Just-in-Time Provisioning for Cyber Foraging

Kiryong Ha, Padmanabhan Pillai[†], Wolfgang Richter,
Yoshihisa Abe, Mahadev Satyanarayanan

December 2012
CMU-CS-12-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Intel Labs

Abstract

VM-based cloudlets have been proposed as offload sites for the resource-intensive and latency-sensitive computations typically associated with mobile multimedia applications. Although many offload strategies have been explored, they all rely on precisely-configured back-end software. Since precise pre-provisioning of cloudlets across multiple domains is difficult at global scale, we describe just-in-time (JIT) provisioning under the control of an associated mobile device. We demonstrate a prototype system that is capable of provisioning a cloudlet with a non-trivial VM image in just 10 seconds. This speed is achieved through a series of optimizations detailed in the paper, using a suite of five representative mobile applications to guide the process.

This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0833882 and IIS-1065336, by an Intel Science and Technology Center grant, and by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), a federally funded research and development center. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, Intel, DoD, SEI, or Carnegie Mellon University. This material has been approved for public release and unlimited distribution except as restricted by copyright.

Keywords: mobile computing, cloud computing, cyber foraging, smartphones, virtual machines, system architecture, cloudlets, VM synthesis, face recognition, speech recognition, object recognition, augmented reality, simulation-based graphics

1 Introduction

VM-based cloudlets that are dispersed at the edges of the Internet and located just one WiFi hop away from associated mobile devices have been proposed as *cyber foraging* sites for an emerging class of resource-intensive yet latency-sensitive mobile applications [27]. Cyber foraging overcomes the resource limitations of wireless mobile devices by leveraging fixed infrastructure. Since it was first proposed in 2001, this technique has gained broad acceptance. MAUI [5] and CloneCloud [3] are two recent examples of systems that use cyber foraging. Flinn [6] traces the evolution of this technique and gives a comprehensive review of work in this area. For reasons discussed in Section 2.1, it will remain an important technique for the future despite mobile hardware improvements.

Mobile computing today spans many device operating systems and application environments (e.g. Android, iOS, Windows 8), as well as diverse approaches to partitioning and offloading computation. The latter range from language-specific approaches such as MAUI to legacy implementations that depend on specific back-end operating systems and runtime support [1]. There is churn in this space from new OS versions, patches to existing OS versions, new libraries, new versions of cyber foraging tools, new language runtime systems, and so on. VMs cleanly encapsulate this messy complexity, but create the problem of precisely provisioning a cloudlet from a large and continuously evolving space of VM images.

The large size of VM images complicates dynamic provisioning of cloudlets. At the same time, the presumption of *ubiquity* in mobile computing deprecates a static provisioning strategy. A mobile user expects good service for all his applications at any place and time. Wide-area physical mobility (e.g. an international traveler stepping off his flight) makes it difficult to always guarantee that a nearby cloudlet will have the precise VM image needed for cyber foraging (e.g. natural language translation with customized vocabulary and speaker-trained voice recognition via the traveler's smartphone). The VM guest state space is simply too large and too volatile for static provisioning of cloudlets at global scale. A different provisioning challenge involves the deployment of new cloudlets for load balancing, hardware upgrades, or recovery from disasters. Dynamic self provisioning of cloudlets will greatly simplify such deployments.

Rapid just-in-time provisioning of cloudlets is the focus of this paper. We show how a cloudlet can be provisioned in as little as 10 seconds with a complete copy of a new VM image that is the back-end of an offloaded application such as face recognition, object recognition or augmented reality. The compressed sizes of these VM images can range from 400 MB for a stripped-down Linux guest, to well over 2 GB for typical Windows based images. The key to rapid provisioning is the recognition that a large part of a VM image is devoted to the guest OS, software libraries, and supporting software packages. The customizations of a base system needed for a particular application are usually relatively small. Therefore, if the base VM already exists on the cloudlet, only its difference relative to the desired custom VM needs to be transferred. This approach, called *dynamic VM synthesis,* is described in Section 3. Early proof-of-concept experiments showed provisioning times of 1–2 minutes [27]. In this paper, we present optimizations that reduce this time by an order of magnitude.

Although motivated by mobile computing, dynamic VM synthesis has broader relevance. Today, public clouds such as Amazon's EC2 service are well-optimized for launching images that already exist in their storage tier, but do not provide fast options for provisioning that tier with a new, custom image. One must either launch an existing image and laboriously modify it, or suffer the long, tedious upload of the custom image. For really large images, Amazon recommends mailing a hard drive! We show that dynamic VM synthesis can rapidly provision public clouds such as EC2.

2 Background

2.1 Need for Cyber Foraging

The initial observation that mobile devices are resource-poor relative to server hardware of comparable vintage dates back to the mid-1990s [25]. Table 1, adapted from Flinn [6], illustrates the consistent large gap in the processing power of typical server and mobile device hardware between 1997 and 2011. This stubborn gap reflects a fundamental reality of user preferences: Moore's Law has to be leveraged differently on hardware that people

Year	Typical Server Processor	Speed	Typical Handheld Device	Speed
1997	Pentium® II	266 MHz	Palm Pilot	16 MHz
2002	Itanium®	1 GHz	Blackberry 5810	133 MHz
2007	Intel® Core™ 2	9.6 GHz (4 cores)	Apple iPhone	412 MHz
2011	Intel® Xeon® X5	32 GHz (2x6 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)

Table 1: Hardware Performance (adapted from Flinn [6])

	No Offload		Offload	
Application	median	99%	median	99%
SPEECH	1.22 s	6.69 s	0.23 s	1.25 s
FACE	0.42 s	4.12 s	0.16 s	1.47 s

Table 2: Speech and Face Recognition Today (Source: [15])

carry or wear for extended periods of time. This is not just a temporary limitation of current mobile hardware technology, but is intrinsic to mobility. The most sought-after features of a mobile device always include light weight, small size, long battery life, comfortable ergonomics, and tolerable heat dissipation. Processor speed, memory size, and disk capacity are secondary. For as long as our appetite for resource-intensive applications exceeds what mobile devices can sustain, cyber foraging will continue to be relevant.

Today, tasks such as free-form speech recognition, natural language translation, face recognition, object recognition, dynamic action interpretation from video, and body language interpretation lie beyond the limits of standalone mobile computing technology. Table 2, from Ha et al [15], shows the median and 99th percentile response times for speech recognition and face recognition on typical mobile hardware in 2012, with and without offloading. The results show that cyber foraging improves both the absolute response times and their variance. Looking further into the future, one can imagine cognitive assistance applications built from these primitives (such as an advanced version of IBM's Watson [35]) seamlessly augmenting human perception and cognition, and assisting attention-challenged mobile users in their real-world and cyber-world interactions. Cyber foraging will be essential to realizing such a futuristic world.

2.2 VM-based Cloudlets

Which specific part of the infrastructure should one leverage for cyber foraging? The obvious answer today is "the cloud." Public cloud infrastructure such as Amazon EC2 data centers are natural offload sites for resource-intensive computations triggered by mobile devices. Unfortunately, these large consolidated data centers are suboptimal offload sites for a growing number of resource-intensive yet latency-sensitive mobile applications. In the course of its travels, a roaming mobile device often sees multiple network hops to such a data center [33]. The resulting end-to-end latency can be large enough to seriously affect many emerging mobile applications [15]. In hostile environments such as military operations and disaster recovery, physical distance also increases vulnerability to network disruptions [14].

A cloudlet is a new architectural element for cyber foraging that represents the middle tier of a 3-tier hierarchy: mobile device – cloudlet – cloud. To serve as shared infrastructure, a cloudlet must provide safety and strong isolation between untrusted computations from different mobile devices. To sustain viable business models, it has to incorporate authentication, access control, and metering mechanisms, and has to meet service level expectations/guarantees through dynamic resource management. To be cost-effective, cloudlet deployments need to support the widest possible range of user-level computations, with minimal restrictions on their process structure, programming languages or operating systems.

In public and private clouds, these requirements are met using the VM abstraction. For precisely the same reasons VMs are also valuable as the organizing principle of cloudlets. An important difference is that cloudlets only contain soft state that is cached or otherwise re-creatable, while clouds contain both hard and soft state. In the context of this paper, the most important components of "state" are VM images. A cloudlet can thus be viewed as a "data center in a box" that "brings the cloud closer." Today, "micro data centers" from companies such as Myoonet [19] and AOL [18] are already available for repurposing as cloudlets.

