Adaptive VM Handoff Across Cloudlets

Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu,
Brandon Amos, Padmanabhan Pillai[†],
Mahadev Satyanarayanan

June 2015
CMU-CS-15-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Intel Labs

Abstract

Cloudlet offload is a valuable technique for ensuring low end-to-end latency of resource-intensive cloud processing for many emerging mobile applications. This paper examines the impact of user mobility on cloudlet offload, and shows that even modest user mobility can result in significant network degradation. We propose VM handoff as a technique for seamlessly transferring VM-encapsulated execution to a more optimal offload site as users move. Our approach can perform handoff in roughly a minute even over limited WANs by adaptively reducing data transferred. We present experimental results to validate our implementation and to demonstrate effectiveness of adaptation to changing network conditions and processing capacity.

This research was supported by the National Science Foundation (NSF) under grant number IIS-1065336. Additional support was provided by the Intel Corporation, Google, Vodafone, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and should not be attributed to their employers or funding sources.

Keywords: mobile computing, cloud computing, edge computing, cloudlet, virtual machines, mobility, virtual machine migration, network latency

1 Introduction

Seamless handoff is a core concept in cellular networks. Without any disruption to ongoing voice or data transmission, a user is able to freely move over a significant geographic area that is spanned by many base stations of limited individual coverage. In this paper, we describe an analogous mechanism for cloud offload. For reasons of low end-to-end latency, some emerging mobile applications require cloud offload infrastructure to be close to the mobile device. In the ideal case, it is just one wireless hop away. For example, the offload infrastructure could be located in a cellular base station or it could be LAN-connected to a set of Wi-Fi base stations. In keeping with recent usage [24], we refer to individual elements of this offload infrastructure as *cloudlets*. An industry initiative on Mobile-edge Computing (MEC) was recently created by the European Telecommunications Standards Institute (ETSI) [12] to standardize cloudlets and to create a new ecosystem and value chain.

Since virtual machine (VM) encapsulation is used for safety, isolation, resource allocation, and provisioning of multi-tenant cloudlets, we refer to our mechanism as *VM handoff*. As a user moves, his VM is seamlessly transferred from cloudlet to cloudlet in order to preserve low end-to-end latency. This mechanism resembles live migration of VMs in data centers, but differs in at least three important ways.

First, the two mechanisms are optimized for very different performance metrics. The figure of merit in VM handoff is total time to completion, since degraded end-to-end latency persists until the end of the operation. Live migration, on the other hand, aims at short duration of the very last step (called "down time"), during which the VM instance is suspended. The total time to completion is a secondary consideration. As we show in Section 5.1, this difference in optimization metric can result in an order of magnitude difference in total completion time. Second, the economics of cloudlet deployments force it to accept whatever network and computing resources exist across dispersed cloudlets. Unlike the data center assumptions of live migration, VM handoff cannot rely on the presence of a dedicated high-bandwidth network. Hence, our system needs to tolerate high variability in bandwidth and compute capacity due to workloads from other users even during the course of a single handoff. By dynamically adapting to these changing conditions, VM handoff can offer significant performance improvement as shown in Section 5.3. Third, VM handoff leverages the presence at the destination of a *base VM* from which its image was derived and uses this in combination with delta provisioning of VM states as originally proposed by Ha et al. [15].

VM handoff targets emerging mobile applications that require low end-to-end latency. For example, the report of the 2013 NSF Workshop on Future Directions in Wireless Networking [4] identifies a new genre of applications called *cognitive assistive applications* that it characterizes as "astonishingly transformative." These applications use cloud resources in the critical path of real-time user interaction. Consequently, they cannot tolerate end-to-end operation latencies of more than a few tens of milliseconds. Augmented reality applications that use head-tracked systems require end-to-end latencies of less than 16 ms [11]. Thin client access to cloud-based virtual desktop applications require an end-to-end latency of less than 60 ms to match the perceived quality of local execution [28]. Action games with remote rendering also require low latencies and high bandwidth [16].

The interactive response of such latency-sensitive applications will degrade as the logical network distance increases. As discussed in the next section, this degradation can be far worse than physical distance may suggest. VM handoff can mitigate this effect of user mobility while remain-

	5%	10%	50%	90%	95%
Home A & B	18.5	19.2	26.4	77.8	133.6
Home B & C	36.4	37.2	44.9	87.2	98.0
Home C & A	38.8	39.3	44.9	75.1	92.6

(a) Latency Distribution (milliseconds)

	5%	10%	50%	90%	95%
Home A & B	0.5	0.6	1.9	2.3	2.3
Home B & C	0.5	0.7	0.8	0.9	0.9
Home C & A	0.5	0.5	0.8	0.9	0.9

(b) Upload Bandwidth Distribution (Mbps)

These are the observed distributions of latency and upload bandwidth over a one-week period in November 2014 between three homes that are located within a one square mile area. All the homes have broadband Internet connectivity provided by ISP1 for A and B, and ISP2 for C.

Figure 1: Network Quality Between Homes

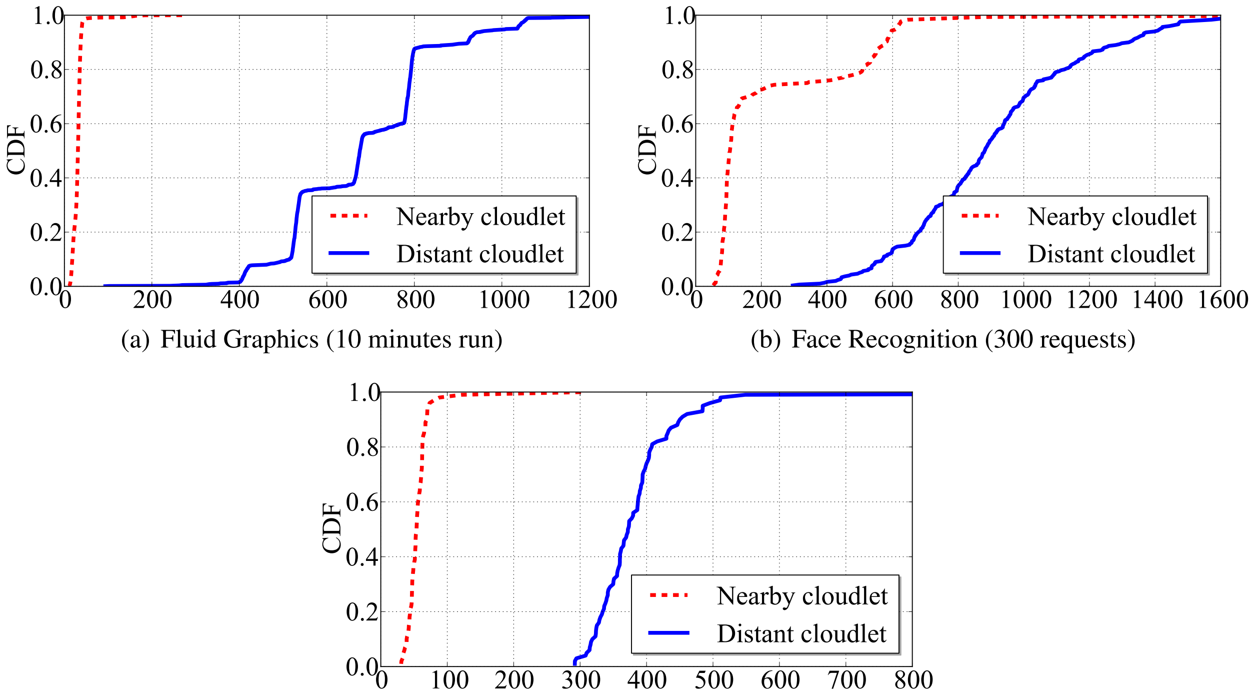
ing transparent to applications.

