Augmented Smartphone Applications Through Clone Cloud Execution

Byung-Gon Chun, Petros Maniatis
Intel Research Berkeley

Abstract

Smartphones enable a new, rich user experience in pervasive computing, but their hardware is still very limited in terms of computation, memory, and energy reserves, thus limiting potential applications. In this paper, we propose a novel architecture that addresses these challenges via seamlessly—but partially—off-loading execution from the smartphone to a computational infrastructure hosting a cloud of smartphone *clones*. We outline new *augmented* execution opportunities for smartphones enabled by our *CloneCloud* architecture.

1 Introduction

Smartphones with Internet access, GPS, sensors, and various applications are recently seeing explosive adoption. The Apple iPhone [2], Blackberry smartphones [3], and the Google Android phone [1] are a few prominent examples. In a slightly more advanced capability bracket also lie mobile Internet devices (MIDs) such as the Nokia N810 [7] and Moblin-based devices [6] that provide a richer untethered Internet experience.

With popularity, such devices also see new applications by a broader set of developers, beyond the mobile staples of personal information management and music playback. Now mobile users play games; capture, edit, annotate and upload video; handle their finances; manage their personal health and "wellness" (e.g., iPhone Heart Monitor [16] and Diamedic [15]). However, with greater application power comes greater responsibility for the mobile execution platform: it is now important to track memory leaks and runaway processes sucking up power, to avoid or detect malicious intrusions and private data disclosure, and to manage applications with expensive tastes for high-volume data or advanced computational capabilities such as floating-point or vector operations.

Solutions for all these advanced capabilities have been known and are in (fairly) common practice in traditional desktop and server platforms; this is, after all, why smartphone users expect to apply those solutions to their mobile devices. Alas, such solutions tend to be expensive when cast to mobile architectures. The hardware capabilities of those devices are similar to those of the desktop PCs of the mid-1990's, many generations of hardware and software behind (see Table 1 and contrast to Table 2).

For example, anti-virus software operates by performing frequent complete scans of all files in a file system, and by imposing on-access scans on the virtual memory contents of a process, including memory-mapped files. On a smartphone, even if the user were patient enough to wait until such a CPU- and I/O-intensive scan were over, she might still hit memory limits or run out of battery power. It only gets worse if one considers tools like taint-checking [23] for data leak prevention, floating-point and vector operations for mathematical or signal-processing applications such as face detection in media, etc.

Phone	CPU (MHz)	RAM (MB)	Battery (talk time in hrs)
IPhone 3G	412	512	5
Android HTC G1	528	192	6
Blackberry Bold	624	128	4.5

Table 1: Specifications of a few high-end smartphones. Their network connectivities include Wi-Fi, UMTS, WCDMA, HSDPA, GSM/GPRS/EDGE, and Bluetooth 2.0.

Computer	CPU	RAM
MacBook Pro Laptop	2.5GHz 2-core	4GB
Dell Precision T7400	3.3GHz 4-core	8GB

Table 2: Specification of a commodity laptop and a desktop. Their connectivities include 1Gbps Ethernet and Wi-Fi, and they are frequently powered from the electric grid.

In this paper we (re)discover an opportunity that might overcome these concerns. On one hand, laptop, desktop and server resources are abundant, ubiquitous, and continuously reachable, as ensured by cloud computing, multi-core desktop processors and plentiful wireless connectivity such as 3G, UltraWideBand, Wi-Fi, and WiMax technologies. The disparity in capability between such computers and the untethered smartphone is high and persistent. On the other hand, technologies for replicating/migrating execution among connected computing substrates, including live virtual machine migration and incremental checkpointing, have matured and are used in production systems [9, 10].

We capitalize on this opportunity here by proposing a simple idea: *let the smartphone host its expensive, exotic applications*. However, do so on an execution engine that *augments* the smartphone's capabilities by seamlessly off-loading some tasks to a nearby computer, where they are executed in a cloned whole-system image of the device, reintegrating the results in the smartphone's execution upon completion. This augmented execution overcomes smartphone hardware limitations and it is provided (semi)-automatically to applications whose developers need few or no modifications to their applications.