3 Dynamic VM Synthesis

3.1 Speed Goal

How can a cloudlet rapidly acquire and launch a missing VM image required by a newly-associated mobile device? "Rapidly" is an imprecise quantity, ranging from a few seconds to a few hours depending on context. In the context of mobility, we use laptop suspend and resume times as a well-understood and widely-accepted definition of "rapidly." Over two decades of evolution of mobile hardware, suspend and resume times have remained stable in the 10–15 second range. This appears to be an acceptable delay for a mobile user to re-establish working context and to resume paused actions. While shorter times would be even better, 10–15 seconds is already a very difficult target for provisioning a cloudlet with a multi-gigabyte VM image.

Our figure of merit is the time delay between association and first result. It excludes the time taken for cloudlet discovery and association, but includes the time for acquiring the missing VM image, creating a VM instance from it, launching the application back-end inside the VM instance, sending it the first user input, performing the first offloaded operation, and transmitting the result to the user. In the global traveler example, this reflects the delay perceived by the user between his first utterance and the appearance of its successful translation on his smartphone display (which could be shown to someone nearby).

3.2 Basic Approach

The intuition behind our approach is that although each VM customization is unique, it is typically derived from a small set of common base systems such as a freshly-installed Windows 7 guest or Linux guest. We refer to the VM image used for offloading as a *launch VM*. It is created by installing relevant software into a *base VM*. The compressed binary difference between the base VM image and the launch VM image is called a *VM overlay*.

At run-time, *dynamic VM synthesis* (sometimes shortened to "VM synthesis" or just "synthesis") reverses the process of overlay creation. Figure 1 shows the relevant steps. A mobile device delivers the VM overlay to a cloudlet that already possesses the base VM from which this overlay was derived. The cloudlet decompresses the overlay, applies it to the base to derive the launch VM, and then creates a VM instance from it. The mobile device can now begin performing offload operations on this instance. The instance is destroyed at the end of the session, but the launch VM image can be retained in a persistent cache for future sessions. As a slight variant of this process, a mobile device can ask that cloudlet to obtain the overlay from the cloud. This indirection reduces the energy used for data transmission on the mobile device, but can improve time only when WAN bandwidth to the cloud exceeds local WiFi bandwidth.

Note that the cloudlet and mobile device can have different hardware architectures: the mobile device is merely serving as transport for the VM overlay. Normally, each offload session starts with a pristine instance of the launch VM. However, there are some use cases where modified state in the launch VM needs to be preserved for future offloads. For example, the launch VM may incorporate a machine learning model that adapts to a specific user over time. Each offload session then generates training data for an improved model that needs to be incorporated into the VM overlay for future offload sessions. This is achieved in Figure 1 by generating a *VM residue* that can be

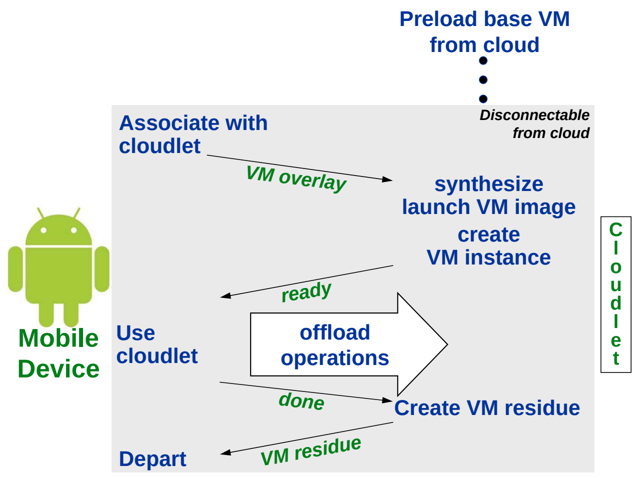


Figure 1: Dynamic VM Synthesis from Mobile Device

sent back to the mobile device and incorporated into its overlay.

There are no constraints on the guest OS of the base VM; our prototype works with both Linux and Windows. We anticipate that a relatively small number of base VMs will be popular on cloudlets at any given time. To increase the chances of success, a mobile device can carry overlays for multiple base VMs and discover the best one to use through negotiation with the cloudlet. Keep in mind that the VMs here are virtual appliances that are specifically configured for serving as the back-ends of mobile applications. They are not VMs that encapsulate general purpose desktop environments, each with a wide range of functionality.

It is useful to contrast dynamic VM synthesis with demand paging the launch VM from the mobile device or cloud using a mechanism such as the Internet Suspend/Resume system [28]. Synthesis requires the base VM to be available on the cloudlet. In contrast, demand paging works even for a freshly-created VM image that has no ancestral state on the cloudlet. Synthesis can use efficient streaming to transmit the overlay, while demand paging incurs the overhead of many small data transfers. However, some of the state that is proactively transferred in an overlay may be wasted if the launch VM includes substantial state that is not accessed. Synthesis incurs a longer startup delay before VM launch. However, once launched, the VM incurs no stalls. This may be of value for offloads of soft real-time mobile applications such as augmented reality.

It is also useful to contrast VM synthesis with launching the base VM and then performing package installations and configuration modifications to transform it into the launch VM. This is, of course, exactly what happens offline when creating the overlay; the difference is that the steps are now being performed at runtime on each association with a cloudlet. On the one hand, this approach can be attractive because the total size of install packages is often smaller than the corresponding VM overlay (e.g., Table 3) and therefore involves less transmission overhead. On the other hand, the time delay of installing the packages and performing configuration is incurred at run time. Unlike optimization of VM synthesis, which is fully under our control even if the guest is closed-source, speeding up the package installation and configuration process requires individual optimizations to many external software components. Some of those may be closed-source, proprietary components. Of even greater significance is the concern that installing a sequence of packages and then performing post-installation configuration is a fragile and error-prone task even when scripted. Defensive engineering suggests that these fragile steps be performed only once, during offline overlay creation. Once a launch VM image is correctly created offline, the synthesis process ensures that precisely the same image is re-created on each cloudlet use. This bit-exact precision of cloudlet provisioning is valuable to a mobile user, giving him high confidence that his applications will work as expected no matter where he is in the world. Finally, the installation approach requires the application to be started fresh every time. Execution state is lost between subsequent uses, destroying any sense of seamless continuity of the user experience.

OBJECT: identifies known objects and their positions in an image. Originally intended for a robotics application [32], it computes SIFT features [17] to match objects from a database, and computes position based on geometry of matched features. The backend of this application runs in a Linux environment.

FACE: detects and attempts to identify faces in an image from a prepopulated database. The algorithm uses Haar Cascades of classifiers for detection and the Eigenfaces method [36] for identification. The application backend is based on OpenCV [22] computer vision routines, and runs on a Microsoft Windows environment.

SPEECH: performs speech-to-text conversion of spoken English sentences using a Hidden Markov Model (HMM) recognition system [31]. The Java-based application backend runs on Linux.

AR: [34] is an augmented reality application that identifies buildings and landmarks in a scene captured by a phone's camera, and labels them precisely in the live view. An 80 GB database constructed from over 1000 images of 200 buildings is used to perform identification. The application backend uses multiple threads, OpenCV [22] libraries, and runs on Microsoft Windows.

FLUID: is an interactive fluid dynamics simulation, that renders a liquid sloshing in a container on the screen of a phone based on accelerometer inputs. The application backend runs on Linux and performs a smoothed particle hydrodynamics [30] physics simulation using 2218 particles, generating up to 50 frames per second. The structure of this application is representative of real-time (i.e., not turn-based) games.

Figure 2: Example Mobile Multimedia Applications

App name	Install size (MB)	Overlay Size (MB) disk	Overlay Size (MB) memory	Synthesis time (s)
OBJECT	39.5	92.8	113.3	62.8
FACE	8.3	21.8	99.2	37.0
SPEECH	64.8	106.2	111.5	63.0
AR	97.5	192.3	287.9	140.2
FLUID	0.5	1.8	14.1	7.3

Table 3: Baseline performance

3.3 Baseline Performance

We have built an instantiation of the basic VM synthesis approach, using the KVM virtual machine monitor. In our prototype, the overlay is created using the `xdelta3` binary differencing tool. Our experience has been that `xdelta3` generates smaller overlays than the native VM differencing mechanism provided by KVM. The VM overlay is then compressed using the Lempel-Ziv-Markov algorithm (LZMA), which is optimized for high compression ratios and fast decompression at the price of relatively slow compression [37]. This is an appropriate trade-off because decompression takes place in the critical path of execution at run-time and contributes to user-perceived delay. Further, compression is only done once offline but decompression occurs each time VM synthesis is performed.