2 Why VM Handoff?

In practice, how important is VM handoff to user experience? Consider the scenario of a user visiting his neighbor who lives just down the street. The user is running a mobile application that is latency-sensitive and is using the services of a cloudlet in his home. When he reaches the second home, he associates with a Wi-Fi access point there but continues to use the original cloudlet. Because of two traversals of last-mile links connecting the homes to their ISPs, the user is likely to see significant degradation of latency and bandwidth to his cloudlet.

To validate this intuition, we measured network quality over a one-week period between the homes of three authors of this paper that are located within the same neighborhood in a city. As the results in Figure 1 show, end-to-end latency and bandwidth are poor in spite of physical proximity. For example, even though homes A and B connect to the same ISP, the median latency between them is 26.4 milliseconds. This is consistent with measurements reported by Sundaresan et al [26]. Homes that are connected to different ISPs can expect even worse network quality between them. For example, homes B and C are just one block apart, but the median latency between them is more than 40 milliseconds. Without VM handoff, even modest user mobility may result in unacceptable degradation of network quality to the associated cloudlet.

Degraded network quality translates into slower response times for latency-sensitive applications. To illustrate this, we evaluate the performance of three representative mobile applications. In our experiments, user interaction occurs on a mobile device while the compute-intensive processing of each interaction occurs in a back-end server encapsulated in a VM instance on a cloudlet. In the `FLUID` application, the mobile device sends sensor readings from a smartphone to a cloudlet; there, a compute-intensive particle simulation of a liquid with specific viscosity properties is performed [25], and a frame containing the liquid's state is generated and returned for rendering. In



(a) Fluid Graphics (10 minutes run)

(b) Face Recognition (300 requests)

(c) Augmented Reality (100 requests)

Figure 2: CDF of Response Times (milliseconds)

the `FACE` application, an image is shipped from the mobile device to the cloudlet; there, a face recognition algorithm is executed and a text string of the name of the person is returned. This algorithm uses a Haar Cascade of classifiers for face detection and the Eigenfaces method for face identification [29]. In the `MAR` application, an image is shipped from the mobile device to the cloudlet; there, a back-end server identifies buildings and landmarks, and returns a modified image with these annotations [27].

In Figure 2, the label "nearby cloudlet" corresponds to the case where the mobile device and cloudlet are both located at home C. In this configuration, the mobile device has one-hop Wi-Fi connectivity to its cloudlet. The label "distant cloudlet" corresponds to the case where the mobile device is at C, but its associated cloudlet is at A. In all cases, a nearby cloudlet yields significantly lower response times. For example, the median response time of `FACE` is 104 ms when it is associated to the nearby cloudlet, but it is 882 ms when using the distant cloudlet. Since this performance difference is solely due to differing network conditions, it confirms the importance of VM handoff.

3 Background and Related work

3.1 Live Migration for Data Centers

Live migration [8] is the de facto standard for VM migration in data centers. Its goal is to minimize "down time" during which a migrating VM instance is suspended. To achieve this goal, live migration allows the VM instance to continue execution at the source host while transferring mod-

VM	Total time	Down time	State transfer size
OBJECT	127 Min (0.03)	1.45 s (0.1)	8.42 GB (0.004)
MAR	159 Min (394)	7.44 s (0.3)	10.56 GB (0.43)

(a) No Base Image at Destination (`no-share`)

VM	Total time	Down time	State transfer size
OBJECT	12 Min (0.07)	1.54 s (0.3)	0.80 GB (0.004)
MAR	52 Min (20.4)	7.63 s (0.8)	3.45 GB (1.35)

(b) Using Base Image at Destination (`incremental`)

VM for both OBJECT and MAR are configured with 8GB disk and 1GB memory. The OBJECT guest operating system is Ubuntu Linux 12.04, while that of MAR is Windows 7. Average of 3 runs is reported with standard deviation in parentheses.

Figure 3: QEMU-KVM Live Migration on WAN

ified memory state in the background to the destination host. During the transfer, which may take many seconds to tens of seconds in typical usage, additional memory state may be modified by the executing VM instance. The entire process is repeated for multiple iterations (typically of shorter and shorter durations) until the very last step. In that final step, the VM instance is suspended at the source, all its remaining modified state is sent to the destination, and execution is resumed there. Through this convergent series of iterations, live migration minimizes down time to as little as hundreds of milliseconds on LAN. The entire migration process typically takes between tens of seconds to a few minutes.

As originally described, live migration does not transfer disk state; it only transfers memory state. This is acceptable in a data center environment because the source and destination hosts can be assumed to share disk storage through a mechanism such as a SAN (storage area network) or NAS (network-attached storage) device. A number of efforts [2, 6, 19, 32] have extended the basic live migration mechanism to work across data centers that are connected by a WAN. These efforts address the transfer of both memory state and disk state during migration, since shared-storage mechanisms do not work well over WANs.

3.2 Using Live Migration for VM Handoff

One may wonder whether VM live migration can be used with cloudlets because it is designed for seamless migration of computing states across machines. To verify feasibility, we conducted experiments using QEMU-KVM in Ubuntu Linux 12.04. This production-quality VMM implementation is widely used in OpenStack data centers for cloud computing [20]. As representative cloudlet workloads, we use `MAR,` the augmented reality application that was described in Section 2, and `OBJECT`, an object recognition application. The back-end server of `OBJECT` uses the MOPED algorithm [9] on an image sent by the mobile device, and returns the bounding box and identity of recognized objects. We performed live migration between two cloudlets that were con-

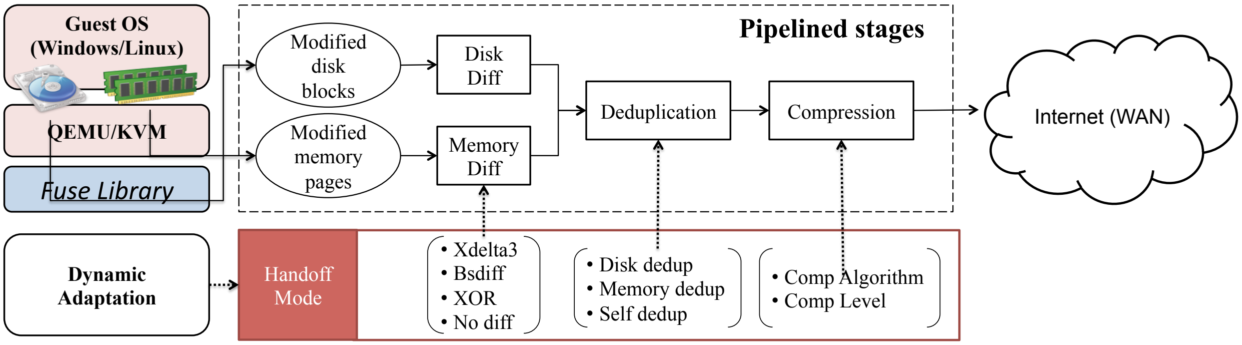


Figure 4: Overall System Diagram for VM Handoff

nected by a stable 10 Mbps, 50 ms RTT network provided by a Linktropy emulator on a gigabit Ethernet.

QEMU-KVM hypervisor provides a mechanism to migrate both memory states and disk states, called *live block migration*. In live block migration, one can either transfer entire disk state to the destination or only transmit incremental disk state assuming the source and destination share base disk states. Figure 3(a) presents the results for the expected worst case, which transfers the entire disk state to the destination. The results show a total migration time of over two hours for both images, but a down time of just a few seconds. Two hours is a very long time for a mobile user to suffer degraded network access to his cloudlet. The fact that down time is just a few seconds is little consolation. Most mobile users would gladly accept much longer down times if it resulted in a faster switch to a nearby cloudlet.

