CloneCloud: Elastic Execution between Mobile Device and Cloud

Byung-Gon Chun
Intel Labs Berkeley
byung-gon.chun@intel.com

Sunghwan Ihm
Princeton University
sihm@cs.princeton.edu

Petros Maniatis
Intel Labs Berkeley
petros.maniatis@intel.com

Mayur Naik
Intel Labs Berkeley
mayur.naik@intel.com

Ashwin Patti
Intel Labs Berkeley
ashwin.patti@intel.com

Abstract

Mobile applications are becoming increasingly ubiquitous and provide ever richer functionality on mobile devices. At the same time, such devices often enjoy strong connectivity with more powerful machines ranging from laptops and desktops to commercial clouds. This paper presents the design and implementation of CloneCloud, a system that automatically transforms mobile applications to benefit from the cloud. The system is a flexible application partitioner and execution runtime that enables unmodified mobile applications running in an application-level virtual machine to seamlessly off-load part of their execution from mobile devices onto device clones operating in a computational cloud. CloneCloud uses a combination of static analysis and dynamic profiling to partition applications automatically at a fine granularity while optimizing execution time and energy use for a target computation and communication environment. At runtime, the application partitioning is effected by migrating a thread from the mobile device at a chosen point to the clone in the cloud, executing there for the remainder of the partition, and re-integrating the migrated thread back to the mobile device. Our evaluation shows that CloneCloud can adapt application partitioning to different environments, and can help some applications achieve as much as a 20x execution speed-up and a 20-fold decrease of energy spent on the mobile device.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Client/server

General Terms Algorithms, Design, Experimentation, Performance

EuroSys'11 April 10–13, 2011, Salzburg, Austria.

Reprinted from EuroSys'11, Proceedings, April 10–13, 2011, Salzburg, Austria., pp. 181–194.

Keywords Mobile cloud computing, partitioning, offloading, migration, smartphones

1. Introduction

Mobile cloud computing is the next big thing. Recent market research predicts that by the end of 2014 mobile cloud applications will deliver annual revenues of 20 billion dollars [Beccue 2009]. Although it is hard to validate precise predictions, this is hardly implausible: mobile devices as simple as phones and as complex as mobile Internet devices with various network connections, strong connectivity especially in developed areas, camera(s), GPS, and other sensors are the current computing wave, competing heavily with desktops and laptops for market and popularity. Connectivity offers immediate access to available computing, storage, and communications on commercial clouds, at nearby wireless hot-spots equipped with computational resources [Satyanarayanan 2009], or at the user's PC and plugged-in laptop.

This abundance of cloud resources and the mobile opportunity to use them is met by the blinding variety of flash-popular applications in application stores by Apple, Google, Microsoft, and others. Now mobile users look up songs by audio samples; play games; capture, edit, and upload video; analyze, index, and aggregate their mobile photo collections; analyze their finances; and manage their personal health and wellness. Also, new rich media, mobile augmented reality, and data analytics applications change how mobile users remember, experience, and understand the world around them. Such applications recruit increasing amounts of computation, storage, and communications from a constrained supply on mobile devices—certainly compared to tethered, wall-socket-powered devices like desktops and laptops—and place demands on an extremely limited supply of energy.

Yet bringing a demanding mobile application to needed cloud resources tends to be inflexible: an application is either written as a monolithic process, cramming all it needs to do

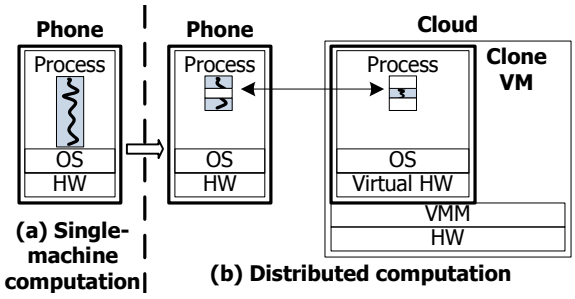

Figure 1. CloneCloud system model. CloneCloud transforms a single-machine execution (mobile device computation) into a distributed execution (mobile device and cloud computation) automatically.

on to the mobile device; or it is split in the traditional client-server paradigm, pushing most computation to the remote server; or it is perhaps tailored to match an expected combination of client (e.g., given browser on particular phone platform), environment (a carrier's expected network conditions), and service. But what might be the right split for a low-end mobile device with good connectivity may be the wrong split for a high-end mobile device with intermittent connectivity. Often the choice is unknown to application developers ahead of time, or the possible configurations are too numerous to customize for all of them.

To address this problem, in this paper we realize our CloneCloud vision [Chun 2009] of a flexible architecture for the seamless use of ambient computation to augment mobile device applications, making them fast and energy-efficient. CloneCloud boosts *unmodified* mobile applications by off-loading the *right* portion of their execution onto *device clones* operating in a computational cloud[1]. Conceptually, our system automatically transforms a single-machine execution (e.g., computation on a smartphone) into a distributed execution optimized for the network connection to the cloud, the processing capabilities of the device and cloud, and the application's computing patterns (Figure 1).

The underlying motivation for CloneCloud lies in the following intuition: as long as execution on the cloud is significantly faster than execution on the mobile device (or more reliable, more secure, etc.), paying the cost for sending the relevant data and code from the device to the cloud and back may be worth it. Unlike partitioning a service statically by design between client and server portions, CloneCloud late-binds this design decision. In practice, the partitioning decision may be more fine-grained than a yes/no answer (i.e., it may result in carving off different *amounts* of the original application for cloud execution), depending on the expected workload and execution conditions (CPU speeds, network performance). A fundamental design goal for CloneCloud is to allow such fine-grained flexibility on what to run where.

[1] Throughout this paper, we use the term "cloud" in a broad sense to include diverse ambient computational resources discussed above.

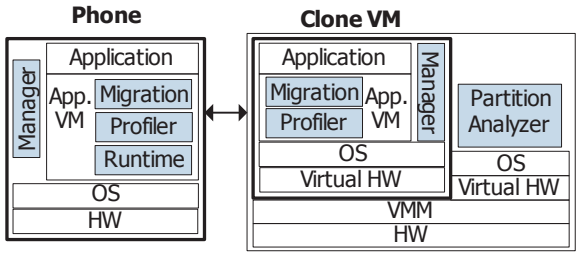

Figure 2. The CloneCloud prototype architecture.

Another design goal for CloneCloud is to take the programmer out of the business of application partitioning. The kinds of applications on mobile platforms that are featured on application stores and gain flash popularity tend to be low-margin products, whose developers have little incentive to optimize manually for different combinations of architectures, network conditions, battery lives, and hosting infrastructures. Consequently, CloneCloud aims to make application partitioning automatic and seamless.

Our work in this paper applies primarily to application-layer virtual machines (VMs), such as the Java VM, DalvikVM from the Android Platform, and Microsoft's .NET. We choose application-layer VMs since they are widely used in mobile platforms. In addition, the application-layer VM model has the relative ease of manipulating application executables and migrating pieces thereof to computing devices of diverging architectures, even different instruction set architectures (e.g., ARM-based smartphones and x86-based servers).

The CloneCloud prototype meets all design goals mentioned above, by rewriting an unmodified application executable. While the modified executable is running, at automatically chosen points individual threads migrate from the mobile device to a device clone in a cloud; remaining functionality on the mobile device keeps executing, but blocks if it attempts to access migrated state, thereby exhibiting opportunistic but very conservative concurrency. The migrated thread executes on the clone, possibly accessing native features of the hosting platform such as the fast CPU, network, hardware accelerators, storage, etc. Eventually, the thread returns back to the mobile device, along with remotely created state, which it merges back into the original process. The choice of where to migrate is made by a partitioning component, which uses static analysis to discover constraints on possible migration points, and dynamic profiling to build a cost model for execution and migration. A mathematical optimizer chooses migration points that optimize objective (such as total execution time or mobile-device energy consumption) given the application and the cost model. Finally, the run-time system chooses what partition to use. Figure 2 shows the high-level architecture of our prototype.