We test the efficacy of VM synthesis in reducing data transfer costs and application launch times on the VM backends of several mobile applications, summarized in Figure 2. These applications, written by various researchers and described in recent literature, are the building blocks of futuristic applications that seamlessly augment human perception and cognition. These five examples of compute-intensive yet latency-sensitive applications are used in all the experiments reported in this paper. In each case, user interaction occurs on a mobile device while the compute-intensive back-end processing of each interaction occurs in a VM instance on a cloudlet.

We first construct base VM images using standard builds of Linux (Ubuntu 12.04 server) and Windows 7. These VMs are configured with 8 GB of disk and 1 GB of memory. An instance of each image is booted, and then paused; the resulting VM disk image and memory snapshot serve as a base VM. To construct a launch VM, we resume an instance of the appropriate base image, install and configure the application binaries, and launch the application. At that point, we pause the VM. The resulting disk image and memory snapshot constitute the launch VM image. As soon as an instance is created from this image, the application will be in a state ready to respond to offload requests from the mobile device — there will be no reboot delay.

The overlay for each application is the compressed binary difference between the launch VM image and its base

VM image, produced using `xdelta3` and LZMA compression. The sizes of the overlays, divided into disk and memory components, are reported in Table 3. For comparison, the size of the compressed application installation packages are also reported. Relative to VM image size, the VM synthesis approach greatly reduces the amount of data that must be transferred to create VM instances. Compared to the launch VM images (nominal 8 GB disk image plus memory snapshot), Table 3 shows that overlays are an order of magnitude smaller. While they are larger than the install packages from which they were derived, VM synthesis eliminates the fragile and error-prone process of runtime package installation and configuration as discussed in Section 3.2. In fact, as we show later in Section 8, provisioning using the most optimized version of VM synthesis is faster than runtime installation and configuration.

The total time to perform VM synthesis is also reported in Table 3. These times were measured using a netbook, which serves the application overlays to a desktop machine with a quad-core Intel® Core™ i7-3770 processor. The latter acts as a cloudlet, which performs synthesis and executes the application VMs. For each application the total time reported includes the time needed to transfer the overlay across 802.11n WiFi (38 Mbps measured average bandwidth), decompress it, apply the overlay to the base image, and resume the constructed application image. We use this hardware setup for all of the experiments in this paper.

Although this baseline implementation of VM synthesis achieves bit-exact provisioning without transferring full VM images, its performance falls far short of our speed goal. Table 3 shows that only one of the applications studied (FLUID) comes close to our goal of 10–15 seconds. A synthesis time of 60 to 200 seconds is more typical for other applications that we have studied.

In the rest of this paper, we present a multi-pronged approach to accelerating VM synthesis. We first reduce the size of the overlay using aggressive deduplication (Section 4) and by bridging the semantic gap between the VMM and guest OS (Section 5). We then accelerate the launch of the VM image by pipelining its synthesis (Section 6), and by optimistically launching before synthesis is complete (Section 7). The results presented in each of these sections shows the speedup attributable to that optimization.

4 Deduplication

4.1 Concept

Our first optimization leverages the fact that there are many sources of data redundancy in a VM overlay. Through deduplication we can eliminate this redundancy and thus shrink the overlay. A smaller overlay incurs less transmission delay and also consumes less energy on the mobile device for transmission. Even if performing deduplication is expensive, it is only an offline cost that is paid once during overlay creation. The overhead of re-inflating deduplicated data during synthesis is trivial since the cloudlet is a powerful machine that is not energy-constrained.

From a number of sources, we can anticipate some duplication of data between the memory snapshot and the disk image of the launch VM. For example, at the moment the launch VM is suspended during overlay construction, the I/O buffer cache of the guest OS contains some data that is also present in its virtual disk. Additionally, data from some files on the virtual disk may have been read by the application back-end into its virtual memory during initialization. Further, depending on the runtime specifics of the programming language in which the application is written, there may be copies of variable initialization data both in virtual memory and on virtual disk. These are only a few of the many sources of data duplication between the memory snapshot and the disk image of the launch VM.

Separately, we can also expect some duplication of data between the overlay and the base VM (which is already present on the cloudlet). Recall that the baseline implementation in Section 3.2 creates a VM overlay by constructing a binary delta between a base VM and the launch VM that is derived from the base VM. This binary delta does not capture duplicate data that has been copied or relocated within the memory image or disk image of the modified VM. Indeed, it can not even take advantage of the fact that many parts of memory should be identical to disk because they are loaded from disk originally—executables, shared libraries, etc. It would be much more efficient to begin with a list of modifications within the modified VM and then perform deduplication to further

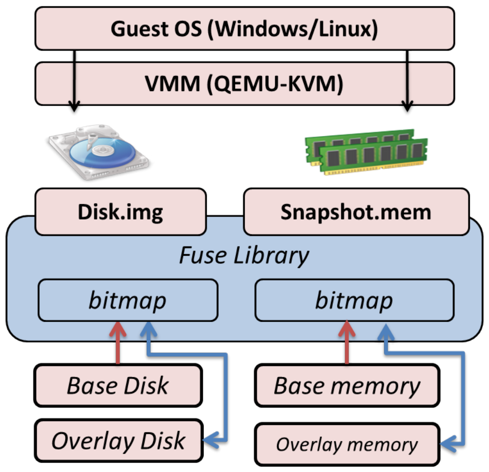


Figure 3: FUSE Interpositioning for Deduplication

reduce this list to contain the minimal set of information needed to transform a base VM into the modified VM. If we could find this minimal set, then we could construct smaller VM overlays.

4.2 Implementation

The choice of granularity is a key design decision for deduplication. Too large a granularity will tend to miss many small regions that are identical. Very small granularity will detect these small regions, but incur large overhead in the data representation. Our choice is a chunk size of 4 KB, because it is a widely-used page size for many popular operating systems today. For example, current versions of Linux, Mac OS X, and Windows all use a 4 KB page size. An additional benefit of deduplicating at this granularity is that most operating systems use Direct Memory Access (DMA) for I/O, which means the disk is accessed with memory page size granularity. Thus, the 4 KB chunk size is likely to work well for both memory and disk deduplication.

To discover the modified portions of disk and memory associated with the process of creating a launch VM, we introduce a shim layer between the VMM and the backing files for virtual disk and memory using FUSE as shown in Figure 3. This approach requires no changes to the VMM. We reuse this FUSE shim layer at VM synthesis time to avoid the data copying that would be required to physically merge the overlay virtual disk into the base disk.

During the installation and configuration steps of launch VM construction, the shim layer exposes I/O requests from the VMM to the virtual disk file and memory snapshot file. On every write to either the virtual disk or memory snapshot, we redirect the write to the corresponding overlay file and marks a bitmap indicating this chunk has changed. When reads occur at a later point in time, we consult this bitmap to figure out if the read should be serviced from the original base files, or from the new overlay files. We have found the FUSE overhead to be negligible for virtual disk accesses despite the fact that it is on the critical read and write paths from the VM to its disk. However, memory operations would become prohibitively expensive with this additional component. We therefore do not use FUSE to capture memory changes. (However, we still use it during synthesis to avoid reconstructing the entire launch memory snapshot.) Rather we capture the entire memory snapshot only after we finish customizing the launch VM. We then interpret this memory snapshot, and compare it with *base memory* to obtain the modified memory chunks and corresponding bitmap.

Once we have a list of modified disk and memory chunks, we perform deduplication by computing SHA-256 hashes [10] of their contents. We use these hashes to construct a unique set of pages which are not contained within the base VM and must be included in the transmitted overlay. We construct the set of unique modified disk and memory chunks using five comparison rules: (1) compare to base VM disk chunks, (2) compare to base VM memory chunks, (3) compare to other chunks within itself (within modified disk or modified memory respectively), (4) compare to zero filled chunk, and (5) compare between modified memory and modified disk. These five comparison rules capture various scenarios that are frequent sources of data redundancy, as discussed in

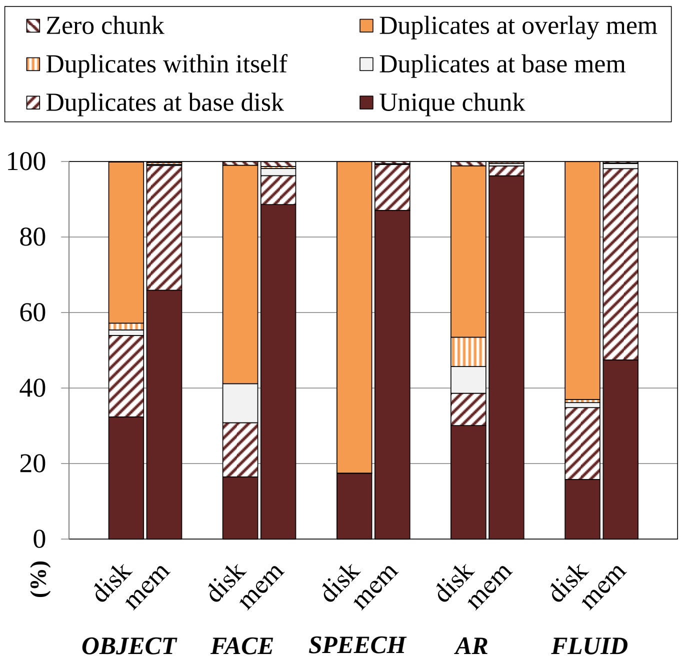


Figure 4: Benefit of Deduplication

Section 4.1.

