a huge amount of communication compared to their execution time. Since the communication overhead increases in the slow network, the programs are very sensitive to the network bandwidth. Therefore, for the slow network connection, though the Native Offloader compiler generates offloading-enabled codes, the dynamic performance estimator in the Native Offloader runtime decides not to offload the offloading target. For example, Native Offloader does not offload `spec_compress` from `164.gzip` according to Equation 1. The dynamic performance estimation 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 offloading execution, `300.twolf` reads a file about cell information 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 programs execute remote input operations that require expensive 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 function pointers. `445.gobmk` and `458.sjeng` have function pointer arrays such as `commands` and `evalRoutines` to manage next commands and piece movements respectively.

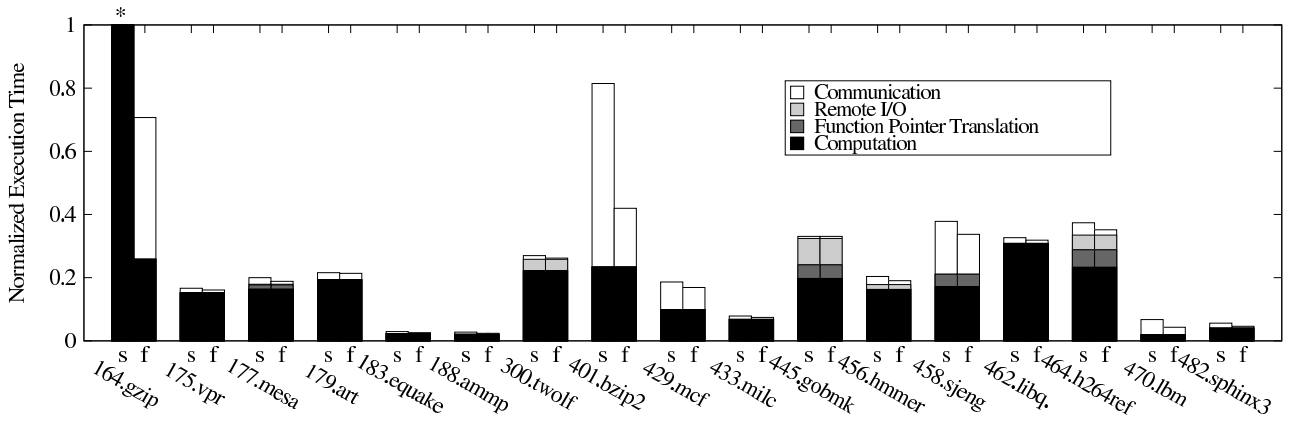
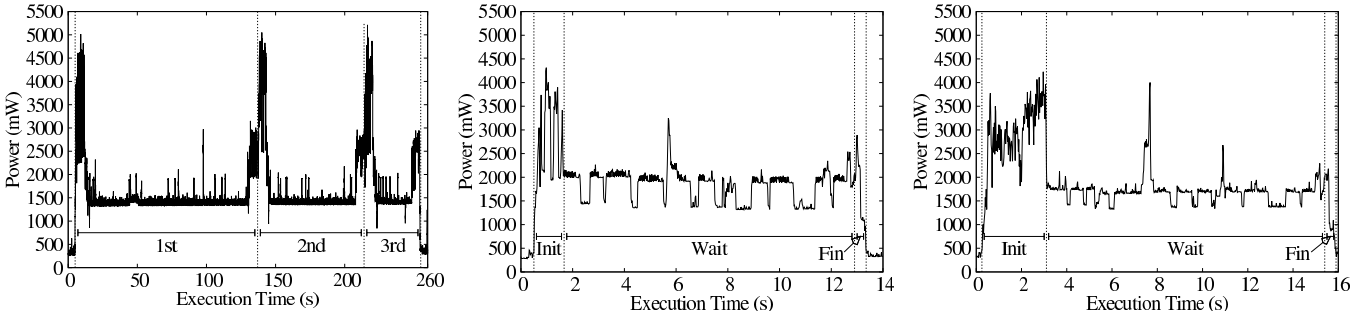


Figure 7: Breakdown of overheads. s and f mean slow and fast networks.



(a) `458.sjeng` (fast network) (b) `445.gobmk` (fast network) (c) `445.gobmk` (slow network)

Figure 8: Power consumption over time for `458.sjeng` and `445.gobmk`

`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 power more 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.