The paradigm of opportunistic use of ambient resources is not new [Balan 2002]; much research has attacked appli-

cation partitioning and migration in the past (see Section 7). CloneCloud is built upon existing technologies, but it combines and augments them in a novel way. We summarize our contributions here as follows.

- Unlike traditional suspend-migrate-resume mechanisms [Satyanarayanan 2005] for application migration, the CloneCloud migrator operates at thread granularity, an essential consideration for mobile applications, which tend to have features that must remain at the mobile device, such as those accessing the camera or managing the user interface.
- Unlike past application-layer VM migrators [Aridor 1999, Zhu 2002], the CloneCloud migrator allows native system operations to execute both at the mobile device and at its clones in the cloud, harnessing not only raw CPU cloud power, but also system facilities or specialized hardware when the underlying library and OS are implemented to exploit them.
- Similar to MAUI [Cuervo 2010], the CloneCloud partitioner automatically identifies costs through static and dynamic code analysis and runs an optimizer to solve partitioning problems, but we go a step further by not asking for the programmer's help (e.g., source annotations).
- We present the design, implementation, and evaluation of an operational system that combines the features in widely-used Android platforms. CloneCloud can achieve up to 20x speedup and 20x less energy consumption of smartphone applications we tested.

In what follows, we first give some brief background on application-layer VMs (Section 2). We then present the design of CloneCloud's partitioning components (Section 3) and its distributed execution mechanism (Section 4). We describe our implementation (Section 5) and experimental evaluation of the prototype (Section 6). We survey related work in Section 7, discuss limitations and future work in Section 8, and conclude in Section 9.

2. Background: Application VMs

An application-level VM is an abstract computing machine that provides hardware and operating system independence. Its instruction sets are platform-independent bytecodes; an executable is a blob of bytecodes. The VM runtime executes bytecodes of methods with threads. There is typically a separation between the *virtual* portion of an execution and the *native* portion; the former is only expressed in terms of objects directly visible to the bytecode, while the latter includes management machinery for the virtual machine, data and computation invoked on behalf of a virtual computation, as well as the process-level data of the OS process containing the VM. Interfacing between the virtual and the native portion happens via native interface frameworks.

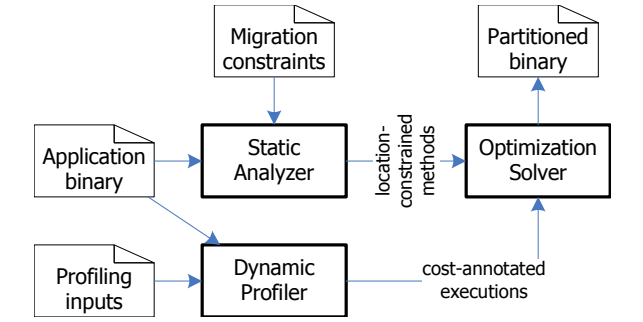

Figure 3. Partitioning analysis framework.

Runtime memory is split between VM-wide and per-thread areas. The *Method Area*, which contains the types of the executing program and libraries as well as static variable contents, and the *Heap*, which holds all dynamically allocated data, are VM-wide. Each thread has its own *Virtual Stack* (stack frames of the virtual hardware), the *Virtual Registers* (e.g., the program counter), and the *Native Stack* (containing any native execution frames of a thread, if it has invoked native functions). Most computation, data structure manipulation, and memory management are done within the abstract machine. However, external processing such as file I/O, networking, using local hardware such as sensors, are done via APIs that punch through the abstract machine into the process's system call interface.

3. Partitioning

The partitioning mechanism in CloneCloud is off-line, and aims to pick which parts of an application's execution to retain on the mobile device and which to migrate to the cloud. Any application targeting the application VM platform may be partitioned; unlike prior approaches, including the recent MAUI project [Cuervo 2010], the programmer need not write the application in a special idiom or annotate it in a non-standard way, and the source code is not needed. The output of the partitioning mechanism is a *partition*, a choice of execution points where the application migrates part of its execution and state between the device and a clone. Given a set of execution conditions (we currently consider network characteristics, CPU speeds, and energy consumption), the partitioning mechanism yields a partition that optimizes for total execution time or energy expended at the mobile device. The partitioning mechanism may be run multiple times for different execution conditions and objective functions, resulting in a database of partitions. At runtime, the distributed execution mechanism (Section 4) picks a partition from the database and implements it via a small and fast set of modifications of the executable before invocation.

Partitioning of an application operates according to the conceptual workflow of Figure 3. Our partitioning framework combines static program analysis with dynamic program profiling to produce a partition.

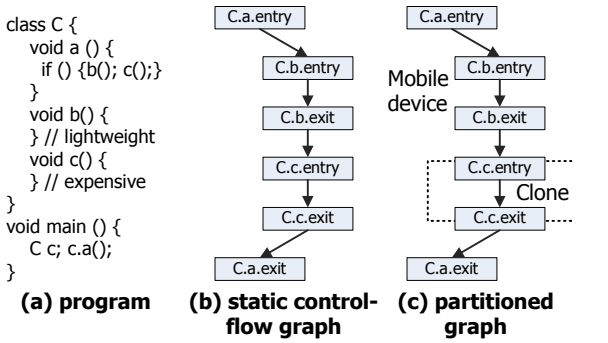

Figure 4. An example of a program, its corresponding static control-flow graph, and a partition.

The *Static Analyzer* identifies legal partitions of the application executable, according to a set of constraints (Section 3.1). Constraints codify the needs of the distributed execution engine, as well as the usage model. The *Dynamic Profiler* (Section 3.2) profiles the input executable on different platforms (the mobile device and on the cloud clone) with a set of inputs, and returns a set of profiled executions. Profiled executions are used to compose a cost model for the application under different partitions. Finally, the *Optimization Solver* finds a legal partition among those enabled by the static analyzer that minimizes an objective function, using the cost model derived by the profiler (Section 3.3). The resulting partition is stored as a configuration file. At runtime, the chosen partition drives the execution of the application.

3.1 Static Analyzer

The partitioner uses static analysis to identify legal choices for placing migration and re-integration points in the code. In principle, these points could be placed anywhere in the code, but we reduce the available choices to make the optimization problem tractable. In particular, we restrict migration and re-integration points to method entry and exit points, respectively. We make two additional restrictions for simplicity. First, we only allow migration at the boundaries of application methods, not core-system library methods, which simplifies the implementation of a partition at runtime. Second, we only allow migration at VM-layer method boundaries, not native method boundaries, since the techniques required to migrate partial execution state differ vastly for the two types of methods. Note, however, that although we disallow migration while already in native execution, we do allow migrated methods to invoke native ones.

Figure 4 shows an example of a program, relevant parts of its static control-flow graph, and a particular legal partition of the program. Class `C` has three methods. Method `a()` calls method `b()`, which performs lightweight processing, followed by method `c()`, which performs expensive processing. The static control-flow graph approximates control flow in the program (inferring exact control flow is undecidable as program reachability is undecidable). The approximation is conservative in that if an execution of the program follows a certain path then that path exists in the graph (but the converse typically does not hold). In the depicted static control-flow graph, only entry and exit nodes of methods are shown, labeled as <class name>.<method name>.<entry | exit>. A possible partition as shown in Figure 4(c) runs the body of method `c()` on the clone, and the rest of the program on the mobile device. As described above, method `c()` may not be a system library or a native method, but may itself invoke system libraries or native methods.

3.1.1 Constraints

We next describe three properties of any legal partition, as required by the migration component, and explain how we use static analysis to obtain constraints that express these properties.

PROPERTY 1. *Methods that access specific features of a machine must be pinned to the machine.*

If a method uses a local resource such as the location service (e.g., GPS) or sensor inputs (e.g., microphones) in a mobile device, the method must be executed on the mobile device. This primarily concerns native methods, but also the main method of a program. The analysis marks the declaration of such methods with a special annotation M—for Mobile device. We manually identify such methods in the VM's API (e.g., VM API methods explicitly referring to the camera); this is done once for a given platform and is not repeated for each application. We also always mark the main method of a program. We refer to methods marked with M as the V_M method set.