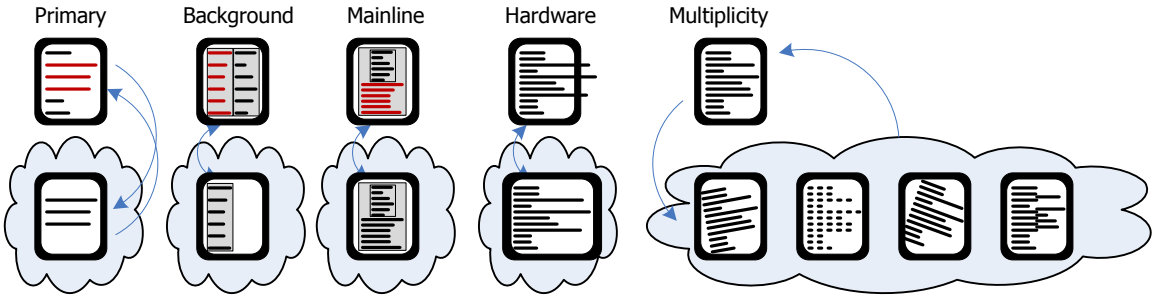

Figure 1: The five categories of augmented execution.

Some augmentation can operate in the background, for asynchronous operations such as periodic file scans. For synchronous operations intrinsic to the application (e.g., a train of floating-point instructions in the application code), augmentation can be performed by blocking progress on the smartphone until the result arrives from the clone in the cloud[1]. For concurrent operations to the application that operate "around" it (e.g., taint-checking), augmentation can also be concurrent in the clone cloud or even speculative with the ability to undo operations on the smartphone according to the result from the clone.

While the ability to off-load expensive computations from weak, mobile devices to powered, powerful devices has been recognized before, the novelty of our approach lies in using loosely synchronized virtualized or emulated replicas of the mobile device on the infrastructure to maintain two illusions: first, that the mobile user has a much more powerful, feature-rich device than she does in reality, and second that the programmer is programming such a powerful, feature-rich device, without having to manually partition his application [28, 29], explicitly provision proxies [20], or just dumb down the application.

In what follows, we outline the categories of augmentation we consider, derive from them a straw-man architecture for our envisioned system, and outline the research challenges ahead.

2 Augmented Execution

The scope of augmented execution from the infrastructure is fairly broad. In this section, we attempt to categorize the types of augmentation we envision (Figure 1). We discuss how to achieve such augmentation in the next sections.

Primary functionality outsourcing: Computation-hungry applications such as speech processing, video indexing, and super-resolution are automatically split, so that the user-interface and other low-octane processing is retained at the smartphone, while the high-power, expensive computation is off-loaded to the infrastructure, synchronously. This is similar to designing the application as a client-server service, where the infrastructure provides the service (e.g., the translation of speech to text), or as a thin-client environment.

Background augmentation: Unlike primary functionality outsourcing, this category deals with functionality that does not need to interact with users in a short time scale. Such is functionality that typically happens in the background, such as scanning the file system for viruses [5], indexing files for faster search [4], analyzing photos for common faces [8], crawling news web pages, etc. In this case, entire processes can be marked (by the user or by the programmer) or automatically inferred as "background" processes, and migrated to the infrastructure wholesale. Furthermore, off-loaded functionality can take on the role of a "virtual client." Even when the smartphone is turned off, the virtual client can continue to run background tasks. Later when the smartphone returns online, it can synchronize its state with the infrastructure.

Mainline augmentation: This category sits between primary functionality outsourcing and background augmentation. Here the user may opt to run a particular application in a *wrapped* fashion, altering the method of its execution but not its semantics. Examples are private-data leak detection (e.g., to taint-check an application or application set), fault-tolerance (e.g., to employ multi-variant execution analysis to protect the application from transparent bugs), or debugging (e.g., keep track dynamically of allocated memory in the heap to catch memory leaks). Unlike background augmentation, mainline augmentation is interspersed in the execution of the application. Many possibilities exist: for example, when a decision point is reached in the taint-check example, the application on the smartphone may block, perhaps causing the clone to rewind back to a known checkpoint, and to re-execute forward with taint-tracking, before deciding.

Hardware augmentation: This category is interesting because it compensates for fundamental weaknesses of the smartphone platform, such as memory caps or other constraints, and hardware peculiarities.

For demonstration, we wrote a file system scanning application in the DalvikVM, the execution environment of the original Google Android phone (HTC G1). We ran it to scan 100,000 directories and files. On the HTC G1 the process took 3953 seconds. This was much higher than

[1]We use the term "cloud" in a broader sense to include personal laptops sitting on a nearby lap, desktops at work or at home, and servers located in accessible data centers. Smartphones may have very different network latencies and bandwidth to each type of computer.

we expected. Through a debugger, we discovered that the program invokes garbage collection very frequently due to memory pressure. Just using faster hardware—we ran on a QEMU-emulated single-core virtual machine on a Dell Desktop with a 2.83GHz CPU and 4GB RAM—significant savings can be observed even while thrashing: our scenario only took 336 seconds (11.8x). If we were to modify the heap and stack allocation of the virtual machine to remove most garbage collection activity, it could improve that significantly. A similarly powerful augmentation might execute a clone on an x86 port of the Android platform, removing the costs of emulating the ARM processor in the G1 Android smartphone.