Figure 3(b) presents the results when the base VM images into which OBJECT and MAR were installed are available at the destination cloudlet. One would expect the total state transfer, and hence migration time, to decrease significantly. This is indeed true for OBJECT, which completes the entire migration in just 12 minutes. Unfortunately, MAR behaves quite differently. The total migration time still takes more than 50 minutes with high variation. It turns out that this unexpected behavior is due to the timing of background activity in the Windows 7 guest. This background activity generates modified memory state at rate high enough to inordinately prolong live migration. And it is reflected in the large total volume of state transferred. Yet, down time remains less than 10 seconds for MAR in Figure 3(b).

In principle, one could re-tune live migration parameters to eliminate this pathological behavior. However, this could hurt down time in situations where that metric matters. Optimizing for the right context-sensitive metric is a non-trivial problem. Classic live migration is a fine choice for data centers, but inappropriate in a cloudlet context. Other possibilities include applying the concept of partial VM migration [5], and leveraging content similarity from VM images distributed across multiple nodes [17, 21, 22, 23]. These approaches are complementary to VM handoff, which focuses on achieving near-ideal post-handoff performance without reliance on external infrastructure.

3.3 Dynamic VM Synthesis

Dynamic VM synthesis [24] leverages the fact that most VM images are derived from a small number of widely-used base VM images (typically Linux or Windows) that can pre-populate cloudlets. The desired application-specific VM image, referred to as the *launch VM,* is initially created through a relatively slow offline process that involves installation of application-specific software (including dynamically-linked libraries and tool chains) into a base VM. The binary difference between a launch VM and its base VM is the *VM overlay,* whose small relative size is the key to previous work on rapid provisioning [15]. At runtime, the launch VM is reconstituted by applying the overlay to its base VM.

Our implementation of VM handoff is inspired by VM synthesis. However, rather than overlay creation being a one-time offline operation, a series of overlays are generated afresh at runtime on the source cloudlet during the course of a single VM handoff. The time for overlay creation, which was ignored in earlier work because it was an offline operation, now becomes a significant limiting factor. In addition, the tuning parameters (such as compression algorithm) used in overlay creation are dynamically re-optimized at run-time in order to reflect the current relative costs of network transmission and cloudlet computation. Thus, although VM handoff borrows concepts from VM synthesis, it represents a substantial new mechanism in its own right.

4 Design and Implementation

Our design of VM handoff reflects the three considerations mentioned in Section 1: (a) optimizing for total handoff time rather than down time; (b) dynamically adapting to WAN bandwidth and cloudlet load; and (c) leveraging existing VM state at cloudlets. Figure 4 illustrates the overall design. A pipeline of processing stages is used to efficiently find and encode the differences between current VM state at the source, and already-present VM state at the destination. This delta encoding is then deduplicated and compressed (using parallelization wherever possible), and then transferred. The algorithms and parameters used in these stages are chosen to match current processing resources and network bandwidth. We describe these details below.

4.1 Tracking Changes

Our system is mostly implemented as modules separate from the hypervisor. This allows the system to be more flexible in controlling the resource usage for handoff, minimizes modifications to the hypervisor, and potentially allows support for different hypervisors, though our current implementation uses QEMU/KVM. On the other hand, it is more difficult to track VM disk and memory state changes from outside the hypervisor.

To efficiently track changes to VM disk state, our system uses the Linux FUSE interface to implement a user-level filesystem on which the VM disk image is stored. All of the running VM's disk accesses are passed through the FUSE layer, which can accurately track modified blocks with little performance impact to the running VM. At the start of handoff, our system can then immediately capture all VM disk blocks that differ from those in the corresponding standard base VM image. As in live migration, the VM can continue to run, and any further disk modifications will be tracked for subsequent iterations of transmission.

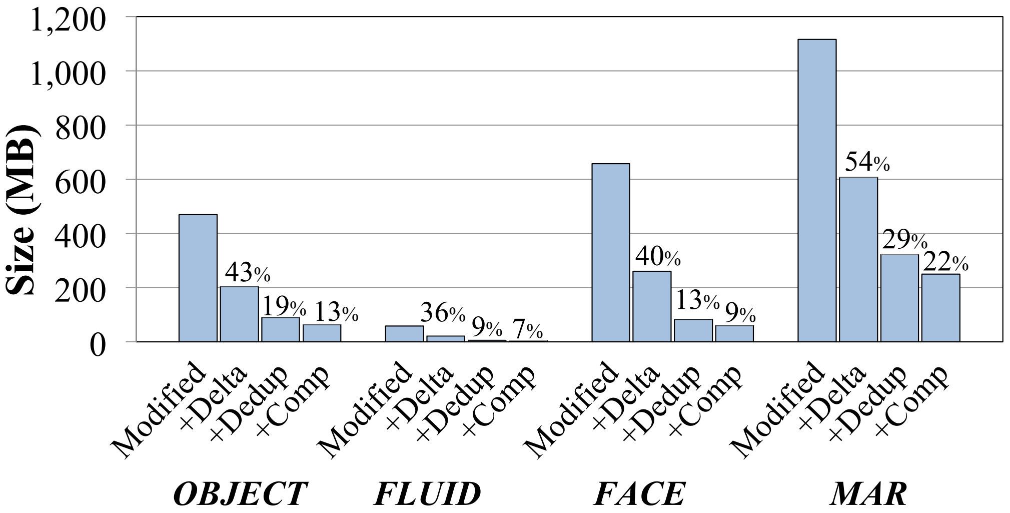


Figure 5: Cumulative reductions in size of VM state transferred. (`Xdelta3` for binary delta and `LZMA` level 9 for compression)

For tracking VM memory modifications, a FUSE-like approach would incur too much overhead on every memory write. Instead, we capture the memory snapshot at handoff, and determine the changed blocks in our code. To get the memory state, we rely on QEMU/KVM's live migration mechanism. We use a Unix socket to send a command to QEMU/KVM to start migrating state. This will cause it to mark all VM pages as read-only to trap and track any further modifications, and to start a complete transfer of the memory state. Rather than sending this state over the network, our system redirects this through a pipe to our processing stages that filter out unmodified pages, and heavily compress the remaining data before transmission using various techniques. As this capture of memory snapshot is based on a mechanism intended for live migration, QEMU/KVM will then iterate this process, sending pages that were modified over the duration of the previous iteration of modified pages. To limit repeated transmission of a set of rapidly changing pages, our system can regulate the start of these iterations, limit how many iterations are performed, or can eliminate them completely by pausing the VM before starting the memory snapshot.

4.2 Reducing Data Size

As the network bandwidth is often the bottleneck in VM handoff, our system tries to aggressively reduce the data volume transmitted across the network. We implement a pipeline of processing stages to delta-encode, deduplicate, and compress data before it hits the network.

We study the effectiveness of these processing stages in reducing data size on four applications that are representative of cloudlet workloads. The behavior of these applications, `OBJECT`, `FACE`, `MAR`, and `FLUID` are explained in Figure 2 and 3. Figure 5 shows that the system can significantly reduce the volume of data transferred to between 1/5 and 1/10 of the total modified data blocks. However, the processing costs of these operations may result in CPU, rather than network transfer, becoming the bottleneck. To avoid this, our system can use different algorithms and settings to balance the processing requirements and data size reductions. Details of these options are discussed below.