For each unique chunk, we compare it to the corresponding chunk (same position on disk or in memory) in the base VM. We use the `xdelta3` algorithm to compute a binary delta of the chunk and transmit only the delta if it is smaller in size than the chunk. The intuition here is that even if the hashes mismatch, there might be significant overlap at a finer byte granularity which a binary delta algorithm would leverage.

4.3 Evaluation

Figure 4 shows the benefit of deduplication for the overlay of each application. For any deduplication between memory and disk, we choose to only retain the duplicated chunks within the memory image. For deduplication purposes, it does not matter if we make the canonical chunk reside within disk or memory, but for representing results we chose the memory image as the canonical source of chunks.

Averaged across the five applications, only 22% of the modified disk and 77% of the modified memory is unique. The biggest source of redundancy is between modified memory and modified disk: each application exhibits greater than 58% duplication, with SPEECH exhibiting 83% duplication. The base disk is the second biggest source of the duplication. On average, 13% of the modified disk and 21% of the modified memory are identical with the base disk. We analyzed files associated with the duplicated chunks for the OBJECT application. Our findings are consistent with our intuition: most of the associated files in the modified disk are shared resources located within the `/usr/shared/`, `/usr/lib/`, and `/var/lib/` directories, and a large portion of the files are shared libraries such as `libgdk-x11`, `libX11-xcb`, and `libjpeg`. The overlay memory shows similar results, but it also includes copies of executed binaries such as `wget`, `sudo`, `xz`, `dpkg-trigger`, and `dpkg-deb` in addition to shared libraries.

5 Bridging the Semantic Gap

5.1 Concept

The strong boundary enforced by VM technology between the guest and host environments is a double-edged sword. On the one hand, this strong boundary ensures isolation between the host, the guest, and other guests. On the other hand, it forces the host to view each guest as a black box, whose disk and memory contents cannot be interpreted in terms of higher-level abstractions such as files or application-level data structures.

This inability to span the semantic gap between low-level representations of memory and disk, and higher-level abstractions can be problematic when constructing VM overlays. For example, suppose a guest application downloads a 100 MB file, and later deletes it. Ideally, this should result in no increase in the size of the VM overlay. However, the VMM will see up to 200 MB of modifications: 100 MB of changed disk state, and 100 MB of changed memory state. This is because the file data moves through the in-memory I/O buffer cache of the guest OS before reaching the disk, effectively modifying both memory state and disk state. When the file is deleted, the guest OS marks the disk blocks and corresponding page cache entries as free, but their (now garbage) contents remain. To the VMM, this is indistinguishable from important state modifications that need to be preserved. Deduplication (described in Section 4) can cut this state in half, but we would still unnecessarily add 100 MB to the overlay.

Ideally, only the state that actually matters to the guest should be included in the overlay. When files are deleted or memory pages freed, none of their contents should be incorporated into the overlay. In essence, we need semantic knowledge from the guest regarding what state needs to be preserved and what can be discarded. In separate sections below, we show how this semantic gap can bridged for disk and memory state.

5.2 Implementation: Disk

To accurately account for disk blocks that are garbage, we need either (1) a method of communicating this information from the guest OS to the host, or (2) a method of scanning the contents of the file system on the virtual disk to glean this OS-level information. The first approach requires guest support, and may not be possible for every guest OS. The second approach requires no guest support, but does require an understanding of the on-disk file system format. Both approaches may be used in tandem to cross-check their results.

Exploiting TRIM support: The TRIM command in the ATA standard enables an OS to inform a disk which sectors are no longer in use. This command is important for modern devices such as Solid State Drives (SSDs) which implement logic to aggressively remap writes to unused sectors. Wear-leveling algorithms and garbage collection inside of SSDs use this knowledge to increase write performance and device life.

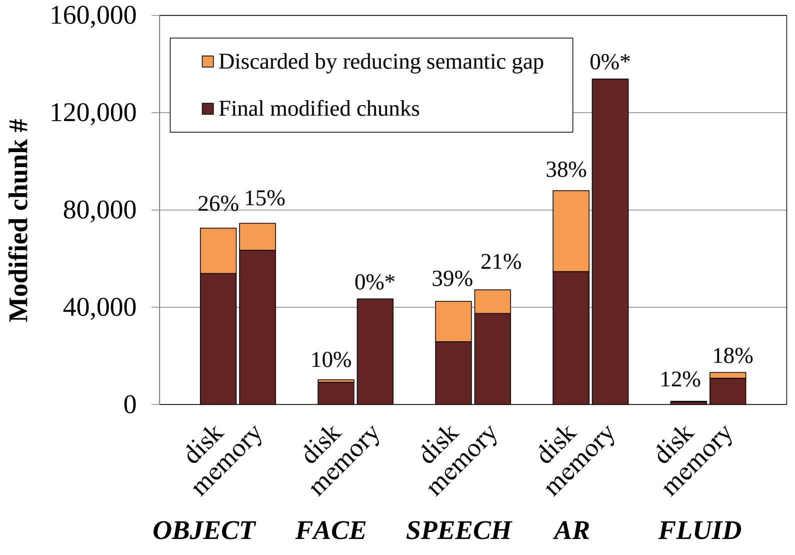
The TRIM command provides precisely the mechanism we desire — an industry-standard mechanism for communicating semantic information about unused sectors from an OS to the underlying hardware. We can exploit this mechanism to communicate free disk block information from the guest OS to the host to reduce VM overlay size. We modify the VMM (KVM/QEMU) to capture TRIM events and to log these over a named pipe to our overlay generation code. When generating the overlay, we merge this TRIM log with a trace of sector writes by timestamp to determine which blocks are free when the VM is suspended; these blocks can be safely omitted from the overlay. To make use of this technique, we simply need to ensure that TRIM support is enabled in the guest OS. As TRIM is an industry standard, it is supported by almost all modern operating systems, including recent Linux distributions and Windows 7.

Introspecting the file system: An alternative approach is to use knowledge of the on-disk file system format to directly inspect the contents of a virtual disk and determine which blocks are currently unused. Many file systems maintain lists of free blocks forming a canonical set of blocks which should not be included in an overlay. In the worst case, the entire file system can be crawled to determine which blocks are in use by files within the file system. Although this approach is file-system-specific, it avoids the need to communicate information from a running guest, or to carefully trace TRIM and write events.

We implement introspection of free disk blocks via a static analysis tool [23]. This tool reads and interprets a virtual disk image and produces a list of free blocks. Our tool has support for the `ext2/3/4` family of Linux file systems and the `NTFS` file system for Windows.

5.3 Implementation: Memory

It is difficult to determine which memory pages are considered free by a guest OS. Although the VMM can inspect the page tables, this is not sufficient to determine if a page is in use because unmapped pages are not necessarily



*Our implementation cannot determine free memory pages for Windows guests.

Figure 5: Savings by Closing the Semantic Gap

free. Inspecting page contents is also not good enough, because free pages normally contain random data and are not zeroed.

To bridge this gap, there are two natural approaches: (1) communicate free page information from the guest OS to the host, or (2) interpret memory layout data structures maintained by the guest OS. Unfortunately, there is no standard way of accomplishing the first approach (i.e., no memory counterpart to TRIM support), so we focus our efforts on the second approach. In order to obtain the list of free memory pages we first introduce a tiny kernel module into Linux guests. This module exposes the memory addresses of two data structures for memory management through the `/proc` file system in the guest. We suspend the VM, and feed these addresses and the memory snapshot to an offline scanning program. This scanning program reads the memory snapshot and parses the memory management data structures at the specified addresses to identify the free pages.

Since our approach requires modifying the guest OS, it is not usable on closed-source OSs such as Windows. Further, in-memory data formats tend to be highly volatile across OS releases, and to evolve much more rapidly than file system formats. Even an open-source kernel such as Linux will require significant maintenance effort to track these changes.