PROPERTY 2. *Methods that share native state must be collocated at the same machine.*

An application may have native methods that create and access state below the VM. Native methods may share native state. Such methods must be collocated at the same machine as our migration component does not migrate native state (Section 4.1). For example, when an image processing class has `initialize, detect, fetchresult` methods that access native state, they need to be collocated at the same machine. To avoid a manual-annotation burden, native state annotations are inferred automatically by the following simple approximation, which works well in practice: we assign a unique annotation Nat_C to all native methods declared in the same class C; the set V_{Nat_C} contains all methods with that annotation.

PROPERTY 3. *Prevent nested migration.*

With one phone and one clone, this implies that there should be no nested suspends and no nested resumes. Once a program is suspended for migration at the entry point of a method, the program should not be suspended again without a resume, i.e., migration and re-integration points must be

executed alternately. To enforce this property, the static analysis builds the static control-flow graph of an application, capturing the caller-callee method relation; it exports this as two relations, $DC(m_1, m_2)$, read as "method m_1 Directly Calls method m_2," and $TC(m_1, m_2)$ read as "method m_1 Transitively Calls method m_2," which is the transitive closure of DC. For the example in Figure 4, this ensures that if partitioning points are placed in `a()`, they are not placed in `b()` or `c()`. The remaining legal partitions place migration points at at `b()`, at `c()`, or at both `b()` and `c()`.

3.2 Dynamic Profiler

The profiler collects the data that will be used to construct a cost model for the application under different execution settings. The cost metric can vary, but our prototype uses execution time and energy consumed at the mobile device.

The profiler is invoked on multiple executions of the application, each using a randomly chosen set of input data (e.g., command-line arguments and user-interface events), and each executed once on the mobile device and once on the clone in the cloud. The profiler outputs a set S of executions, and for each execution a *profile tree* T and T', from the mobile device and the clone, respectively. We note that random inputs may not explore all relevant execution paths of the application. In our future work, we hope to explore symbolic-execution-based techniques for high-coverage input generation [Cadar 2008].

A profile tree is a compact representation of an execution on a single platform. It is a tree with one node for each method invocation in the execution; it is rooted at the starting (user-defined) method invocation of the application (e.g., `main`). Specific method calls in the execution are represented as edges from the node of the caller method invocation (parent) to the nodes of the callees (children); edge order is not important. Each node is annotated with the cost of its particular invocation in the cost metric (execution time in our case). In addition to its called-method children, every non-leaf node also has a leaf child called its *residual node*. The residual node i' for node i represents the residual cost of invocation i that is not due to the calls invoked within i; in other words, node i' represents the cost of running the body of code excluding the costs of the methods called by it. Finally, each edge is annotated with the state size at the time of invocation of the child node, plus the state size at the end of that invocation; this would be the amount of data that the migrator (Section 4.1) would need to capture and transmit in both directions, if the edge were to be a migration point. Edges between a node and its residual child have no cost.

Figure 5 is an example of an execution trace and its corresponding profile tree. `a` is called twice in `main`, one `a` call invoking `b` and `c`, and one `a` call invoking no other method. A tree node on the right holds the execution time of the corresponding method in the trace (the length of the square bracket on the left). `main'` and `a'` are residual nodes, and they hold the difference between the value of

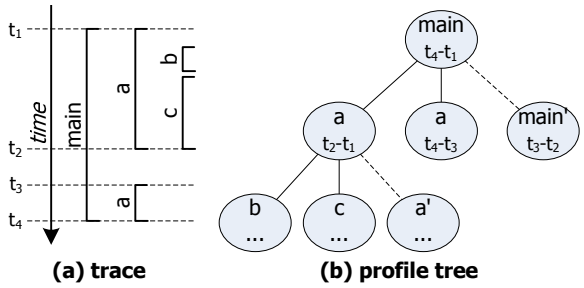

Figure 5. An example of an execution trace (a) and its corresponding profile tree (b). Edge costs are not shown.

their parent node and the sum of their sibling nodes. For example, node `main'` holds the value $T[\texttt{main'}] \equiv t_3 - t_2 = (t_4 - t_1) - ((t_4 - t_3) + (t_2 - t_1))$.

To fill in profile trees, we temporarily instrument application-method entry and exit points during each profile run on each platform (recall that system-library and native methods are not valid partitioning points, so we do not instrument them). For our execution-time cost metric, we collect timings at method entry and exit points, which we process to fill in tree node annotations. We compute migration costs (edge weights) by simulating migration at each profiled method: we perform the suspend-and-capture operation of the migrator (Section 4.1) and measure the captured state size, both when invoking the child node and when returning from it; we set the annotation of the parent edge $TE[i]$ of invocation i with that value. Recall that for every execution E, we capture two profile trees T and T', one per platform with different annotations.

For each invocation i in profiling execution E, we define a computation cost $C_c(i, l)$ and a migration cost $C_s(i)$, where l is the location of the invocation. We start with execution time cost. We fill in $C_c(i, l)$ from the corresponding profile tree collected at location l (if $l = 0$, the location is the mobile device and is filled from tree T, and if $l = 1$, the location is the clone and is filled from tree T'). If i is a leaf profile tree node, we set $C_c(i, l)$ to be the annotation of that node (e.g., $C_c(i, 0) \equiv T[i]$); otherwise, we set it to the annotation of the residual node i'. We fill $C_s(i)$ as the cost of making invocation i a migrant invocation. This cost is the sum of a suspend/resume cost and a transfer cost. The former is the time required to suspend a thread and resume a thread. The latter is a volume-dependent cost, the time it takes to capture, serialize, transmit, deserialize, and reinstantiate state of a particular size (assuming for simplicity all objects have the same such cost per byte). We precompute this per-byte cost[2], and use the edge annotations from the mobile-device profile tree to calculate the cost.

[2] One could also estimate this per-byte cost from memory, processor, and storage speeds, as well as network latency and bandwidth, but we took the simpler approach of just measuring it.

For energy consumption, we use a simple model that appears consistent with the kinds of energy measurements we can perform with off-board equipment (the Monsoon power monitor [Mon], in our setup). Specifically, we use a model that maps three system variables to a power level. We consider CPU activity (processing/idle), display state (on/off), and network state (transmitting or receiving/idle), and translate them to a power draw via a function P from $\langle CPU, Scr, Net\rangle$ triples to a power value. We estimate function P experimentally, and use it to construct two cost models, one where screen is on, and one where screen is off, as follows. For the cost model with the screen on, we set all $C_c(i,0) \equiv P(CPUOn, ScrOn, NetIdle) \times T[i]$, i.e., the execution time at the device at power levels consistent with high CPU utilization, display on, but not network activity. We set all $C_c(i,1) \equiv P(CPUIdle, ScrOn, NetIdle)$, i.e., the execution time at the clone, but at power levels consistent with idle CPU at the device; recall that we do not care about energy at the clone, but about energy expended at the mobile device while the clone is processing. Finally, we set all $C_s(i)$ from the execution-time model $C_s(i)$'s above, which hold the time it takes to migrate for an invocation, multiplied by power $P(CPUOn, ScrOn, NetOn)$. We note that our energy consumption model is a coarse starting point, with some noise, especially for very close decisions to migrate or not (see Section 6).

3.3 Optimization Solver

The purpose of our optimizer is to pick which application methods to migrate to the clone from the mobile device, so as to minimize the expected cost of the partitioned application. Given a particular execution E and its two profile trees T on the mobile device and T' on the clone, one might intuitively picture this task as optimally replacing annotations in T with those in T', so as to minimize the total node and weight cost of the hybrid profile tree. Our static analysis dictates the legal ways to fetch annotations from T' into T, and our dynamic profiling dictates the actual trees T and T'. We do not differentiate among different executions E in the execution set S; we consider them all equiprobable, although one might assign non-uniform frequencies in practice to match a particular expected workload.