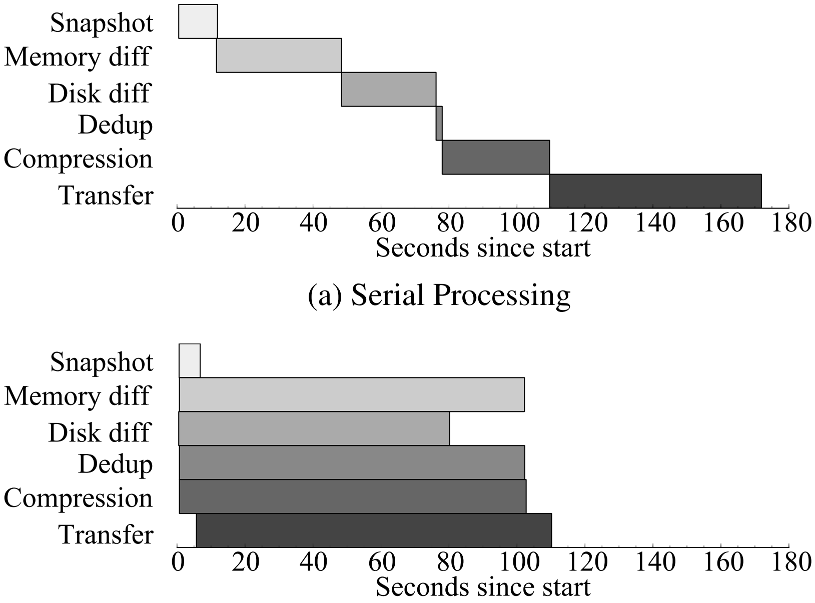
Delta encoding of modified pages and blocks: The streams of modified disk blocks and all VM memory pages are fed to two delta encoding stages (*Disk diff* and *Memory diff* stages in Figure 4). The data streams are split into 4KB (page/block size) chunks, and are compared to the corresponding chunks in the base VM. We use a SHA-256 hash [14] to make these comparisons. The hash values are preserved for later use. Chunks that are identical to those in the base VM are omitted. For each modified chunk, we use binary delta algorithm to encode the difference between the chunk and the corresponding one in the base VM image, and only transmit the delta if it is smaller in size than the chunk. The idea here is that small or partial modifications are common and there may be significant overlap between the modified and original block when viewed at finer granularities. Our system can be dynamically configured to use either `xdelta3`, `bsdiff4`, or `xor` to perform the binary delta encoding, or to simply pass through modified chunks without delta encoding. As both the hash computations and the delta encoding steps are compute intensive, we parallelize these on multiple processing threads.

Deduplication: The streams of modified disk and memory chunks, along with the computed hash values, are merged and passed to the deduplication stage. Deduplication has been widely used and is very effective in reducing redundant data in a variety of contexts. Here, deduplication is particularly effective, as multiple copies of the same data commonly occur in a running VM. For example, multiple copies of the same data may reside in kernel and user-level buffers, or on disk and OS page caches. For each modified chunk, we compare the hash value to those of (1) all base VM disk chunks, (2) all base VM memory chunks, (3) a zero-filled chunk, (4) all prior chunks seen by this stage. The last is important to capture multiple copies of new data in the system, in either disk or memory. This stage filters out the duplicates that are found, replacing them with pointers to identical chunks in the base VM image, or that were previously emitted. As the SHA-256 hash used for matching was already computed in the previous stage, deduplication primarily reduces to fast hash lookups operations, so this stage can be easily run as a single thread.

Compression: Compression is a final stage of processing before the VM modification data is sent to the network. In this stage, we attempt to squeeze the data further by applying one of several off-the-shelf compression algorithms, including `GZIP` (deflate algorithm) [10], `BZIP2` [7], and `LZMA` [31]. These algorithms vary in the data compression achieved and processing speed. GZIP provides relatively modest compression ratios, but uses very little processing time. LZMA is optimized for high compression ratios and fast decompression, but at the price of slow compression. BZIP2 falls in the middle in terms of compression rate and processing requirements. As these compression algorithms (particularly LZMA) work best on large blocks of data, this stage aggregates the modified chunk stream into approximately 1 MB segments before applying compression. Finally, as this is a CPU intensive stage, we run multiple instances of the compression algorithms in separate execution threads, sending segments to the threads in round-robin fashion. This lets us take advantage of all available cores, and parallelizes well, without requiring multi-threaded implementations of the compression algorithms.

4.3 Pipelined Execution

Our system pipelines the execution of these processing stages, so all of them are active simultaneously, and data is streamed through the various processing steps. This has two main advantages



(a) Serial Processing

(b) Pipelined Processing

Memory/Disk diff stage use `xdelta3` algorithm and compression stages uses LZMA compression level 5.

Figure 6: Serial versus Pipelined Processing

over a serialized implementation, where all data is processed through a particular stage before starting the next stage. First, it allows downstream stages to start before the preceding ones complete. In particular, we can start transferring data on the network quickly, in parallel to the processing stages. Increasing network bandwidth utilization is critical for VM handoff between cloudlets, where network bandwidth is a scarce resource that should not be wasted or left unsused. Pipelining helps ensure migration data begins to reach the network as quickly as possible. Secondly, it requires less memory to buffer the intermediate data generated by the individual stages, as they are consumed quickly by downstream ones. Note that the total processing time is not significantly affected by the pipelining. This is because the total amount of computation is roughly the same, and even in the serialized case, our multi-threaded stages can make good use of multiple processing cores.

The example measurements in Figure 6 illustrate the benefit of a pipelined implementation. In the serial case, each stage has to be completed before the next stage can begin. In the pipelined case, all stages are available to begin processing data very soon after initialization. Figure 6 shows that pipelining produces a roughly 5% reduction in total processing time. (Serial processing takes 109.6 s and pipelined processing takes 102.7 s from its start to finishing compression). However, total handoff time is reduced by 36% (171.9 s → 110.2 s) because network transfers are overlapped with execution of the processing stages.

We also measure how well our pipelined system scales as the available CPU resources increase at the cloudlet. Figure 7 shows VM handoff time for different workloads with differing number of CPU cores. Except for `FLUID`, the processing throughput increases as we use more cores. Much of the gain comes from parallel processing of the input data, since we use multiple execution threads

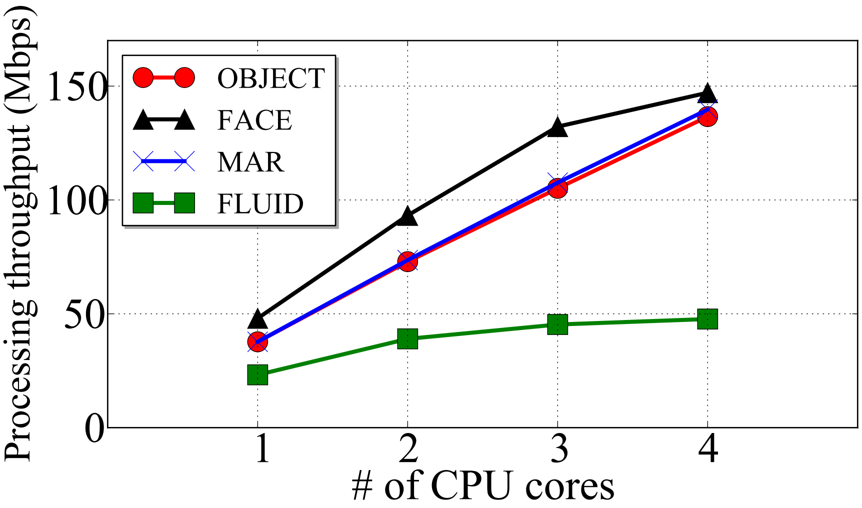


Figure 7: Processing Scalability of the System

for the stages that are CPU-intensive such as binary delta and compression. For `FLUID`, the total volume of data processed is too small to significantly benefit from multiple cores.

4.4 Dynamic Adaptation

In the example handoff performance data from Figure 6, due to the particular choices of algorithms and parameters used, the system is clearly processing bound. The processing takes 109.6 s, and dominates the total handoff time, while network transfer only requires 62.3 s. Here, it would have been better to select compression algorithms to reduce the processing requirements, even at the expense of data transfer size in order to reduce the overall handoff time.