Other techniques could be employed to infer free pages without the need for guest support. For example, a VMM could monitor memory accesses since the guest's boot and keep track of pages that have been touched. This would avoid guest modification at the cost of lower fidelity—some of the pages reported as used could have been touched, but later freed. Perhaps with the advent of Non-Volatile Memories (NVMs), which provide persistent storage with memory-like, byte-addressable interfaces, there may be a need to introduce a standardized TRIM-like feature for memory. Such support would make it possible to bridge the memory semantic gap in an OS-agnostic way in the future.

5.4 Evaluation

For the disk semantic gap, our experiments show that the TRIM and introspection approaches produce nearly identical results. Just a few additional free blocks are found by the introspection approach that were not captured by TRIM.

Figure 5 shows how much we gain by closing the disk semantic gap. For each application we construct the VM image by downloading its installation package, installing it, and then deleting the installation package. We therefore expect our approach to find and discard the blocks that held the installation package, reducing overlay size by approximately the installation package size. Our results confirm this for all of the applications except

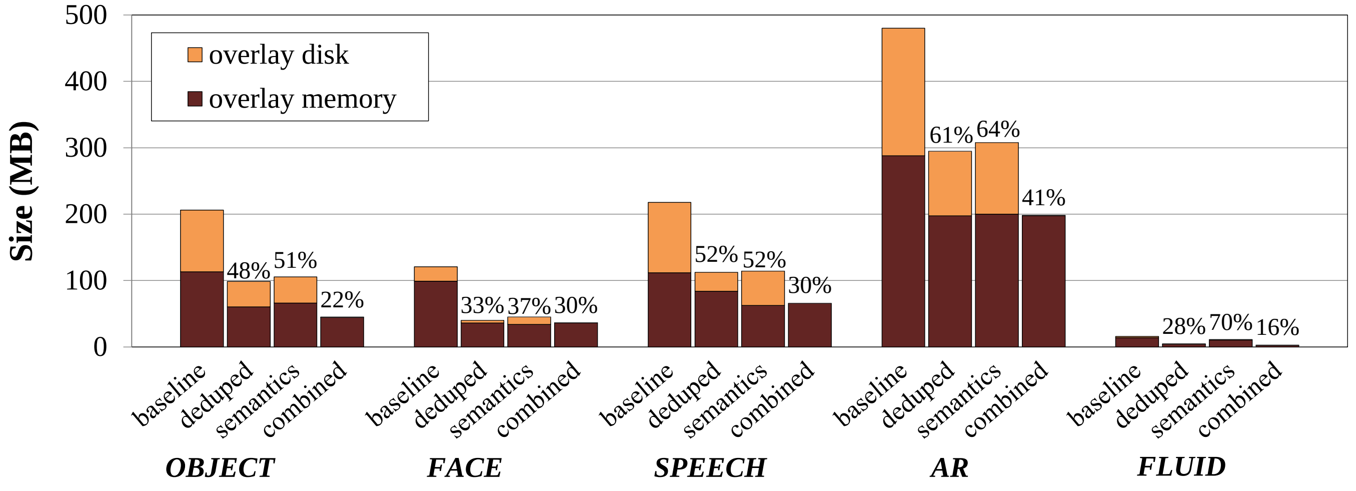


Figure 6: Overlay Size Compared to Baseline

one: for FACE, the semantically discarded disk blocks together were smaller than the installation package. On investigation, we found that this was due to the freed blocks being reused post-install. On average, across the five applications, bridging the disk semantic gap allows 25% of modified disk chunks to be omitted from the overlay.

Figure 5 also shows the savings we can achieve by discarding free memory pages from the VM overlay. We can discard on average 18% of modified memory chunks for the Linux applications OBJECT, SPEECH, and FLUID. Since our implementation is limited to Linux, we cannot reduce the memory overlays for the two Windows-based applications (FACE and AR).

Combining deduplication and bridging of the semantic gap can be highly effective in reducing the VM overlay size. Figure 6 shows VM overlay size with each optimization individually represented, and also combined together. The “baseline” represents VM overlay size using the approach described in Section 3. The bar labeled “deduped” is the VM overlay with deduplication applied; “semantics” is the VM overlay with semantic knowledge applied (only disk for Windows applications); and, “combined” is the VM overlay size with both optimizations applied. On average compared to the baseline implementation, the deduplication optimization reduces the VM overlay size to 44%. Using semantic knowledge reduces the VM overlay size to 55% of its baseline size. Both optimizations applied together reduce overlay size to 20% of baseline.

The final overlay disk almost disappears when we combine both optimizations. This is because a large portion of disk chunks are associated with installation packages. Recall that to install each application, we first download an installation package in the VM and remove it later when it finishes installation. This installation file is already compressed, so further compression does little. In addition, this newly introduced data is less likely to be duplicated inside the base VM. Therefore, applying semantic knowledge removes most of the unique chunks not found by deduplication. For example in AR, 25,887 unique chunks remained after deduplication, but 96% of them are discarded by applying semantic knowledge.

6 Pipelining

6.1 Concept

There are three time-consuming steps in VM synthesis. First, the VM overlay is transferred across the network. Next, the VM overlay is decompressed. Finally, the decompressed VM overlay is applied to the base VM (i.e., `xdelta3` in reverse). These steps are serialized because we need the output of the preceding step as input to the next one. Figure 7 illustrates this serialized process for the disk and memory components of the overlay. This serialization adds significantly to the VM start latency on a cloudlet. If we could begin the later steps before the preceding ones complete, we would be pipelining the whole process and thereby shrink the total time for synthesis.

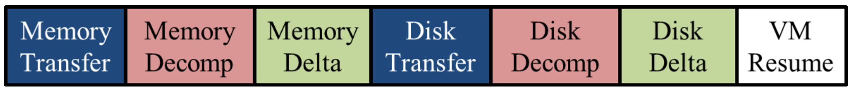


Figure 7: Baseline VM Synthesis

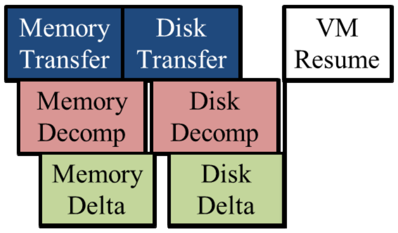


Figure 8: Pipelined VM Synthesis

Figure 8 illustrates the pipelined transformation of Figure 7.

6.2 Implementation

The implementation follows directly from the pipelining concept. We split the VM overlay into a set of segments and operate on each segment independently. The VM synthesis steps can now be pipelined. The decompression of a particular segment starts as soon as it is transferred, and happens in parallel with the transfer of the next segment. Likewise, the application of a particular overlay segment to the base VM proceeds in parallel with the decompression of the next segment. Given sufficiently small segment size, the total time will approach that of the bottleneck step (typically the transfer time), plus any serial steps such as VM instance creation and launch.

6.3 Evaluation

Figure 9 compares the performance of the baseline synthesis approach to an optimized one that combines deduplication, semantic gap closing, and pipelining. The results confirm that once pipelining is introduced, transfer time becomes the dominant contributor to the total synthesis time. The synthesis time shown in Figure 9 includes all of the time needed to get the VM to the point where it is fully resumed and ready to accept offload requests from the mobile device. Two applications now launch within 10 seconds (FACE and FLUID), while two others launch within 15 seconds (OBJECT and SPEECH). Only AR takes much longer (44 seconds), but this is because its overlay size and, therefore, transfer time remains high. On average, we observe a 3x–5x speedup compared to the baseline VM synthesis approach from Section 3.

All of these experiments were conducted using 802.11n WiFi at 2.4 GHz, with an average measured throughput of 38 Mbps. We expect these times to improve in the future as new wireless technologies and network optimizations are introduced, thus increasing the bandwidth of WiFi networks. In other words, VM synthesis is now directly correlated to network bandwidth. While WAN bandwidth improvements require large infrastructure changes, mobile bandwidth to the wireless AP at a cloudlet only requires localized hardware and software changes. New WiFi standards such as 802.11ac promise up to 500 Mbps and are actively being deployed [38]. Recent research [11] also demonstrates methods of increasing bandwidth up to 700% with software-level changes for WiFi networks facing contention. Thus, both industry and research are focused on increasing WiFi bandwidth. This directly translates into faster VM synthesis. Based on our measurements, until actual transfer times improve by 3x, the transfer stage will remain the bottleneck (assuming the cloudlet processor remains constant). Beyond this, we will need to parallelize the decompression and overlay application stages across multiple cores to benefit from further improvements in network bandwidth.