Augmentation through multiplicity: The last category we consider is unique in that it uses multiple copies of the system image executed in different ways. This can help running data parallel applications (e.g., doing indexing for disjoint sets of images). This can also help the application to "see the future," by exhaustively exploring all possible next steps within some small horizon—as would be done for model checking—or to evaluate in maximum detail all possible choices for a decision before making that decision. Consider, for example, an energy-conserving process scheduler that, in the absence of future knowledge, can only guarantee decisions close but not at the optimum. Instead, the whole system image could be replicated multiple times in the infrastructure, choosing all possible interleavings of processes during execution, and evaluating energy expenditure via some consumption model for the device, ultimately making the scheduling decision that results in the minimum expenditure. In this category of augmentation, infrastructure cycles are lavished on essentially a Monte-Carlo simulation of all possible outcomes of the scheduler's choices to make the optimal decision. We end up wasting much energy (at the infrastructure) to save a little bit of energy on the mobile device. However, given the opportunity cost of being left with a dead battery during a critical time, this rather extravagant use of the infrastructure may have significant benefits.

3 Architecture

Conceptually, our system provides a way to boost a smartphone application by utilizing heterogeneous computing platforms through cloning and computation transformation. For doing so, our system (semi)-automatically transforms a single-machine execution (e.g., smartphone computation) into a distributed execution (e.g., smartphone plus cloud computation) in which the resource-intensive part of the execution is run in powerful clones. An additional benefit of cloning is that if the smartphone is lost or destroyed, the clone can be used as a backup. Figure 2 illustrates the high-level system model of our approach.

Augmented execution is performed in four steps: 1) Initially, a clone of the smartphone is created within the cloud (laptop, desktop, or server nodes); 2) The state of the primary (phone) and the clone is periodically or on-demand synchronized; 3) Application augmentations (whole applications or augmented pieces of applications) are executed in the clone, automatically or upon request; and 4) Results from clone execution are re-integrated back into the smartphone state.

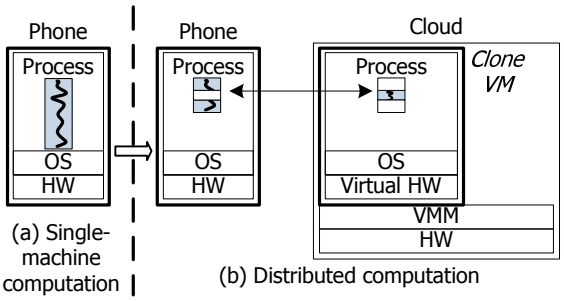

Figure 2: Our system model. Our system transforms a single-machine execution (smartphone computation) into a distributed execution (smartphone and cloud computation) (semi)-automatically.

Figure 3 shows a high-level view of our system architecture. This is one possible design, and we are exploring the design space of different system architectures (e.g., doing this task mostly in DalvikVMs in the case of the Android platform). We achieve this by combining whole-system replication through incremental checkpointing, (semi)-automatic partitioning and invocation of augmented execution, and coordination of computation between the primary (phone) and the clone. The system components are running inside the operating system (OS). The *Replicator* is in charge of synchronizing the changes in phone software and state to the clone. The *Controller* running in the smartphone invokes an augmented execution and merges its results back to the smartphone. It interacts with the Replicator to synchronize states while coordinating the augmentation. The *Augmenter* running in the clone manages the local execution, and returns a result to the primary.

Once a computation block for remote execution is specified, the following steps are performed for the primary functionality outsourcing augmentation category. We omit the steps for other augmentations due to space constraints. First, the smartphone application process enters a sleep state. The process transfers its state to the clone VM. The VM allocates a new process state and overlays what it received from the phone with hardware description translation. The clone executes from the beginning of the computation block until it reaches the end of the computation block. The clone transfers its process state back to the phone. The phone receives the process state and reintegrates it, and wakes up the sleeping process to continue its execution. This description omits much detail, and other augmentation categories can be even less straightforward. We outline the open research challenges involved in this

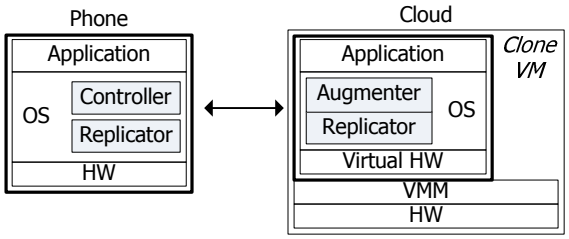

Figure 3: Clone execution architecture for smartphones.

architecture next.