If we knew exactly how much compression could be achieved and exactly how long this would take to transfer the modified VM state for all algorithms, and had guarantees on the available bandwidth, we could find a static configuration of processing stages to optimize the handoff time. However, we cannot know all of these in advance, as this is highly dependent on the actual data that needs to be transferred. Furthermore, network bandwidth can fluctuate significantly over time, as can available processing resources. Thus, selecting the best processing parameters *a priori* is not practical, and in any case, a static configuration may not remain the best choice as conditions change over the duration of handoff. Furthermore, the best static configuration for one workload might not work well for the other workloads because processing time and compression ratio vary depending on the workloads. Instead, our system employs continuous monitoring of handoff performance, and uses this information to dynamically adapt the processing stage settings to reduce handoff time, as detailed below. In the rest of this paper, we refer to a set of selected algorithms and parameters for the processing stages as an *operating mode*.

4.4.1 Pipeline Performance Model

In order to develop an algorithm to select the right operating mode, we first develop a simplified model of our processing pipeline. The goal of this modeling is to calculate the system throughput by identifying the bottlenecks. Our pipelined system has two potential bottlenecks: processing and

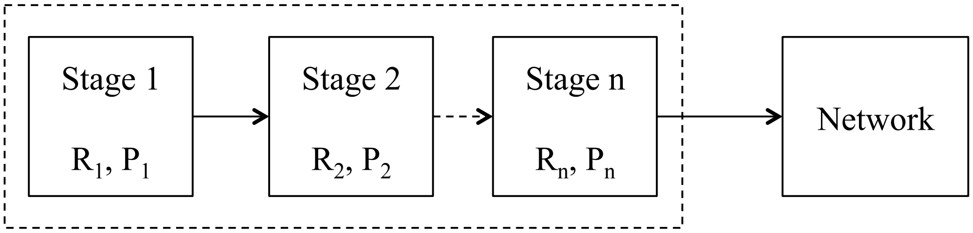


Figure 8: Pipeline modeling

network transmission. On one hand, the handoff operation will be bottlenecked by the processing speed if the system is configured to aggressively reduce migration data size using processing-intensive algorithms and settings. On the other hand, the speed of handoff may be limited by the available network bandwidth if the pipeline does less processing in order to more fully utilize network bandwidth. In this modeling, we characterize system throughput with respect to these potential bottlenecks.

Figure 8 shows a simplified model of our pipeline. The processing is a sequence of stages 1 – n, each of which takes input data and outputs a transformed, smaller version of the data. For each stage i, we define:

$$P_i = \text{processing time},\ R_i = \frac{\text{output size}}{\text{input size}} \quad \text{at stage i}$$

These values are defined for a particular operating mode (set of selected algorithms and parameters for the processing stages). From these, we can compute the processing time as:

$$Time_{processing} = \sum_{1\leq i\leq n} P_i \tag{1}$$

The time to transfer on the network is computed as:

$$Time_{network} = \frac{Size_{migration}}{\text{Network bandwidth}} \tag{2}$$

(where $Size_{migration} = Size_{\text{modified\_VM}} \times (R_1 \times \cdots \times R_n)$)

Since our pipeline overlaps processing and network transmission, from (1) and (2), the total total handoff time is:

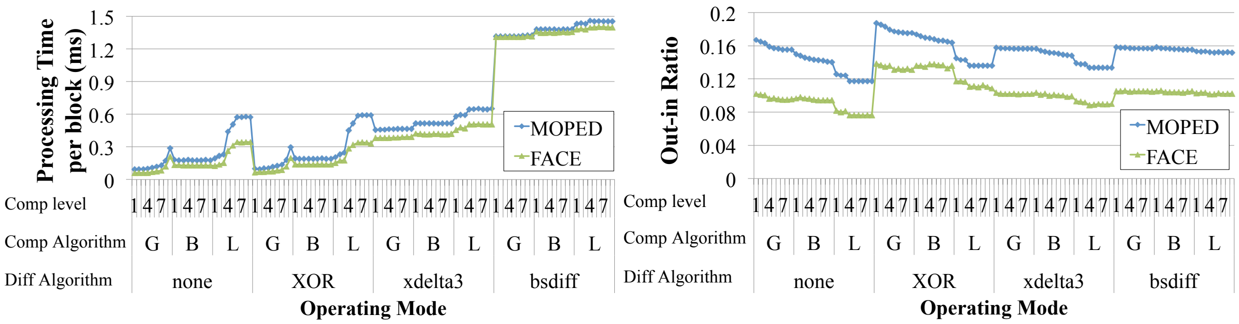
$$Time_{handoff} = max(Time_{processing}, Time_{network}) \tag{3}$$

In the implementation, we use processing throughput and network transmission throughput instead of processing time and network transfer time, because calculating total transmission time requires undetermined information such as total input size. Therefore, the total system throughput is

$$Thru_{system} = min(Thru_{processing}, Thru_{network}) \tag{4}$$

where,

$$\begin{aligned} Thru_{processing} &= \frac{1}{\sum_{1\leq i\leq n} P_i} \\ Thru_{network} &= \frac{\text{Network Bandwidth}}{(R_1 \times \cdots \times R_n)} \end{aligned} \tag{5}$$



(a) Processing time (P) in different operating modes

(b) Out-in ratio (R) in different operating modes

Figure 9: Trends of P and R across workloads (G: `Gzip`, B: `Bzip2`, L: `LZMA`)

Intuitively, (5) shows our pipelined system is bottlenecked by either processing speed or network transmission speed. It also indicates that we can calculate the system throughput if we measure processing time (P) and out-in ratio (R).

4.4.2 Adaptation Heuristic

Applying this model to our system, we develop a heuristic to dynamically adapt the operating mode. The goal of the heuristic is to select the operating mode that maximizes the system throughput $Thru_{system}$. We profile the values of P and R for various workloads and operating settings. However, as discussed earlier, the actual computational demands and compression ratios achieved vary greatly depending on the actual content of the modified VM data. Hence, using the profiled P and R values directly may be highly misleading. Figure 9 illustrates this with measured P and R values for varying operating modes of two different workloads, `FACE` and `OBJECT`. Each data point in the figure is P or R per block (i.e. modified memory page or disk blocks) of a particular mode. As expected, the P and R values differ significantly for the two workloads even with the same compression settings. However, the ratio of P (or R) values between different operating modes are relatively stable between the different workloads. In other words, the trends of P and R in varying operating modes are similar across different workloads. The intuition behind this is that although one workload may be much harder than another, it impacts the various compression algorithms to a similar degree, and the relative performance remains roughly similar. We confirm this on the 4 very different workloads used here: the absolute values of P and R of each workload are different, but their trends are similar. The robustness of this similarity across workloads from different developers and which span different operating systems, libraries, and databases suggests that it is a stable metric upon which base adaptation. Our heuristic uses ratios of P (or R) from the profiled data, rather than the absolute values, to determine which operating modes will likely minimize handoff time. Each iteration of the heuristic proceeds as follows:

1. First, measure the current P ($P_{current}$) and R ($R_{current}$) values of the running workload with the current operating mode ($M_{current}$). The system also measures the current network bandwidth by observing the rate of data segment acknowledgments received from the handoff destination.

2. From the profile, find the profiled value P ($P_{profile}$) and R ($R_{profile}$) of the matching operating

Mode, M. Then, compute the scaling factor for P and R.

$$Scale_P = \frac{P_{current}}{P_{profile}}, Scale_R = \frac{R_{current}}{R_{profile}}$$

3. Apply these scaling factors to "adjust" the profiled values for the current workload. Then, our heuristic calculates processing throughput ($Thru_{processing}$) and network transmission throughput ($Thru_{network}$) using (5), for each operating mode.

4. Finally, select an operating mode that maximizes the system throughput according to (4).

This heuristic can react to changes in the networking bandwidth, available processing resources (due to other loads on the cloudlet), or to changes in the compressibility of the VM modifications. In practice, the P values are actually measured in terms of processing time per input data block. The total possible combinations of settings (number of operating modes) is fairly small, so we can exhaustively enumerate them with little processing effort. The adaptation loop is repeated every 100 ms to let the system react quickly to any changes, or quickly discover any mispredictions. An updated operating mode will last for at least 5 s to provide hysteresis and give the system enough time to let effects of the change propagate throughout the pipeline, and reflect in stable measurements of P and R values.