More specifically, the output of our optimizer is a value assignment to binary decision variables $R(m)$, where m is every method in the application. If the optimizer chooses $R(m) = 1$ then the partitioner will place a migration point at the entry into the method, and a re-integration point at the exit from the method. If the optimizer chooses $R(m) = 0$, method m is unmodified in the application binary. For simplicity and to constrain the optimization problem, our migration strategy chooses to migrate or not migrate *all invocations* of a method. Despite its simplicity, this conservative strategy provides us with undeniable benefits (Section 6); we leave further refining differentiations depending on calling stack, method arguments, etc., to future work.

Not all partitioning choices for $R(.)$ are legal (Section 3.1.1). To express these constraints in the optimization problem, we define an auxiliary decision variable $L(m)$ indicating the location of every method m, and three relations I, as well as DC and TC computed during static analysis. $I(i,m)$ is read as "i is an invocation of method m," and is trivially defined from the profile runs. Whereas DC and TC are computed once for each application, I is updated with new invocations only when the set S of profiling executions changes.

Using the decision variables $R(.)$, the auxiliary decision variables $L(.)$, the method sets V_M and V_{Nat_C} for all classes C defined during static analysis, and the relations I, DC and TC from above, we formulate the optimization constraints as follows:

$$L(m_1) \neq L(m_2), \quad \forall m_1, m_2 : DC(m_1, m_2) = 1 \wedge R(m_2) = 1 \quad (1)$$

$$L(m) = 0, \quad \forall m \in V_M \quad (2)$$

$$L(m_1) = L(m_2), \quad \forall m_1, m_2, C : m_1, m_2 \in V_{Nat_C} \quad (3)$$

$$R(m_2) = 0, \quad \forall m_1, m_2 : TC(m_1, m_2) = 1 \wedge R(m_1) = 1 \quad (4)$$

Constraint 1 is a soundness constraint, and requires that if a method causes migration to happen, it cannot be collocated with its callers. The remaining three correspond to the three properties defined in the static analysis. Constraint 2 requires that all methods pinned at the mobile device run on the mobile device (Property 1). Constraint 3 requires that methods dependent on the native state of the same class C are collocated, at either location (Property 2). And constraint 4 requires that all methods transitively called by a migrated method cannot be themselves migrated (Property 3).

The cost of a (legal) partition $R(.)$ of execution E is defined as follows, in terms of the auxiliary variables $L(.)$, the relation I and the cost variables C_c and C_s from the dynamic profiler:

$$C(E) = Comp(E) + Migr(E)$$

$$Comp(E) = \sum_{i \in E, m} [(1 - L(m))I(i,m)C_c(i,0) + L(m)I(i,m)C_c(i,1)]$$

$$Migr(E) = \sum_{i \in E, m} R(m)I(i,m)C_s(i)$$

$Comp(E)$ is the computation cost of the partitioned execution E and $Migr(E)$ is its migration cost. For every invocation $i \in E$, the computation cost takes its value from the mobile-device cost variables $C_c(i,0)$, if the method m being invoked is to run on the mobile device, or from the clone variables $C_c(i,1)$ otherwise. The migration cost sums the individual migration costs $C_s(i)$ of only those invocations i whose methods are migration points. Finally, the

optimization objective is to choose $R()$ so as to minimize $\sum_{E \in S} C(E)$. We use a standard integer linear programming (ILP) solver to solve this optimization problem with the above constraints. One can extend our optimization formulation to include a constraint limiting total energy consumption while optimizing total execution time or one limiting execution time while optimizing energy consumption.

4. Distributed Execution

The purpose of the distributed execution mechanism in CloneCloud is to implement a specific partition of an application process running inside an application-layer virtual machine, as determined during partitioning (Section 3).

The life-cycle of a partitioned application is as follows. When the user attempts to launch a partitioned application, current execution conditions (availability of cloud resources and network link characteristics between the mobile device and the cloud) are looked up in a database of pre-computed partitions. The lookup result is a partition configuration file. The application binary loads the partition and instruments the chosen methods with *migration* and *re-integration* points—special VM instructions in our prototype. When execution of the process on the mobile device reaches a migration point, the executing thread is suspended and its state (including virtual state, program counter, registers, and stack) is packaged and shipped to a synchronized clone. There, the thread state is instantiated into a new thread with the same stack and reachable heap objects, and then resumed. When the migrated thread reaches a re-integration point, it is similarly suspended and packaged as before, and then shipped back to the mobile device. Finally, the returned packaged thread is merged into the state of the original process.

CloneCloud migration operates at the granularity of a thread. This allows a multi-threaded process (e.g., a process with a UI thread and a worker thread) running on the phone to off-load functionality, one thread-at-a-time. For example, a process with a UI thread and a worker thread can migrate the functionality of the worker thread. The UI thread continues processing, unless it attempts to access migrated state, in which case it blocks until the offloaded thread comes back. Note that the current CloneCloud system does not support a distributed shared memory (DSM) model; when there are two worker threads that share the same state, they cannot be offloaded at the same time. CloneCloud enables threads, local and migrated, to use—but not migrate—native, non-virtualized features of the platform on which they operate: this includes the network and natively implemented API functionality (such as expensive-to-virtualize image processing routines), etc. Furthermore, when the underlying library and OS are designed to exploit unvirtualized hardware accelerators such as GPUs and cryptographic accelerators, the system seamlessly gain benefits from these special features. In contrast, most prior work providing application-layer virtual-machine migration keeps native features and functionality exclusively on the original platform, only permitting the off-loading of pure, virtualized computation.

These two unique features of CloneCloud, thread-granularity migration and native-everywhere operation, enable interesting execution models. For example, a mobile application can retain its user interface threads running and interacting with the user, while off-loading worker threads to the cloud if this is beneficial. This would have been impossible with monolithic process or VM suspend-resume migration, since the user would have to migrate to the cloud along with the code. Similarly, a mobile application can migrate a thread that performs heavy 3D rendering operations to a clone with GPUs, without having to modify the original application source; this would have been impossible to do seamlessly if only migration of virtualized computation were allowed.

CloneCloud migration is effected via three distinct components: (a) a per-process *migrator thread* that assists a process with suspending, packaging, resuming, and merging thread state, (b) a per-node *node manager* that handles node-to-node communication of packaged threads, clone image synchronization and provisioning; and (c) a simple partition database that determines what partition to use.

The migrator functionality manipulates internal state of the application-layer virtual machine; consequently we chose to place it within the same address space as the VM, simplifying the procedure significantly. A manager, in contrast, makes more sense as a per-node component shared by multiple applications, for several reasons. First, it enables application-unspecific node maintenance, including file-system synchronization between the device and the cloud. Second, it amortizes the cost of communicating with the cloud over a single, possibly authenticated and encrypted, transport channel. Finally, it paves the way for future optimizations such as chunk-based or similarity-enhanced data transfer [Muthitacharoen 2001, Tolia 2006]. Our current prototype has a simple configuration interface that allows the user to manually pick out a partition from the database, and to choose new configurations to partition for. We next delve more deeply into the design of the distributed execution facilities in CloneCloud.

4.1 Suspend and Capture

Upon reaching a migration point, the job of the thread migrator is to suspend a migrant thread, collect all of its state, and pass that state to the node manager for data transfer. The thread migrator is a native thread, operating within the same address space as the migrant thread, but outside the virtual machine. As such, the migrator has the ability to view and manipulate both native process state and virtualized state.

To capture thread state, the migrator must collect several distinct data sets: execution stack frames and relevant data objects in the process heap, and register contents at the migration point. Virtualized stack frames—each containing register contents and local object types and contents—are

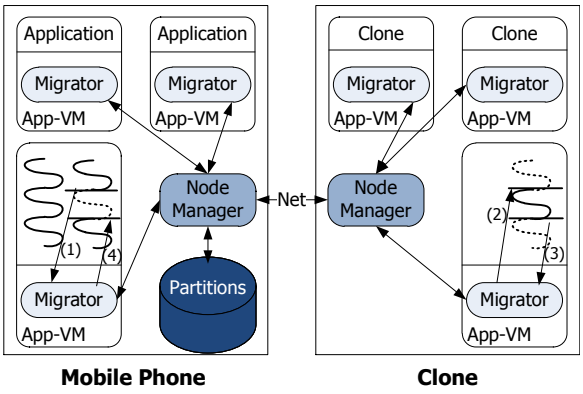

Figure 6. Migration overview.

readily accessible, since they are maintained by the VM management software. Starting with local data objects in the collected stack frames, the migrator recursively follows references to identify all relevant heap objects, in a manner similar to any mark-and-sweep garbage collector. For each relevant heap object, the migrator stores its field values, as well as relevant static class fields.