4 Research Agenda

In this section, we discuss major research questions we need to address, and the directions we are currently taking to build our system.

How is computation transformation done for augmented execution? We expect that for some applications augmentation is automated, while for others it involves a simple manual process, e.g., annotating a self-contained resource-intensive block of execution such as complicated image processing code, or profiling and run-time partitioning of applications to use augmented execution. For example, we can run static or dynamic analysis in the clone VM(s) to extract computationally expensive blocks of computation and annotate the blocks for off-loading. We plan to explore different *policies* that decide when to perform this computation transformation considering the computation and network latency and resource usage such as power. Note that augmentation here is *cascaded*: we augment the application with a profiler and partitioner, so as to better augment it in subsequent executions. Automated partitioning is an important research question.

For background or mainline augmentation, our system can do simple automatic partitioning. For background augmentation, an application is initially configured to use the augmentation. When an operation (e.g., virus scanning) is performed in the clone, the application simply conveys the results to the phone user. For mainline augmentation, the application developers can specify where augmentation may be applicable if available. For example, for taint checking, when an input is received from the network or an output is sent to the network, the Controller can invoke mainline augmentation to verify that the operation does not violate the application's security policy.

For applications that require the primary functionality outsourcing or mainline augmentation, we can profile runtime performance and feasibility of operations, and offload computation based on the profiling information. If some operations take too long or are not possible to run because available memory is not large enough, the operations are tagged to invoke augmentation.

How do we do synchronization of states? The Replicator faces the challenges of optimizing wireless bandwidth and battery power while replicating fresh images to the clone. To save bandwidth, it performs *incremental checkpointing*, i.e., sends deltas of two checkpoints, and *two-level synchronization*. By default, it periodically performs synchronization in a coarse time scale (e.g., once every few hours). For asynchronous operations like background augmentation, the basic synchronization may be sufficient. For mainline augmentation and primary functionality outsourcing, we perform more fine-grained synchronization of in-memory and persistent states. The Replicator achieves this goal in coordination with the Controller.

A main research question is to decide when and how a mobile device performs synchronization (*policies* of synchronization) considering the trade-offs between latency/accuracy and resource usage. In addition to periodic synchronization, the Replicator may exploit opportunities for optimization by performing opportunistic synchronization. For example, if the smartphone discovers a high-speed Wi-Fi connection, it can do more aggressive synchronization to avoid using 3G cellular connections. Also, if it is charging at night, it can synchronize without draining the battery.

How do we coordinate execution between the primary and the clone? Depending on operation types, we use different coordination strategies. For background augmentation, the execution is off-loaded to the clone without tight time constraints. The clone runs the computation with some snapshot. When the clone finishes its computation, the Augmenter sends the result back to the Controller. Synchronous operations are more difficult to support. When an operation is in the critical path of execution, the Controller invokes the operation at the clone, and pauses the primary's execution until it gets the result back from the Augmenter. Once the Controller receives the result, it restarts the primary's execution.

For mainline augmentation, we use more complicated coordination to hide the latency. The primary performs *speculative execution*, which has been used in local and distributed file systems and virtual machine replication for high availability [14,24,25] while invoking augmented execution. The primary buffers any externally visible output while the augmented execution is running. Once it receives a *commit notification* from the clone, it releases the buffered outputs.

How does hardware augmentation work? We provide two kinds of hardware augmentation. First, we modify the virtual hardware for *capability inflation*. We increase the CPU clock rate of the virtual hardware, the number of virtual CPU cores (if there are multiple cores available), and the memory size of VMs. This requires a mechanism that reconciles the difference between the smartphone hardware and the virtual hardware. Second, we expose any special capabilities of the hardware platform (e.g., a cryptographic accelerator) to VMs through virtual hardware.