4.5 Workload Distribution

The actual load placed on the processing pipeline and network are directly related to the number of modified pages and blocks that are transferred. For disk blocks, our change tracking mechanisms ensure only the modified disk blocks are delivered to the processing pipeline. For the memory image, however, the entire snapshot, including both modified and unmodified pages are processed. The relative loads on the network and processor, as well as our P and R values, vary depending on the ratio of modified and unmodified pages arriving at the pipeline.

Unfortunately, operating systems often manage memory space such that allocations, and therefore modifications, tend to be clustered. Figure 10 illustrates the set of modified pages for a VM by physical address. Clearly, the modifications are non-uniform, and highly clustered. Sending this memory snapshot to our processing pipeline would result in a highly bursty workload. This is problematic for two reasons. First, long sequences of unmodified pages could drain later pipeline stages of useful work and may idle the network, wasting this precious resource. Long strings of modified pages could result in high processing loads, requiring lower compression rates to keep the network fully utilized. Both of these are detrimental to our goal of minimizing transfer time.

To address this problem and have a balanced workload during the process of VM handoff, we use a technique called *workload distribution*. Workload distribution randomizes the order of page frames passed by the hypervisor to our processing pipeline. This way, even if the memory snapshot has a long series of unmodified pages at the beginning of physical memory, all of our pipeline stages will quickly receive work to do, and neither processing nor network resources are left idling for long. More importantly, the ratio of modified and unmodified pages arriving at the processing pipeline remains stable compared to when we simply pass pages in the sequential, address order. Figure 11 shows how workload distribution moderates the processing time per block, eliminating both spikes and troughs. The spikes correspond to CPU-bound conditions, causing

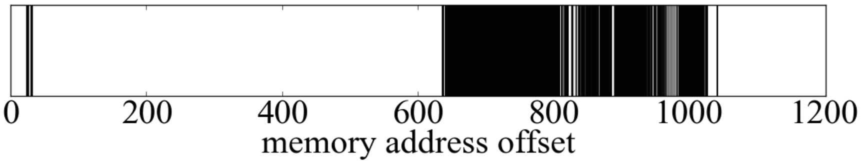


Figure 10: Modified Memory Regions (Black)

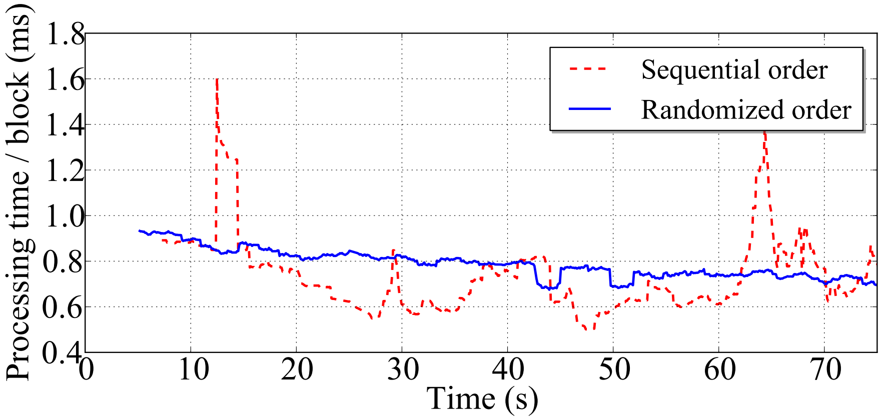


Figure 11: Randomized versus Sequential memory order (1-second moving average)

underutilization of network, while the troughs result in underutilization of CPU resources due to network bottlenecks. Workload distribution avoids the extremes and helps our system efficiently use both resources. Note that in this figure, no adaptation is performed (i.e., static mode is used), so the overall average processing times are the same for both plots. Furthermore, no network transfer is actually performed, so effects of network bottlenecks are not shown here.

4.6 Iterative Transfer for Liveness

VM handoff has to strike a delicate balance between brief service disruption and extended service degradation. If total handoff time were the sole metric of interest, the approach of suspending, transferring, and then resuming the VM would be optimal. Unfortunately, even with all our optimizations, this is likely to disrupt offload service too long (many minutes on a slow WAN) for good user experience. At the other extreme, if reducing the duration of service disruption were the sole metric of interest, classic live migration would be optimal. However, as shown earlier, this may extend degraded service unacceptably. The reality of mobile user experience is that neither extreme is acceptable.

Our solution to this problem is to borrow the concept of iterative transfer from live migration, but to embed it in the very different context of adaptive VM state transfer. As in live migration, the VM instance continues to run and accrue further changes which are transferred in subsequent iterations. However, unlike live migration, which focuses solely on volume of data transfer to drive the process, VM handoff is sensitive to multiple factors: data volume, processing speed, compression ratio achieved, and current bandwidth. Our system uses an input queue threshold to trigger the next iteration and uses the duration of an iteration to capture all factors affecting the speed of migration. If the duration of the iteration is sufficiently short, then our system suspends the VM and completes the handoff operation. We have empirically set the input queue threshold to 10 MB, and the interval to 2 seconds.

5 Evaluation

In this section, we evaluate the performance of our system for different workloads under various network and computing conditions. Specifically, we investigate our VM handoff system by answering the following questions:

- Is our system able to provide short VM handoff time on a slow WAN? How does it perform compared to classic live migration? (Section 5.1)
- How effectively does our system select operating modes at a given network bandwidth and computing resources? How well does this adaptation compare to the static modes? (Section 5.2)
- Is the implementation complexity of dynamic adaptation necessary? How dynamically does our system adapt its pipeline to varying conditions? (Section 5.3)

In our experiments, we emulate WAN-like conditions using the Linux Traffic Control (`tc` [18] tool), on physical machines that are connected by gigabit Ethernet. We configure bandwidth in a range from 5 Mbps to 25 Mbps, according to the average bandwidths observed over the Internet [1, 30], and use a fixed latency of 50 ms. To control computing resource availability, we use *CPU affinity masks* to assign a fixed number of CPU cores to our system. Our cloudlet machines (both handoff source and destination) each have an Intel® Core™ i7-3770 processor (3.4 GHz, 4 cores, 8 threads) and 32 GB main memory. To measure VM down time, we synchronize time between the source and destination machines using NTP. For difference encoding, our system selects from `xdelta3`, `bsdiff`, `xor`, or null. For compression, it uses the `gzip`, `bzip2`, or `LZMA` algorithms at compression levels (1–9). We use `OBJECT`, `FACE`, `FLUID`, and `MAR`, described earlier, as VM handoff workloads. For each workload, a back-end server is running and ready to serve mobile clients.