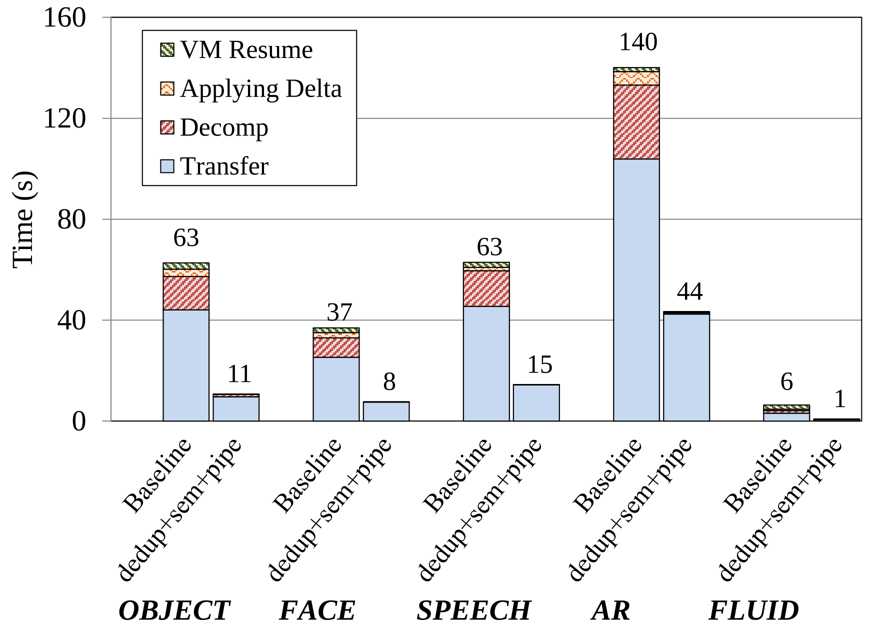


Figure 9: Effect of Pipelining + Earlier Optimizations

7 Early Start

7.1 Concept

We have shown that the optimizations described in the previous sections greatly reduce the size of overlays and streamline their transfer and processing. For several of the applications, the optimized overlay size is close to the size of the install image. Hence, there is little scope for further reducing size to improve launch times. Instead, we consider whether one really needs to transfer the entire overlay before launching the VM instance. This may not be necessary for a number of reasons. For example, during the overlay creation process, the guest OS was already booted up and the application was already launched at the point when the VM was suspended. Any state that is used only during guest boot-up or application initialization will not be needed again. As another example, some VM state may only be accessed during exception handling or other rare events. One can optimistically bet that these circumstances will not occur immediately after VM instance creation.

The potential benefits of optimism can be significant. Table 4 shows the percentage of chunks in the overlay that are actually accessed by the five benchmark applications between VM launch and completion of first request. A substantial number of chunks are not used immediately. We can speed up VM launch by transferring just the needed chunks first, synthesizing only those parts of the launch VM, and then creating the VM instance. The transfer of the missing parts of the overlay and synthesis of the rest of the launch VM can continue in the background until it is completed.

7.2 Implementation

We explored a number of alternatives in translating the concept of early start into a viable implementation. One option is to profile the resume of the launch VM, and order the chunks in the overlay accordingly. When offloading, the VM is resumed concurrently with the synthesis operations. If the VM attempts to access chunks that have not yet been synthesized, it will be blocked until the chunk becomes available. If the order of chunks is correct, the VM can begin running significantly before the VM synthesis completes.

Unfortunately, it is difficult to get this order perfectly right. In our early experiments, multiple profiling runs produced slightly different chunk access patterns. In particular a small number of chunks may be accessed early in one run, but not at all in another. With a large number of chunks, it is unlikely that every chunk that is needed early

	OBJECT	FACE	SPEECH	AR	FLUID
% chunks	17.4	56.9	26.8	65.2	27.1
% MB	30.6	63.0	33.0	87.9	50.3

This table shows the fraction of the overlay accessed between VM launch and completion of first request from the mobile device.

Table 4: Percentage of Overlay Needed Initially

in an actual VM resume will have been picked up in the profiling. More likely, one or more of these chunks will be missed in profiling, and will be placed near the end of the overlay. Getting even one chunk wrong can force the VM to wait for all chunks to be transferred and VM synthesis to complete.

Alternatively, we can avoid trying to predict the chunk access order by using a demand fetching approach. Here, the VM is started first, and the portions of the overlay needed to synthesize accessed chunks are fetched on demand from the mobile device. Unfortunately, this approach, too, has some issues. Demand fetching individual chunks (which can be very small due to deduplication and delta encoding) requires many small network transfers, with a round trip penalty imposed on each, resulting in poor effective bandwidth and slow transfers. To alleviate this, we can cut the overlay into larger segments comprised of many chunks, and perform demand fetching at segment granularities. This will help amortize the demand fetching costs, but leaves open the question of sizing the segments. Smaller segments lets one fetch more closely just the needed chunks. Larger segments, in addition to being more bandwidth friendly, can achieve better compression ratios, but will be less selective in transferring just what is needed. Finally, how chunks are grouped into segments can also significantly influence performance. For example, if needed chunks are randomly distributed among segments, one will likely need to transfer the entire overlay to run the VM.

Our implementation uses a hybrid approach. It makes a reasonable attempt to order the chunks according to a profiled access pattern computed offline. We then break the overlay into segments. During offload, we start the VM and begin streaming the segments in order, but also allow out-of-order demand fetches of segments to preempt the original ordering. Thus, we use demand fetching to retrieve chunks that were not predicted by the profiling, while simultaneously bulk-streaming segments in a work-conserving manner to quickly transfer and synthesize all chunks. This approach bears some resemblance to classic prefetching with out-of-band handling of demand misses. However, it should be kept in mind that these concepts are being applied here to an overlay rather than a VM image.

Figure 10 illustrates our implementation of this hybrid approach. A critical issue is that all of the widely used VMMs, including KVM, Xen, VirtualBox, and VMware, require the entire memory snapshot before resuming a VM, hindering early start. So, we first modify the VMM (KVM in our case) to resume a VM without first reading in the entire memory snapshot. Rather, it now memory maps the snapshot file, so portions are implicitly loaded when accessed. We then implement a FUSE file system that hosts the VM disk image and memory snapshot. This routes disk accesses of the VMM to our user-level code that can perform just-in-time VM synthesis on the accessed chunks (disk or memory). Our code consults a small bitmap that indicates whether the particular chunk needs to be served from the base image or the overlay. If the overlay is needed, then a local cache of processed chunks is checked. If the chunk is not available, a demand-fetch of the needed overlay segment is issued to the mobile device. Concurrently, in the background, the code processes the stream of overlay segments as it is received from the mobile device. With this implementation, only the small bitmap assigning chunks to overlay or base image needs to be transferred before the VM is launched.

7.3 Evaluation

We evaluate our early start approach with a few different combinations of segment size and chunk order. We test with small (approx. 64KB) and medium (approx. 1MB) sized segments, as well as with just a single segment comprising the entire overlay. (The latter effectively disables demand-fetching). We also test with chunks sorted by access-order (based on a single profiling run of each application) and offset-order (with memory before disk chunks). We slightly modify the ordering so that duplicate chunks are contained within the same segment, avoiding

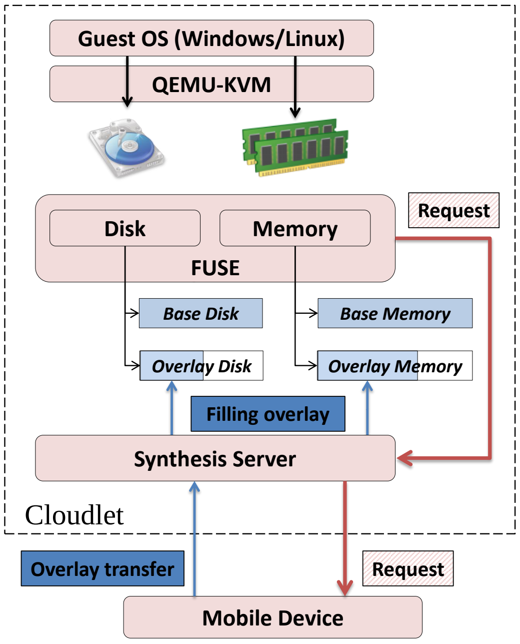


Figure 10: System Implementation for Early Start

any need for pointer-chasing between segments when handling deduplication.

With early-start, the VM synthesis time itself is less relevant. Rather, the metric we use is the first response time of the application. The response time is measured from the initial VM start request to when the first reply returns to the client, thus including the overlapped transfer time, synthesis time, and application execution time. Figure 11 compares performance of early start for different chunk ordering policies and segment sizes. The values are normalized to the first response time when the VM begins execution once VM synthesis (including all of the other optimizations discussed previously) completes. Access ordering alone does not help significantly due to the inaccuracies of the single profiling runs. However, access-ordered chunks with demand fetching with 64 KB or 1 MB segments can significantly reduce first response times. We see up to 60% reduction for *SPEECH* and *OBJECT*. For *FLUID*, the response time without early start is already so short that small fluctuations due to compression and network affects the normalized response time adversely. *AR*, which requires close to 90% of the data in its overlay, does not benefit much from early start.