Captured state must be conditioned for transfer to be portable to adapt to different instruction set architectures (e.g., ARM and x86 architectures). First, object field values are stored in network byte order to allow for incompatibilities between different processor architectures. Second, whereas typically a stack frame contains a local native pointer to the particular class method it executes (which is not portable across address spaces or processor architectures), we store instead the class name and method name, which are portable.

In our prototype, state that is captured and migrated to the clone is marked in the phone VM. Remaining threads continue processing, unless they attempt to access such marked (migrated state), in which case they block until the off-loaded thread returns. This reduces the concurrency of some applications, making our current prototype better suited to applications with loosely-coupled threads. A distributed memory synchronization mechanism may alleviate this shortcoming, but we have yet to study the implications of the additional complexity involved.

4.2 Resume and Merge

As soon as the captured thread state is transferred to the target clone device, the node manager passes it to the migrator of a freshly allocated process. To resume that migrant thread, the migrator overlays the thread context over the fresh address space, essentially reversing the capture process described in Section 4.1. The executable text is loaded (it can be found under the same filename in the synchronized file system of the clone). Then all captured classes and object instances are allocated in the virtual machine's heap, updating static and instance field contents with those from the captured context. As soon as the address space contains all the data relevant to the migrant thread, the thread itself is created, given the stack frames from the capture, the register contents are filled to match the state of the original thread at the migration point in the mobile device, and the thread is marked as runnable to resume execution.

As described above, the cloned thread will eventually reach a reintegration point in its executable, signaling that it should migrate back to the mobile device. Reintegration is almost identical conceptually to the original migration: the clone's migrator captures and packages the thread state, the node manager transfers the capture back to the mobile device, and the migrator in the original process is given the capture for resumption. There is, however, a subtle difference in this reverse migration direction. Whereas in the forward direction—from mobile device to clone—a captured thread context is used to create a new thread from scratch, in the reverse direction—from clone to mobile device—the context must *update* the original thread state to match the changes effected at the clone. We call this a *state merge*.

A successful design for merging states in such a fashion depends on our ability to map objects at the original address space to the objects they "became" at the cloned address space; object references themselves are not sufficient in that respect, since in most application-layer VMs, references are implemented as native memory addresses, which look different in different processes, across different devices and possibly architectures, and tend to be reused over time for different objects.

Our solution is an *object mapping table*, which is only used during state capture and reinstantiation in either direction, and only stored while a thread is executing at a clone. We instrument the VM to assign a per-VM unique object ID to each data object created within the VM, using a local monotonically increasing counter. For clarity, we call the ID at the mobile device `MID` and at the clone `CID`. Once migration is initiated at the mobile device, a mapping table is first created for captured objects, filling for each the `MID` but leaving the `CID` null; this indicates that the object has no clone counterpart yet. After instantiation at the clone, the clone recreates all the objects with null `CID`s, assigning valid fresh `CID`s to them, and remembers the local object address corresponding to each mapping entry. At this point, all migrated objects have valid mappings.

During migration in the reverse direction, objects that came from the original thread are captured and keep their valid mapping. Newly created objects at the clone have the locally assigned ID placed in their `CID`, but get a null `MID`. Objects from the original thread that may have been deleted at the clone are ignored and no mapping is sent back for them. During the merge back at the mobile device, we know which objects should be freshly created (those with null `MID`s) and which objects should be overwritten with the contents fetched back from the clone (those with non-null `MID`s). "Orphaned" objects that were migrated out but died at the

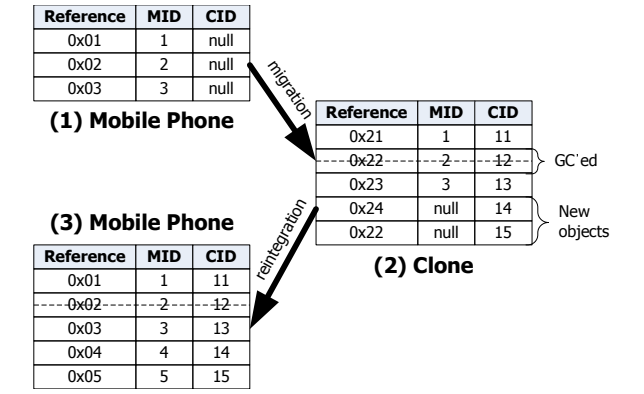

Figure 7. Object mapping example.

clone become disconnected from the thread object roots and are garbage-collected subsequently. Note that the mapping table is constructed and used only during capture and reintegration, not during normal memory operations either at the mobile device or at the clone.

Figure 7 shows an example scenario demonstrating the use of object mapping. During initial migration, objects at addresses 0x01, 0x02, and 0x03 are captured. The migrator creates the mapping table with three entries, one for each object, with the local ID of each object—1, 2, and 3, respectively—in `MID`, and null `CID`s. At the clone, the mapping table is stored, updating each entry with the local address of each object (0x21, 0x22, and 0x23, respectively). When the thread is about to return back to the mobile device, new entries are created in the table for captured objects whose IDs are not already in the `CID` column (objects with IDs 14 and 15). Entries in the table whose `CID` does not appear in captured objects are deleted (the second entry in the figure). Remaining entries belong to objects that came from the original thread and are also going back (those with `CID` 11 and 13). Note that memory address 0x22 was reused at the clone after the original object was destroyed, but the object has a different ID from the original object, allowing the migrator to differentiate between the two. Back at the mobile device, new objects are created for entries with null `MID`s (bottom two entries), objects with non-null `MID`s are updated with the returned state (first and third entries), and one object (with local address 0x02) is left to be garbage-collected.

We use object mappings for a subtly different purpose, as well. Because new processes are forked as copies of a "template" process —the *Zygote*, in the Android nomenclature— and because that template exists in all booted instances of the Android platform, we can avoid transmitting unchanged system heap objects. However, whereas application objects can be named at the time of creation, Zygote objects are created concurrently defying consistent naming. To address the challenge, we name each system object according to its class name and invocation sequence among all objects of that class—this assumes that objects from each class are constructed in the same order at Zygote processes on different platforms, an assumption that holds true in all Zygote instances we have seen so far.

5. Implementation

We implemented our prototype of CloneCloud partitioning and migration on the cupcake branch of the Android OS [AOS]. We tested our system on the Android Dev Phone 1 [ADP] (an unlocked HTC G1 device) equipped with both WiFi and 3G connections, and on clones running within the Android x86 virtual machine. Clones execute on a server with a 3.0GHz Xeon CPU, running VMware ESX 4.1, connected to the Internet through a layer-3 firewall. We modified the Dalvik VM (Android's application-level, register-based VM, principally targeted by a Java compiler front-end) [Dal] for dynamic profiling and migration. These modifications comprised approximately 8,000 lines of C code.

We implemented our static analysis on Java bytecode using JChord [JCh]. We modified JChord to support root methods of analysis that are different from `main`. We modified Dalvik VM tracing to efficiently trace execution and migration cost and to trace only application methods in which we are interested. To profile energy consumption of the phone, we connected the Monsoon power monitor [Mon], to the phone, measured the current drawn from the phone at a 5KHz frequency, and computed power consumption from the current and the voltage we set. We use lp_solve [lps] to solve the optimization problem for each execution environment.

The migrator uses Dalvik VM's thread suspension, a mechanism common in other application VMs. Threads are only suspended at bytecode instruction boundaries. We capture and represent execution state with a modified version of hprof [HPR]. We extend the format to also store thread stacks and class file paths, as well as store CIDs and MIDs to each object; we modified object creation and destruction in DalvikVM to assign those IDs. The object mapping table is a separate hash table inside the Dalvik VM, created only when migration begins, and destroyed after reintegration.