5.1 Overall Performance

Figure 12 presents the overall performance of VM handoff over a range of network bandwidths. The precise network and cloudlet conditions that trigger VM handoff are application-specific, and may also involve user preferences. Our focus here is on what happens after handoff is started. *Handoff time* is the total duration from the start of VM handoff until the VM resumes on the destination cloudlet. A user may see degraded application performance during this period. *Down time,* which is included in handoff time, is the duration for which the VM is suspended. Our results show that even at 5 Mbps, handoff time is just a few minutes and down time is just a few tens of seconds for all workloads. These are consistent with user expectations under such challenging conditions. As WAN bandwidth improves, handoff time and down time both shrink. At 15 Mbps using two cores, VM handoff completes within one minute for all of the workloads except `MAR`, which is an outlier in terms of size of modified memory state (over 1 GB, see Figure 5). The other outlier is `FLUID`, where the modified state is so small that there is not enough time for our

		1 CPU core		2 CPU cores	
BW (Mbps)		Handoff Time(s)	Down Time(s)	Handoff Time(s)	Down Time(s)
OBJECT	5	113.9	15.8 (6 %)	111.6	17.2 (7 %)
	10	66.9	7.3 (42 %)	58.6	5.5 (5 %)
	15	52.8	5.3 (12 %)	43.6	5.5 (31 %)
	20	49.1	6.9 (12 %)	34.1	2.1 (22 %)
	25	45.0	7.1 (30 %)	30.2	2.1 (26 %)
FLUID	5	25.1	4.0 (5 %)	17.3	4.1 (6 %)
	10	24.6	3.2 (29 %)	15.7	2.5 (4 %)
	15	23.9	2.9 (38 %)	15.6	2.2 (14 %)
	20	23.9	3.0 (38 %)	15.4	2.0 (19 %)
	25	24.0	2.9 (43 %)	15.2	1.9 (20 %)
MAR	5	494.4	24.0 (4 %)	493.4	24.5 (10 %)
	10	257.9	13.7 (25 %)	250.8	12.6 (13 %)
	15	178.2	8.8 (19 %)	170.4	9.0 (17 %)
	20	142.1	7.1 (24 %)	132.3	7.3 (20 %)
	25	121.4	7.8 (22 %)	109.8	6.5 (22 %)
FACE	5	247.0	24.3 (3 %)	245.5	26.5 (7 %)
	10	87.4	15.1 (10 %)	77.4	14.7 (24 %)
	15	60.3	11.4 (8 %)	48.5	6.7 (15 %)
	20	46.9	7.0 (14 %)	36.1	3.6 (12 %)
	25	39.3	5.7 (25 %)	31.3	4.1 (17 %)

Average of 5 runs and relative standard deviations (RSDs, in parentheses) are reported. For handoff times, the RSDs are always smaller than 9 %, generally under 5 %, and omitted for space. For down time, the deviations are relatively high, as this can be affected by workload at the suspending machine.

Figure 12: Overall System Performance

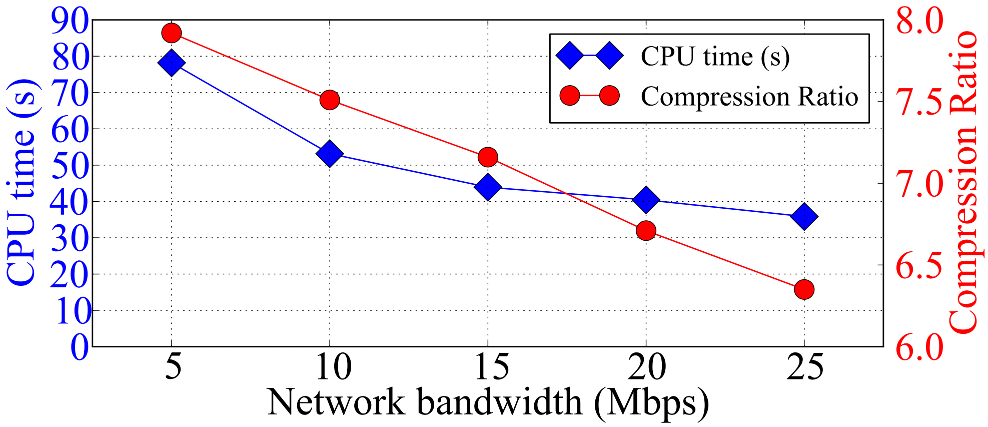


Figure 13: Performance detail of OBJECT using 1 CPU core and varying network

VM	Approach	Total time	Down time	Transfer size
OBJECT	Handoff	1 Min	5.5 s	0.06 GB
	KVM(no-share)	127 Min	1.45 s	8.42 GB
	KVM(incremental)	12 Min	1.54 s	0.80 GB
MAR	Handoff	4.2 Min	12.6s	0.27 GB
	KVM(no-share)	159 Min	7.44 s	10.56 GB
	KVM(incremental)	52 Min	7.63 s	3.45 GB

Figure 14: Comparison with QEMU-KVM live migration at 10 Mbps BW, 2 CPU cores

adaptation mechanism to adjust behavior. Consequently, the numbers at different bandwidths are very similar.

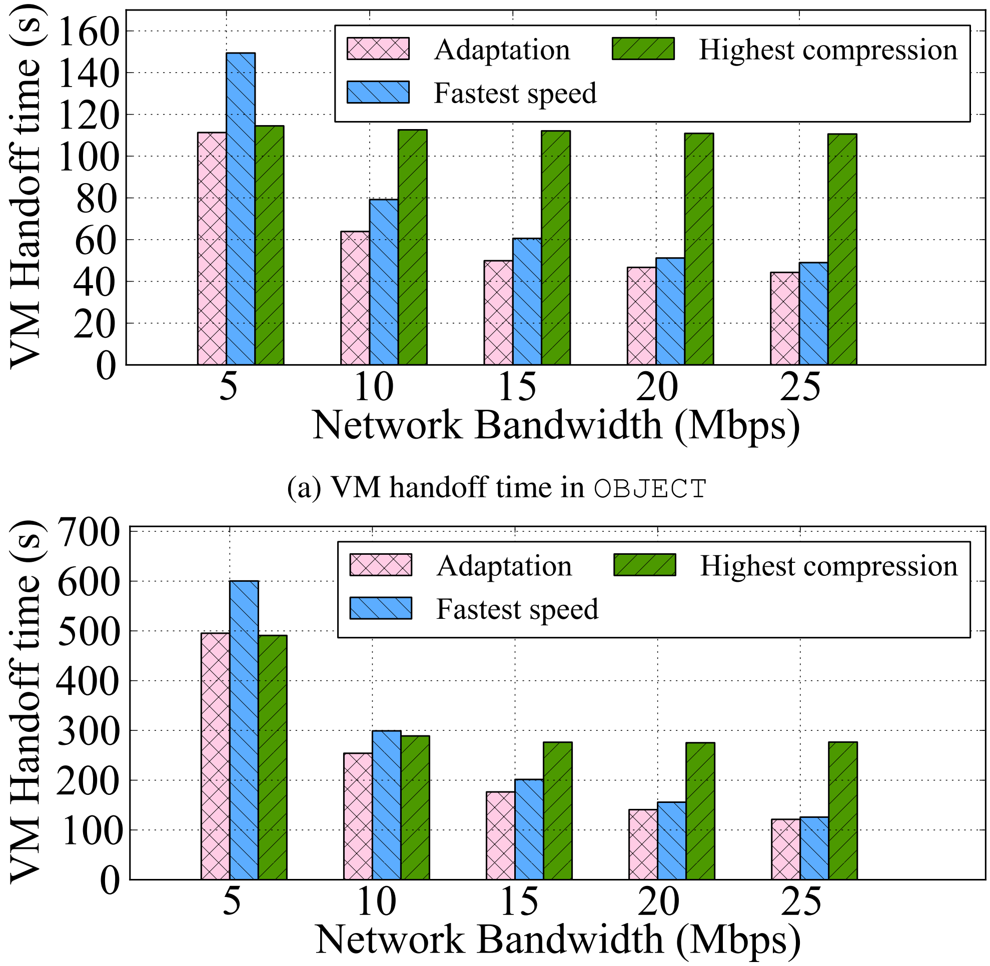
Figure 13 provides details of the processing time and compression ratio achieved under varied bandwidth for the OBJECT workload using a single core. "CPU time" is the absolute CPU usage, in seconds, by the VM handoff process. "Compression ratio" is the ratio of the input data size (i.e., modified VM state) to the output data size (i.e., final data shipped over the network). A higher compression rate indicates a smaller output size, typically achieved with more CPU usage. As the figure shows, our adaptation mechanism uses more CPU cycles to aggressively compress VM state when bandwidth is low, thus reducing the volume of data transmitted. At higher bandwidth, our system selects an operating mode consuming fewer CPU cycles, to avoid making processing a bottleneck. The average CPU usage remains high (between 80% and 90%) across all of the cases, indicating that the mechanism successfully balances computing speed and network transfer speed while using all available resources. We discuss this further in Sections 5.2 and 5.3.