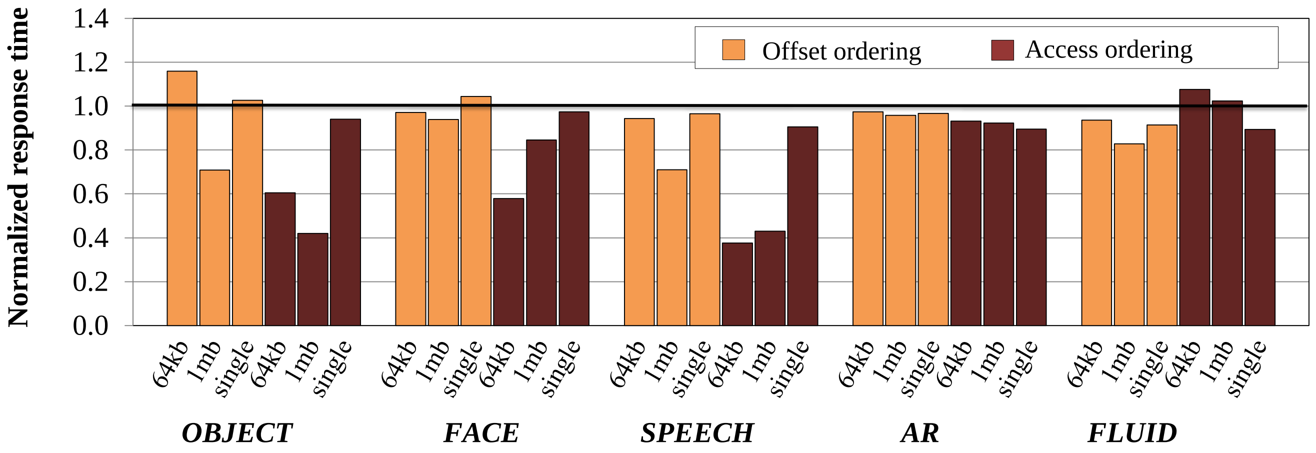


Figure 11: Normalized First Response Time for Early start

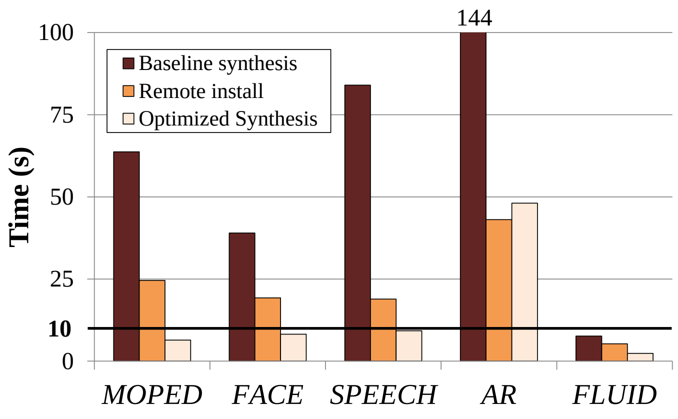


Figure 12: First response times

8 Fully Optimized VM Synthesis

We have shown that all of the various techniques for improving VM synthesis described in this paper work quite well on their own. One may wonder: how effective are these techniques when combined? In this section we evaluate a complete, fully-optimized implementation of VM synthesis incorporating all of the improvements described in this paper. The figure of merit here is the total latency as perceived by the user, from the beginning of the application offload process to when the first reply is returned. This first response metric is dependent on the time consumed by the offloading operation and launch of the application VM.

For this experiment, we construct minimal application overlays via deduplication and by preserving only the semantically meaningful chunks. To minimize the time required, we employ pipelining of the synthesis steps, and employ early start of the VM using a overlay segment size of 1 MB (which provides a good tradeoff between demand fetch granularity and good compression).

Figure 12 shows the improvement in first-response times with our fully-optimized VM synthesis over the baseline version described in Section 3. Overall, we improved performance of VM synthesis by a factor of 3 to 8 across these applications. Except for the AR application, we did better than our original goal of 10–15 s first-response latencies.

We also compare our results to the first-response time for a remote installation approach to running a custom VM image. This involves launching a standard VM, uploading and installing the application packages, and then executing the custom applications. In Section 3.3, we have already dismissed this approach on qualitative grounds; in particular, even scripted install can be fragile, the resulting configuration is not identical every time, and the application is restarted every time so execution state is not preserved. The only redeeming quality is that the install packages tend to be smaller than the baseline VM overlays, potentially making the remote install faster. Here, we use highly optimized application packages that are self-contained (including needed libraries, or statically-compiled binaries), and fully-scripted installation to show the remote install approach at its fastest.

As we can see from Figure 12, however, our optimized VM synthesis approach produces significantly better first-response times than remote install in all but one case. In that case, the two approaches are basically a tie. Thus, our optimized VM synthesis approach can achieve very fast offload and execution of custom application VMs on cloudlets, yet maintain strong guarantees on their reconstructed state.

9 VM Synthesis on Amazon EC2

In this paper, we have presented VM synthesis as a technique to rapidly offload customized application VMs to cloudlet infrastructure near a mobile device. However, the technique is much more general than this, and can help whenever one wishes to transfer VM state across a bottleneck network. In particular, VM synthesis can significantly speed up the upload and launch of a custom VM on commercial cloud services from across a WAN.

1. Create a new EC2 VM instance from an existing Amazon VM image,
2. Attach a cloud block device with VM synthesis tools and base VM image,
3. Change the root file system of the instance to the attached block device,
4. Perform **VM synthesis** over the WAN to construct the modified VM disk,
5. Mount the modified VM disk,
6. Synchronize / copy the modified file system with the instance's original,
7. Detach block device,
8. Reboot with the customized file system.

Figure 13: Steps in VM Synthesis for EC2

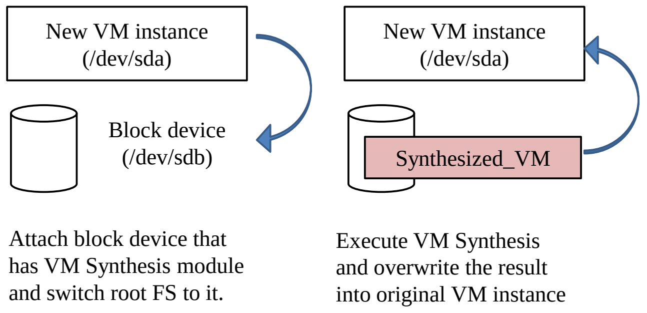

New VM instance (/dev/sda)
Block device (/dev/sdb)
New VM instance (/dev/sda)
Synthesized_VM
Attach block device that has VM Synthesis module and switch root FS to it.
Execute VM Synthesis and overwrite the result into original VM instance

Figure 14: VM Synthesis for EC2

Here, we describe our VM synthesis solution for Amazon's public EC2 cloud.

The normal cloud workflow to launch a customized VM involves three steps: (1) construct the VM image, including installing custom software and libraries, and making requisite configuration changes; (2) upload the VM image to the cloud, a step largely limited by the client to cloud bandwidth; and (3) launch and execute a VM instance based on the uploaded VM image, a step that depends on the cloud provider's backend scheduling and resources. VM synthesis promises to speed up the second step by reducing the amount of state uploaded to a cloud.

Today, no cloud supports VM synthesis as a primitive operation. In our EC2 implementation, we perform VM synthesis entirely within a running VM instance. EC2 does not allow external access to the disk or memory image of an instance, so we cannot manipulate the saved state of a paused instance to effect synthesis. We also cannot generate a data file, treat it as a VM image, and launch an instance based on it. We work around these limitations by performing VM synthesis within a live instance, which modifies its own state and then reboots into the custom VM environment. Assuming that the base VM image and synthesis tools have already been uploaded, synthesis proceeds in the eight steps shown in Figure 13. Steps 1-3 occur on the left hand side of Figure 14, and steps 3-8 occuring on the right hand side of that figure. We do not handle the memory portion of a VM in EC2 because we do not have access to the raw memory image. This requires an unnecessary reboot and wasted time in synchronizing file systems. If EC2 had a VM synthesis primitive, the memory image and VM disk could be directly exposed by their infrastructure and only step 4 would remain; the VM overlay would be transmitted, applied, and then the VM could be directly resumed without reboot.