Migration is initiated and terminated in the modified application via two new system operations: `ccStart()` and `ccStop()`, respectively. The application thread calling these operations notifies the migrator thread inside Dalvik, and suspends itself. Once the migrator thread gains control, it checks with the loaded partition if it should migrate, and if so handles the rest.

6. Evaluation

To evaluate our prototype, we implemented three applications. We ran those applications either on a phone—a status quo, monolithic execution—or by optimally partitioning for two settings: one with WiFi connectivity and one with 3G.

We implemented a virus scanner, image search, and privacy-preserving targeted advertising; we briefly describe

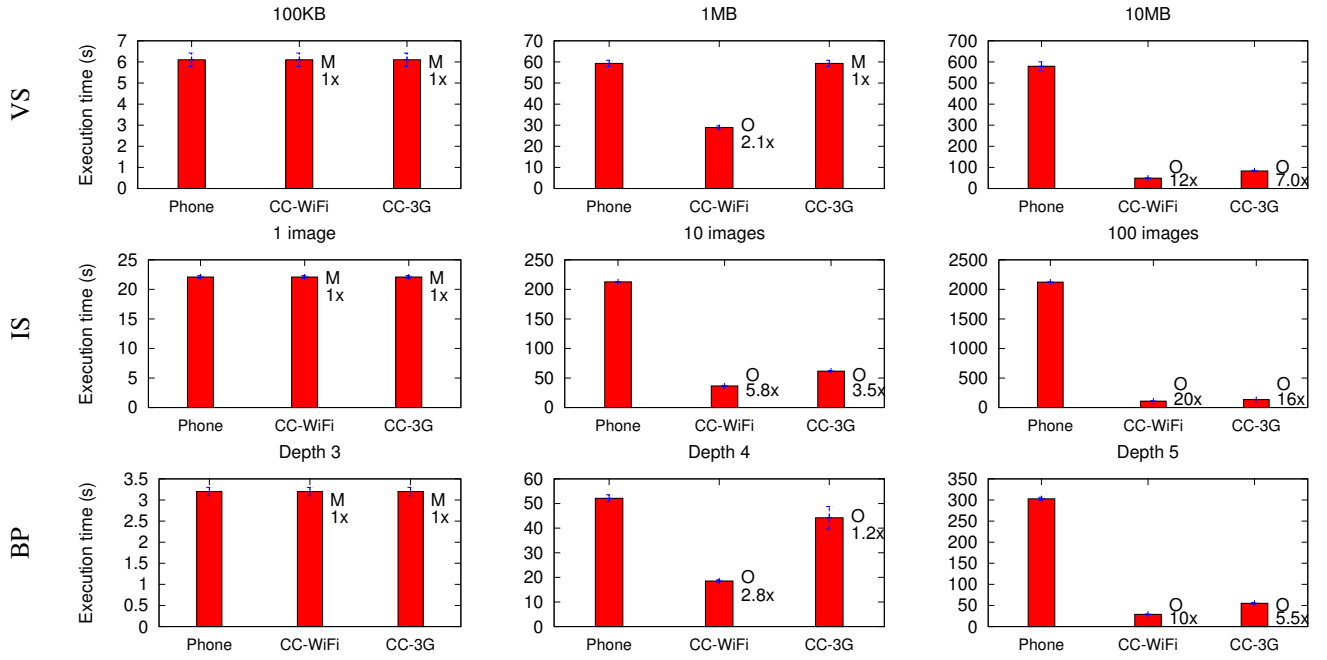

Figure 8. Mean execution times of virus scanning (VS), image search (IS), and behavior profiling (BP) applications with standard deviation error bars, three input sizes for each. For each application and input size, the data shown include execution time at the phone alone, that of CloneCloud with WiFi (CC-WiFi), and that of CloneCloud with 3G (CC-3G). The partition choice is annotated with M for "monolithic" and O for "off-loaded," also indicating the relative improvement from the phone-alone execution.

Application	Input Size	Phone Exec. (sec) Mean (std)	Clone Exec. (sec) Mean (std)
VS	100KB	6.1 (0.32)	0.2 (0.01)
	1MB	59.3 (1.49)	2.2 (0.01)
	10MB	579.5 (20.76)	22.5 (0.08)
IS	1 img	22.1 (0.26)	0.9 (0.07)
	10 img	212.8 (0.44)	8.0 (0.03)
	100 img	2122.1 (1.27)	79.2 (0.44)
BP	depth 3	3.3 (0.10)	0.2 (0.01)
	depth 4	52.1 (1.45)	1.8 (0.07)
	depth 5	302.7 (3.76)	10.9 (0.19)

Table 1. Execution times of virus scanning (VS), image search (IS), and behavior profiling (BP) applications, three input sizes for each. For each application and input size, the data shown include execution time at the phone alone and execution time at the clone alone.

each next. The virus scanner scans the contents of the phone file system against a library of 1000 virus signatures, one file at a time. We vary the size of the file system between 100KB and 10 MB. The image search application finds all faces in images stored on the phone, using a face-detection library that returns the mid-point between the eyes, the distance in between, and the pose of detected faces. We only use images smaller than 100KB, due to memory limitations of the Android face-detection library. We vary the number of images from 1 to 100. The privacy-preserving targeted-advertising application uses behavioral tracking across websites to infer the user's preferences, and selects ads according to a resulting model; by doing this tracking at the user's device, privacy can be protected (see Adnostic [Toubiana 2010]). We implement Adnostic's web page categorization, which maps a user's keywords to one of the hierarchical interest categories—down to nesting levels 3-5—from the DMOZ open directory [dmo]. The application computes the cosine similarity between user interest keywords and predefined category keywords.

For these applications, the static analysis with JChord took 23 seconds on average with sun jdk1.6.0_22 on a desktop machine; recall that this analysis need only be run once per application. Generating an optimizer (ILP) script from the profile trees, constraints and execution conditions, and solving the optimization problem, takes less than one second.

Table 1 shows the execution times of the applications at the phone alone or at the clone alone. The clone execution time is a lower-bound on execution time since, in practice, at least some part of the application must run on the phone. The difference between columns 3 and 4 captures the speedup opportunity due to the disparity between phone and cloud computation resources.