System	Fully-Automatic	Offloading Decision	Requires VM Support	Target Language	Complexity of Target App.
Cuckoo [7]	No (Manual)	Static	Yes	Java	Complex
Li et al. [10]	No (Manual)	Static	No	C	Simple
Roam [2]	No (Manual)	Dynamic	Yes	Java	Complex
MAUI [4]	No (Annotation)	Dynamic	Yes	C#	Complex
ThinkAir [8]	No (Annotation)	Dynamic	Yes	Java	Complex
Wang and Li [14]	No (Annotation)	Dynamic	No	C	Simple
DiET [13]	Yes	Static	Yes	Java	Simple
Chen et al. [1]	Yes	Dynamic	Yes	Java	Simple
HELVM [12, 15]	Yes	Dynamic	Yes	Java	Simple
OLIE [6, 11]	Yes	Dynamic	Yes	Java	Complex
CloneCloud [3]	Yes	Dynamic	Yes	Java	Complex
COMET [5]	Yes	Dynamic	Yes	Java	Complex
CMcloud [9, 31]	Yes	Dynamic	Yes	Java	Complex
Native Offloader [This paper]	Yes	Dynamic	No	C	Complex

Table 5: Comparison of compuation offload systems

6. RELATED WORKS

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 communication 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 across different platforms.

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 el al. [34] and NWSLite [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.

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 geomean 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

[1] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. Irwin, and R. Chandramouli, "Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[2] H.-h. Chu, H. Song, C. Wong, S. Kurakake, and M. Katagiri, "Roam, a seamless application framework," *Journal of Systems and Software*, 2004.

[3] 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*, 2011.

[4] 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*, 2010.

[5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[6] 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*, 2003.

[7] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*, 2012.

[8] 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 (IEEE INFOCOM)*, 2012.

[9] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[10] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: A partition scheme," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[11] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002.

[12] 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*, 2006.

[13] H. Rim, S. Kim, Y. Kim, and H. Han, "Transparent method offloading for slim execution," in *1st International Symposium on Wireless Pervasive Computing*, 2006.

[14] C. Wang and Z. Li, "Parametric analysis for adaptive computation offloading," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[15] K. Yang, S. Ou, and H.-H. Chen, "On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications," *Communications Magazine, IEEE*, 2008.

[16] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing," *IEEE Transactions on Mobile Computing*, 2014.

[17] "F-Droid Official Homepage." http://f-droid.org.

[18] "F-Droid Statistics for Official Repo." http://android.woju.eu/stats.

[19] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[20] U. Kremer, J. Hicks, and J. Rehg, "A compilation framework for power and energy management on mobile computers," in *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, 2003.

[21] Z. Li, C. Wang, and R. Xu, "Task allocation for distributed multimedia processing on wirelessly networked handheld devices," in *Proceedings of the Interational Parallel and Distributed Processing Symposium*, 2002.

[22] 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*, 2007.

[23] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing I/O between mobile systems," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[24] Y. Aridor, M. Factor, and A. Teperman, "cJVM: A single system image of a JVM on a cluster," in *Proceedings of the International Conference on Parallel Processing*, 1999.

[25] J. B. Carter, "Design of the Munin Distributed Shared Memory System," *Journal of Parallel and Distributed Computing*, 1995.

[26] D. J. Scales and K. Gharachorloo, "Towards transparent and efficient software distributed shared memory," in *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[27] W. Zhu, C.-L. Wang, and F. Lau, "JESSICA2: a distributed java virtual machine with transparent thread migration support," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[28] "Monsoon power monitor." http://www.msoon.com/LabEquipment/PowerMonitor/.

[29] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[30] "Standard Performance Evaluation Corporation." http://www.spec.org.

[31] D. Chae, J. Kim, J. Kim, J. Kim, S. Yang, Y. Cho, Y. Kwon, and Y. Paek, "CMcloud: Cloud platform for cost-effective offloading of mobile applications," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[32] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, 2009.

[33] D. Narayanan, J. Flinn, and M. Satyanarayanan, "Using history to improve mobile application adaptation," in *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[34] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi, "Using bandwidth data to make computation offloading decisions," in *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[35] S. Gurun, C. Krintz, and R. Wolski, "NWSLite: A light-weight prediction utility for mobile devices," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, 2004.