We compare the time it takes to perform VM synthesis to the time required in the normal cloud workflow to deploy and execute a custom VM with the OBJECT application. The results are shown in Table 5. For VM synthesis, synthesis setup corresponds to steps 1-3, uploading to step 4, and post-processing corresponds to steps 5-8. synthesis setup because it uploading. With the normal Amazon workflow, there is no analog to sythesis setup. However, after upload, Amazon takes time to provision resources and boot a VM within EC2; this is included in

	10 Mbps		100 Mbps	
	Synthesis	Amazon	Synthesis	Amazon
Synthesis Setup	44 s	—	46 s	—
Uploading[†]	36 s	607 s	8 s	204 s
Post-processing	96 s	139 s	97 s	105 s
Total	180 s	746 s	154 s	310 s

[†]Upload time for VM synthesis includes all processing for synthesis including overlay transfer, decompression, and applying delta.

Table 5: Time for Instantiating Custom VM at Amazon EC2

the total post-processing time. We present results for two WAN bandwidths, 10 Mbps and 100 Mbps, in Table 5. In both cases, VM synthesis wins over the normal cloud workflow with a 4x improvement in the 10 Mbps case and a 2x improvement in the 100 Mbps case. The normal cloud workflow is bottlenecked on bandwidth because it must upload the full 514 MB compressed VM image, but VM synthesis reduces this to a much more compact 42 MB VM overlay. It is important to note here that pre- and post-processing for VM synthesis are artificially inflated because of the lack of native VM synthesis support and the convoluted mechanisms we needed to employ to work around limitations imposed by EC2.

10 Related Work

Offloading computation has a long history in mobile computing, especially to improve application performance and battery life [9, 21, 24]. The broader concept of cyber foraging, or "living off the land" by leveraging nearby computational and data storage resources, was first articulated in 2001 [26]. In that work, the proximity of the helper resources, known as "surrogates," to the mobile device was intuitively assumed, but how to provision them was left as future work. Since then, different aspects of cyber foraging have been explored by a number of researchers. Some of these efforts have looked at the tradeoffs between different goals such as execution speed and energy usage based on adaptive resource-based decisions on local versus remote execution [7, 8]. Other efforts have looked at the problem of estimating resource usage of a future operation based on past observations, and used this estimate to pick the optimal execution site and fidelity setting [12, 13, 20]. Many researchers have explored the problem of how best to partition an application between local and remote execution components, along with development of language-level and runtime tools to support this partitioning [1, 3, 5].

Since 2008, offloading computation from a mobile device over the wide-area Internet to a cloud computing service such as Amazon EC2 [29] has become possible. In virtualized clouds, bridging the semantic gap has been proposed before to provide novel VMM features [16], and, in the case of Kaleidoscope, quickly provisioning cloud resources via VM cloning [2]. But, cloud computing places surrogates far away across a multi-hop WAN rather than nearby on a single-hop WLAN. A 2009 position paper [27] introduced VM-based surrogate infrastructure called "cloudlets." Proximity of offload infrastructure was deemed essential for deeply immersive applications where crisp interactive response requires end-to-end latency to be as low as possible. A recent paper [4] reported on user studies that explored the need for proximity of offload infrastructure when using a large static display in conjunction with a mobile device for thin client (VNC) interaction.

11 Conclusion

Beyond today's familiar desktop, laptop and smartphone applications is a new genre of software seamlessly augmenting human perception and cognition. Supporting the compute-intensive and latency sensitive applications typical of this genre requires the ability to offload computation from mobile devices to widely dispersed cloud infrastructure, a.k.a cloudlets. Physical dispersion of cloudlets makes their provisioning a challenge. In this paper, we have shown how cloudlets can be rapidly and precisely provisioned by a mobile device to meet its exact needs just before use. We have also shown that although our solution, dynamic VM synthesis, was inspired by the specific demands of mobile computing, it also has broader relevance to public cloud computing infrastructure.

Acknowledgements

This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0833882 and IIS-1065336, by an Intel Science and Technology Center grant, and by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), a federally funded research and development center. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, Intel, DoD, SEI, or Carnegie Mellon University. This material has been approved for public release and unlimited distribution except as restricted by copyright.

References

[1] R. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proceedings of the 5th International Conference on Mobile Systems Applications and Services*, San Juan, Puerto Rico, June 2007.

[2] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *Proceedings of the sixth conference on Computer systems*, EuroSys ’11, pages 273–286, New York, NY, USA, 2011. ACM.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of EuroSys 2011*, Salzburg, Switzerland, April 2011.

[4] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan. How Close is Close Enough? Understanding the Role of Cloudlets in Supporting Display Appropriation by Mobile Users. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2012)*, Lugano, Switzerland, March 2012.

[5] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of MobiSys 2010*, San Francisco, CA, June 2010.

[6] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012.

[7] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[8] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy Conservation and Application Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[9] J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, December 1999.

[10] Gallagher, P. Secure Hash Standard (SHS), 2008. `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[11] A. Gupta, J. Min, and I. Rhee. WiFox: Scaling WiFi Performance for Large Audience Environments. In *Proceedings of the Eigth COnference on emerging Networking EXperiments and Technologies*, CoNEXT ’12, New York, NY, USA, 2012. ACM.

[12] S. Gurun, C. Krintz, and R. Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In *Proceedings of MobiSys 2004*, Boston, MA, June 2004.

[13] S. Gurun, C. Krintz, and R. Wolski. NWSLite: A General-Purpose Nonparametric Prediction Utility for Embedded Systems. *ACM Transactions on Embedded Systems*, 7(2), April 2008.

[14] K. Ha, G. Lewis, S. Simanta, and M. Satyanarayanan. Cloud Offload in Hostile Environments. Technical Report CMU-CS-11-146, CMU School of Computer Science, December 2011.

[15] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The Impact of Mobile Multimedia Applications on Data Center Consolidation. Technical Report CMU-CS-12-143, CMU School of Computer Science, October 2012.

[16] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *ACM SIGOPS Operating Systems Review*, 40(5):14–24, 2006.

[17] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[18] R. Miller, August 2012. `http://www.datacenterknowledge.com/archives/2012/08/13/aol-brings-micro-data-center-indoors-adds-wheels`.

[19] Myoonet. Unique Scalable Data Centers, December 2011. `http://www.myoonet.com/unique.html`.

[20] D. Narayanan and M. Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proceedings of the 1st International Conference on Mobile Systems Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.

[21] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, pages 276–287, Saint-Malo, France, October 1997.

[22] OpenCV. OpenCV Wiki. `http://opencv.willowgarage.com/wiki/`.

[23] W. Richter, M. Satyanarayanan, J. Harkes, and B. Gilbert. Near-Real-Time Inference of File-Level Mutations from Virtual Disk Writes. Technical Report CMU-CS-12-103, Carnegie Mellon University, School of Computer Science, February 2012.

[24] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving Portable Computer Battery Power through Remote Process Execution. *Mobile Computing and Communications Review*, 2(1):19–26, January 1998.

[25] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1996.

[26] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8, August 2001.

[27] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), Oct-Dec 2009.

[28] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D. R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D. J. Farber, M. A. Kozuch, C. J. Helfrich, P. Nath, and H. A. Lagar-Cavilla. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing*, 11(2), 2007.

[29] A. W. Services. Overview of Amazon Web Services, December 2010. `http://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf`.

[30] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible SPH. *ACM Transactions on Graphics*, 28(3), 2009.

[31] Sphinx-4. Sphinx-4: A Speech Recognizer Written Entirely in the Java Programming Language. `http://cmusphinx.sourceforge.net/sphinx4/`.

[32] S. Srinivasa, D. Ferguson, C. Helfrich, D. Berenson, A. Collet Romea, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and J. M. Vandeweghe. Herb: a home exploring robotic butler. *Autonomous Robots*, 28(1):5–20, January 2010.

[33] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, August 2011.

[34] G. Takacs, M. E. Choubassi, Y. Wu, and I. Kozintsev. 3D mobile augmented reality in urban scenes. In *Proceedings of IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011.

[35] C. Thompson. What is I.B.M.'s Watson? *New York Times Magazine*, June 2011. `http://www.nytimes.com/2010/06/20/magazine/20Computer-t.html`.

[36] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[37] Wikipedia. Lempel-Ziv-Markov chain algorithm, 2008. [Online; accessed 22-April-2008 at `http://en.wikipedia.org/w/index.php?title=Lempel-Ziv-Markov_chain_algorithm&oldid=206469040`].

[38] Wikipedia. List of 802.11ac Hardware, 2012. `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.