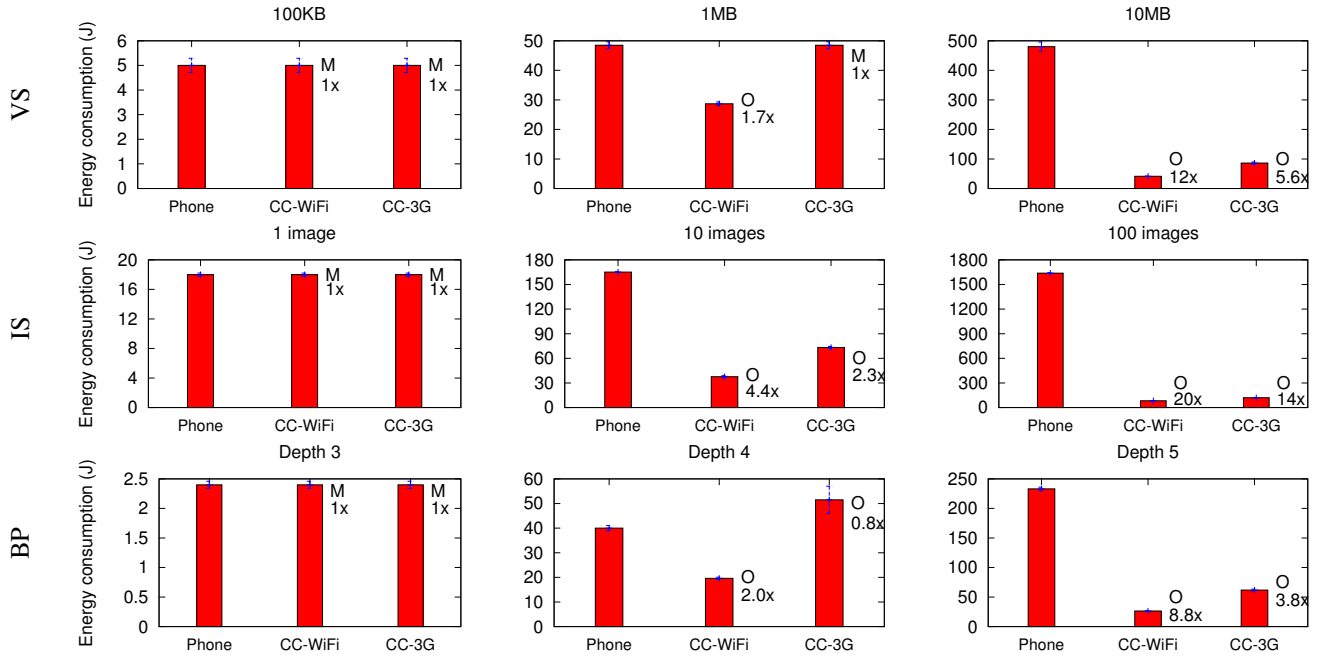

Figure 9. Mean phone energy consumption of virus scanning (VS), image search (IS), and behavior profiling (BP) applications with standard deviation error bars, three input sizes for each. For each application and input size, the data shown include execution time at the phone alone, that of CloneCloud with WiFi (CC-WiFi), and that of CloneCloud with 3G (CC-3G). The partition choice is annotated with M for "monolithic" and O for "off-loaded," also indicating relative improvement over phone-only execution.

Figures 8 and 9 shows execution times and phone energy consumption for the three applications, respectively. All measurements are the average of five runs. Each graph shows Phone, CloneCloud with WiFi (CC-WiFi), and CloneCloud with 3G (CC-3G). CC-Wifi and CC-3G results are annotated with the relative improvement and the partitioning choice, whether the optimal partition was to run monolithically on the phone (M) or to off-load to the cloud (O). In the experiments, WiFi had latency of 69ms and bandwidth of 6.6Mbps, and 3G had latency of 680ms, and bandwidth of 0.4Mbps.

CloneCloud chooses to keep local the smallest workloads from each application, deciding to off-load 6 out of 9 experiments with WiFi. With 3G, out of all 9 experiments, CloneCloud chose to off-load 5 experiments. For off-loaded cases, each application chooses to offload the function that performs core computation from its worker thread: scanning files for virus signature matching for VS, performing image processing for IS, and computing similarities for BP. CC-WiFi exhibits significant speed-ups and energy savings: 12x, 20x, and 10x speed-up, and 12x, 20x, and 9x less energy for the largest workload of each of the three applications, with a completely automatic modification of the application binary without programmer input. A clear trend is that larger workloads benefit from off-loading more: this is due to amortization of the migration cost over a larger computation at the clone that receives a significant speedup.

A secondary trend is that energy consumption mostly follows execution time: unless the phone switches to a deep sleep state while the application is off-loaded at the clone, its energy expenditure is proportional to how long it is waiting for a response. When the user runs a single application at a time, deeper sleep of the phone may further increase observed energy savings. We note that one exception is CC-3G, where although execution time decreases, energy consumption increases slightly for behavior profiling with depth 4. We believe this is due to our coarse energy cost model, and only occurs for close decisions. We hope to explore a finer-grained energy cost model that better captures energy consumption as future work.

CC-3G also exhibits 7x, 16x, and 5x speed-up, and 6x, 14x, and 4x less energy for the largest workload of each of the three applications. Lower gains can be explained given the overhead differences between WiFi and 3G networks. As a result, whereas migration costs about 15-25 seconds with WiFi, it shoots up to 40-50 seconds with 3G, due to the greater latency and lower bandwidth. In both cases, migration costs include a network-unspecific thread-merge cost—patching up references in the running address space from the migrated thread—and the network-specific transmission of the thread state. The former dominates the latter for WiFi, but is dominated by the latter for 3G. Our current implementation uses the DEFLATE compression algorithm to reduce the amount of data to send; we expect off-loading benefits

to improve with other optimizations targeting the network overheads (in particular, 3G network overheads) such as redundant transmission elimination.

7. Related Work

CloneCloud is built upon previous research done in automatic partitioning, migration, and remote execution. Most related is the recent work on MAUI [Cuervo 2010]. Similar to MAUI [Cuervo 2010], CloneCloud partitions applications using a framework that combines static program analysis with dynamic program profiling and optimizes execution time or energy consumption using an optimization solver. For offloaded execution, MAUI performs method shipping with relevant heap objects, but CloneCloud migrates specific threads with relevant execution state on demand and can merge migrated state back to the original process.

Although MAUI and CloneCloud have similar workflows, a number of differences distinguish them. First, supporting native method calls was an important design choice we made, which increases its applicability. MAUI does not support remotely executing virtualized methods calling native functions (e.g., two methods that share native state). Second, CloneCloud requires little programmer help, whereas MAUI requires programmers to annotate methods as REMOTABLE. Third, MAUI does not focus on the details of method shipping, whereas we present a detailed design for state migration and merging, which is a major source of design challenges. Lastly, CloneCloud solves the optimization problem asynchronously, whereas MAUI requires a solver to be running at the server at runtime.

Further out, there is a much prior work on partitioning, migration, and remote execution, which we summarize.

Partitioning We first summarize work on partitioning of distributed systems. Coign [Hunt 1999] automatically partitions a distributed application composed of Microsoft COM components to reduce communication cost of partitioned components. The application must be structured to use COM components and partitioning points are coarse-grained COM boundaries. Coign focuses on static partitioning, which works better in a stable environment. Giurgiu [2009] takes a similar approach for applications designed on top of distribution middleware such as OSGi. Wishbone [Newton 2009] and Pleiades [Kothari 2007] compile a central program into multiple code pieces with stubs for communication. They are primarily intended for sensor networks, and they require programs to be written in special languages (a stream-processing language, and an extended version of C, respectively). J-Orchestra [Tilevich 2002] creates partitioned applications automatically by a compiler that classifies anchored unmodifiable, anchored modifiable, or mobile classes. After the analysis, it rewrites all references into indirect references (i.e., references to proxy objects) for a cluster of machines, and places classes with location constraints (e.g., ones with native state constraints) to proper locations. Finally, for distributed execution of partitioned applications, it relies on the RMI middleware. Chroma [Balan 2003] uses *tactics*, application-specific knowledge for remote execution in a high-level declarative form, for run-time partitioning.

There are also Java program partitioning systems for mobile devices, whose limitation is that only Java classes without native state can be placed remotely [Gu 2003, Messer 2002, Ou 2007]. The general approach is to partition Java *classes* into groups using adapted MINCUT heuristic algorithms to minimize the component interactions between partitions. Also, different proposals consider different additional objectives such as memory, CPU, or bandwidth. Besides disallowing native execution offloading, this previous work does not consider partitioning constraints like our work does, the granularity of partitioning is coarse since it is at class level, and it focuses on static partitioning.

On a related front, Links [Cooper 2006], Hops [Serrano 2006], and UML-based Hilda [Yang 2006] aim to statically partition a client-server program written in a high-level functional language or a high-level declarative language into two or three tiers. Yang [2007] examines partitioning of programs written in Hilda based on cost functions for optimizing user response time. Swift [Chong 2007] statically partitions a program written in the Jif programming language into client-side and server-side computation. Its focus is to achieve confidentiality and integrity of the partitioned program with the help of security labels in the program annotated by programmers.