What if we cannot trust clone VM environments? In

the basic setup, we assume that the environment in which we run clone VMs is trusted. In the future, one can imagine that public infrastructure machines such as public kiosks [21] and digital signs are widely available. We can off-load computation of smartphone applications to such public infrastructure machines, but they cannot be trusted. Our basic system needs to be extended to check that the execution done in the remote machine is trusted. One direction is to utilize trusted hardware that certifies that the computation done in the infrastructure is correct. At a high level, the trusted hardware receives inputs and simple programs written in a little, domain-specific language and sends out outputs and attestation, which is a generalized form of trusted primitives studied in [12, 13]. The smartphone can do a simple verification of the proof to accept the result from the clone. Refactoring computation around this trusted hardware is an interesting research question.

Are smartphones all there is? Although the disparity between the capabilities of smartphones and computers at home or in an infrastructure particularly favors the kind of augmented execution we envision, one can see several paths to applying this architecture more broadly. For instance, one could imagine using this approach in the context of data center architectures, in which some processors are low-power Intel Atom, while others are high-performance Intel Nehalem, or in the context of heterogeneous multi-core architectures, in which some cores have floating-point (FP) instructions, for instance, while others do not. In the latter scenario, a clone executing only the FP code may be a good way to avoid more complex application partitionings, and the fast bus speeds as well as copy-on-write might make our approach particularly desirable. A similarly fortuitous application would be the outsourcing of sensitive tasks to a nearby core with trusted execution features on-package, keeping all other computation on other simpler, perhaps less contested cores.

5 Related Work and Conclusion

Remote execution of resource-intensive applications for resource-poor hardware is a well-known approach in mobile/pervasive computing. All remote execution work carefully designs and partitions applications between local and remote execution, and runs a simple visual, audio output routine at the mobile device and computation-intensive jobs at a remote server [11, 17, 18, 20, 26, 29]. Rudenko et al. [26] and Flinn and Satyanarayanan [18] explored saving power via remote execution. Cyber foraging [11] uses surrogates (untrusted and unmanaged public machines) opportunistically to improve the performance of mobile devices. For example, both data staging [19] and Slingshot [28] use surrogates. In particular, Slingshot creates a secondary replica of a home server at nearby surrogates. ISR [27] provides an ability to suspend on one machine and resume on another machine by storing virtual machine images in a distributed storage system. Coign [22] automatically partitions a distributed application composed of Microsoft COM components.

To our knowledge, our approach is the first to replicate the whole smartphone image and to run the application code with few or no modifications in powerful VM replicas to transform a single-machine computation to a distributed computation (semi)-automatically.

We believe that the CloneCloud architecture enables new, exciting modes of augmented execution for applications in diverse environments, and offers intriguing opportunities for research and for practical deployments that marry the convenience of hand-held devices with the power of cloud computing.

Acknowledgments: We are indebted to Anthony Joseph, Gianluca Iannaccone, Sylvia Ratnasamy, and the workshop reviewers for their comments on our work.

References

[1] Android dev phone 1. `code.google.com/android/dev-devices.html`.
[2] Apple iPhone. `www.apple.com/iphone`.
[3] Blackberry smart phones. `na.blackberry.com/eng/`.
[4] Google desktop. `desktop.google.com/`.
[5] Mcafee. `www.mcafee.com`.
[6] Moblin. `http://moblin.org`.
[7] Nokia N810 Internet tablet. `www.nseries.com/index.html#1=products,n810`.
[8] Picasa. `picasa.google.com/`.
[9] VMware ESX. `www.vmware.com/products/vi/esx/`.
[10] Xen. `www.xen.org`.
[11] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *ACM SIGOPS European Workshop*, 2002.
[12] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *SOSP*, 2007.
[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered Fault Tolerance for Long-Term Integrity. In *FAST*, 2009.
[14] B. Cully et al. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.
[15] Diamedic. Diabetes Glucose Monitoring Logbook. `www.martoon.com/diamedic/`.
[16] J. Diaz. iPhone Heart Monitor Tracks Your Heartbeat Unless You Are Dead. `gizmodo.com/5056167/iphone-heart-monitor-tracks-your-heart-beat-unless-you-are-dead`, 2008.
[17] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *HotOS*, 2001.
[18] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP*, 1999.
[19] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data staging for untrusted surrogates. In *USENIX FAST*, 2003.
[20] 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.
[21] S. Garriss et al. Trustworthy and personalized computing on public kiosks. In *MobiSys*, 2008.
[22] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI*, 1999.
[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.
[24] E. B. Nightingale, P. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.
[25] E. B. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *OSDI*, 2006.
[26] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *MCCR*, 1998.
[27] M. Satyanarayanan et al. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 2007.
[28] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *MobiSys*, 2005.
[29] C. Young et al. Protium, an infrastructure for partitioned applications. In *HotOS*, 2001.