Migration There has been previous work on supporting migration in Java. MERPATI [Suezawa 2000] provides JVM migration checkpointing the entire heap and all the thread execution environments (call stack, local variables, operand stacks) and resuming from a checkpoint. In addition, there have been different approaches on distributed Java virtual machines (DJVMs). They assume a cluster environment where homogeneous machines are connected via fast interconnect, and try to provide a single system image to users. One approach is to build a DJVM upon a cluster below the JVM. Jessica [Ma 1999] and Java/DSM [Yu 1997] rely on page-based distributed shared memory (DSM) systems to solve distributed memory consistency problems. To address the overhead of false sharing in page-based DSM, Jessica2 [Zhu 2002] is an object-based solution. cJVM [Aridor 1999] modifies the JVM to support method shipping to remote objects with proxy objects, creating threads remotely, and supporting distributed stacks. Object migration systems such as Emerald [Jul 1988] move objects to the sites running threads requesting to access the objects. In contrast, CloneCloud migration chooses partial threads to offload, moves only their relevant execution state (thread stack and reachable heap objects), and supports merging between existing state and migrated execution state.

Remote execution Remote execution of resource-intensive applications for resource-poor hardware is a well-known ap-

proach in mobile/pervasive computing. Most remote execution work carefully designs and pre-partitions applications between local and remote execution. Typical remote execution systems run a simple visual, audio output routine at the mobile device and computation-intensive jobs at a remote server [Balan 2002, Flinn 2001; 1999, Fox 1996, Rudenko 1998, Young 2001]. Rudenko [1998] and Flinn [1999] explore saving power via remote execution. Cyber foraging [Balan 2002; 2007] uses surrogates (untrusted and unmanaged public machines) opportunistically to improve the performance of mobile devices, similarly to data staging [Flinn 2003] and Slingshot [Su 2005]. In particular, Slingshot creates a secondary replica of a home server at nearby surrogates. ISR [Satyanarayanan 2005] provides the ability to suspend on one machine and resume on another machine by storing virtual machine (e.g., Xen) images in a distributed storage system.

8. Discussion and Future Work

CloneCloud is limited in some respects by its inability to migrate native state and to export unique native resources remotely. Conceptually, if one were to migrate at a point in the execution in which a thread is executing native code, or has native heap state, the migrator would have to collect such native context for transfer as well. However, the complexity of capturing such information in a portable fashion (and the complexity of integrating such captures after migration) is significantly higher, given processor architecture differences, differences in file descriptors, etc. As a result, CloneCloud focuses on migrating at execution points where no native state (in the stack or the heap) need be collected and migrated.

A related limitation is that CloneCloud does not virtualize access to native resources that are not virtualized already and are not available on the clone. For example, if a method accesses a camera/GPS on the mobile device, CloneCloud requires that method to remain pinned on the mobile device. In contrast, networking hardware or an unvirtualized OS facility (e.g., Android's image processing API) are available on both the mobile device and the clone, so a method that needs to access them need not be pinned. An alternative design would have been to permit migration of such methods, but enable access to the unique native resource via some RPC-like mechanism. We consider this alternative a complementary point in the design space, and plan to pursue it in conjunction with thread-granularity migration in the future.

The system presented in this paper allows only perfunctory concurrency between the unmigrated threads and the migrated thread; pre-existing state on the mobile device remains unmodifiable until the migrant thread returns. As long as local threads only read existing objects and modify only newly created objects, they can operate in tandem with the clone. Otherwise, they have to block. A promising direction, whose benefits may or may not be borne out by the associated complexity, lies in extending this architecture to support full concurrency between the mobile device and clones. To achieve this, we need to add thread synchronization, heap object synchronization, on-demand object paging to access remote objects, etc.

While in this paper we assume that the environment in which we run clone VMs is trusted, the future of roaming devices that use clouds where they find them demands a more careful approach. For instance, many have envisioned a future in which public infrastructure machines such as public kiosks [Garriss 2008] and digital signs are widely available for running opportunistically off-loaded computations. We plan to extend our basic system to check that the execution done in the remote machine is trusted. Automatically refactoring computation around trusted features on the clone is an interesting research question.

9. Conclusion

This paper takes a step towards seamlessly interfacing between the mobile and the cloud. Our system overcomes design and implementation challenges to achieve basic augmented execution of mobile applications on the cloud, representing the whole-sale transfer of control from the device to the clone and back. We combine partitioning, migration with merging, and on-demand instantiation of partitioning to address these challenges. Our prototype delivers up to 20x speedup and 20x energy reduction for the simple applications we tested, without programmer involvement, demonstrating feasibility for the approach, and opening up a path for a rich research agenda in hybrid mobile-cloud systems.

Acknowledgments

We would like to thank our shepherd Pradeep Padala and the anonymous reviewers for their insightful feedback.

References

[ADP] Android dev phone 1. `developer.android.com/guide/developing/device.html`.

[AOS] Android open source project. `source.android.com/`.

[Dal] Dalvik VM. `developer.android.com/guide/basics/what-is-android.html`.

[HPR] Hprof: A heap/cpu profiling tool in J2SE 5.0. `java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

[JCh] JChord. `code.google.com/p/jchord`.

[lps] lp_solve. `lpsolve.sourceforge.net`.

[Mon] Monsoon power monitor. `www.msoon.com`.

[dmo] Open directory project. `www.dmoz.org`.

[Aridor 1999] Y. Aridor, M. Factor, and A. Teperman. cJVM: a single system image of a JVM on a cluster. In *ICPP*, 1999.

[Balan 2002] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *ACM SIGOPS European Workshop*, 2002.

[Balan 2007] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *MobiSys*, 2007.

[Balan 2003] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *MobiSys*, 2003.

[Beccue 2009] Mark Beccue and Dan Shey. Mobile Cloud Computing. Research report, ABI Research, 2009.

[Cadar 2008] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[Chong 2007] S. Chong, J. Liu, A. C. Myers, and X. Qi. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[Chun 2009] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, 2009.

[Cooper 2006] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, 2006.

[Cuervo 2010] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *MobiSys*, 2010.

[Flinn 2001] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *HotOS*, 2001.

[Flinn 1999] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP*, 1999.

[Flinn 2003] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data staging for untrusted surrogates. In *USENIX FAST*, 2003.

[Fox 1996] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *ASPLOS*, 1996.

[Garriss 2008] S. Garriss, R. Càceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *MobiSys*, 2008.

[Giurgiu 2009] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Middleware*, 2009.

[Gu 2003] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *PerCom*, 2003.

[Hunt 1999] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *OSDI*, 1999.

[Jul 1988] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. on Computer Systems*, 1988.

[Kothari 2007] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI*, 2007.

[Ma 1999] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. JESSICA: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 1999.

[Messer 2002] 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 *ICDCS*, 2002.

[Muthitacharoen 2001] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, 2001.

[Newton 2009] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI*, 2009.

[Ou 2007] S. Ou, K. Yang, and J. Zhang. An effective offloading middleware for pervasive services on mobile devices. *Journal of Pervasive and Mobile Computing*, 2007.

[Rudenko 1998] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *MCCR*, 1998.

[Satyanarayanan 2005] M. Satyanarayanan, Michael A. Kozuch, Casey J. Helfrich, and David R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Journal of Pervasive and Mobile Computing*, 2005.

[Satyanarayanan 2009] M. Satyanarayanan, Bahl P., Caceres R., and Davies N. The case for vm-based cloudlets in mobile computing. *Pervasive Computing*, 8(4), 2009.

[Serrano 2006] M. Serrano, E. Gallesio, and F. Loitsch. HOP: a language for programming the web 2.0. In *Proc. 1st Dynamic Languages Symposium*, 2006.

[Su 2005] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *MobiSys*, 2005.

[Suezawa 2000] Takashi Suezawa. Persistent execution state of a java virtual machine. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[Tilevich 2002] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP*, 2002.

[Tolia 2006] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *NSDI*, 2006.

[Toubiana 2010] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[Yang 2007] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, 2007.

[Yang 2006] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web application. In *ICDE*, 2006.

[Young 2001] C. Young, Y. N. Lakshman, T. Szymanski, J. Reppy, and D. Presotto. Protium, an infrastructure for partitioned applications. In *HotOS*, 2001.

[Yu 1997] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 1997.

[Zhu 2002] W. Zhu, C.-L. Wang, and F.C.M.Lau. JESSICA2: A distributed java virtual machine with transparent thread migration support. In *CLUSTER*, 2002.