Figure 14 contrasts the performance of VM handoff and QEMU-KVM live migration over a 10 Mbps network. It shows two migration modes supported by off-the-self KVM live migration: no-share and incremental. The no-share mode migrates the entire memory and disk state, while the incremental mode transfers the entire memory and only modified disk state, assuming that destination has the same base disk image used on the source. VM handoff clearly outperforms KVM's no-share migration. Even with the incremental migration mode (which has a conceptually similar assumption to our approach), VM handoff improves total migration time by an order of magnitude. This is due to the aggressive use of deduplication and compression while maintaining balance between processing rate and network transfer rate.

5.2 Operating Mode Selection

5.2.1 Comparison with Static Modes

Our VM handoff approach uses dynamic adaptation to select an operating mode for the given network conditions, processing resources, and workload. How well does this approach compare to picking a static configuration? To evaluate this, we first compare against two distinctive modes in the spectrum of the operating modes: highest compression and fastest speed. For fastest speed, each stage is tuned to use the least processing resources to achieve fast processing of migration data. In highest compression, we exhaustively run all the possible combinations and choose the option that minimizes the data transfer size. Note that the most CPU-intensive mode might not be the highest compression mode; some configurations can incur high



(a) VM handoff time in `OBJECT`

(b) VM handoff time in `MAR`

Figure 15: Adaptation versus Static modes in varying network conditions using 1 CPU core

processing costs, yet fail to achieve high compression rates.

Figure 15 compares VM handoff time of the two static modes and adaptation using two workloads, `OBJECT` and `MAR`. As expected, `fastest speed` performs best with high bandwidth, but works poorly with limited bandwidth. Except for the highest bandwidth tests, it is network-bound, so performance scales linearly with bandwidth. In contrast, `highest compression` minimizes the handoff time when bandwidth is low, but is worse than the other approaches at higher bandwidth. This is because its speed becomes limited by computation, and bandwidth is not fully utilized. It is largely unaffected by bandwidth change except in the very lowest bandwidth setting where network becomes a bottleneck. Unlike the two static cases that perform well only in certain bandwidth ranges, adaptation always yields good performance. In the extreme cases such as 5 Mbps and 25 Mbps, where the static modes have their best performance, the adaptation is as good as these modes. In the other conditions, it outperforms the static modes.

Figure 16 shows the handoff time for adaptation and the two static modes for differing numbers of CPU cores. This is for the `OBJECT` workload, with bandwidth fixed at 10 Mbps. `Fastest speed` shows constant handoff time regardless of available computing power, because it is not

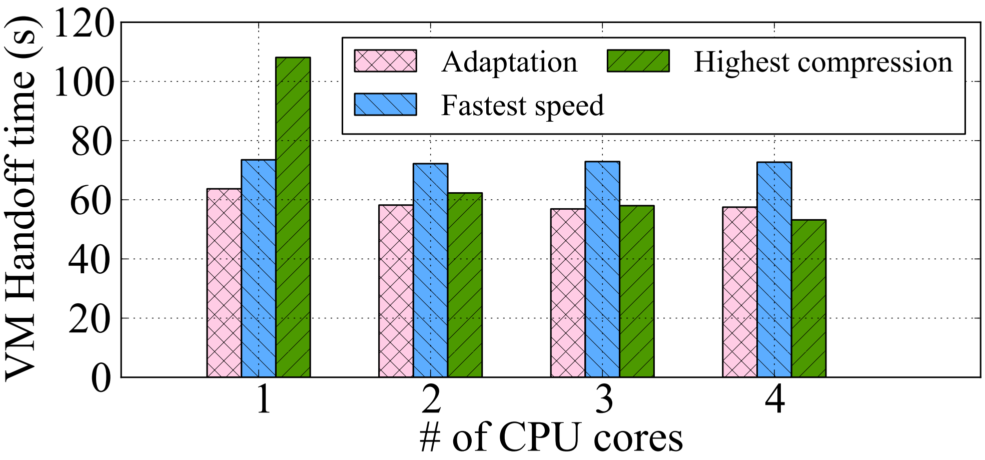


Figure 16: Adaptation versus Static modes as number of cores is varied (`OBJECT`, 10 Mbps)

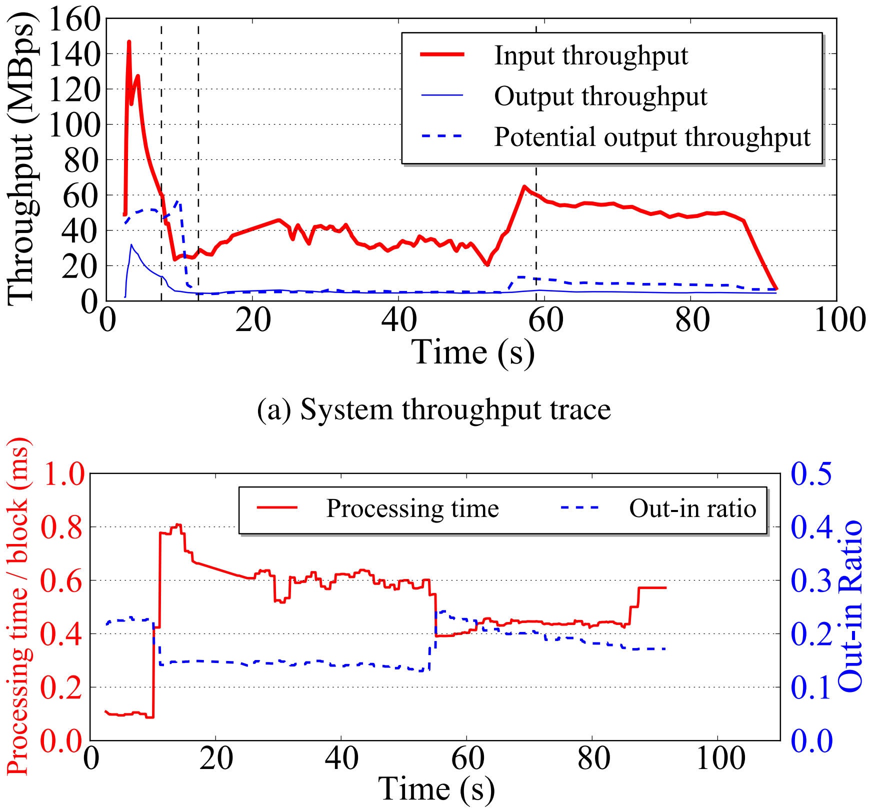
BW	Approach	Handoff time	Down time
5	Adaptation	113.9 s (3 %)	15.8 s (6 %)
	Best static	111.5 s (1 %)	15.9 s (12 %)
	Top 10%	128.3 s (2 %)	20.7 s (9 %)
10	Adaptation	66.9 s (6 %)	7.3 s (42 %)
	Best static	62.0 s (1 %)	5.0 s (11 %)
	Top 10%	72.1 s (1 %)	4.8 s (3 %)
20	Adaptation	49.1 s (8 %)	6.9 s (12 %)
	Best static	45.5 s (3 %)	8.1 s (15 %)
	Top 10%	48.5 s (1 %)	4.9 s (11 %)
30	Adaptation	37.0 s (4 %)	2.6 s (47 %)
	Best static	34.3 s (2 %)	2.1 s (8 %)
	Top 10%	48.5 s (1 %)	4.8 s (3 %)

Figure 17: Performance comparison between adaptation and static modes (`OBJECT`, 1 core, Relative standard deviations are reported in parentheses.)

limited by processing, but by network transfer. `Highest compression` improves as we assign more CPU cores. Again, adaptation is better than or similar to the best performance of the static operating modes in all of the conditions.

5.2.2 Exhaustive Evaluation of Static Modes

We have shown that adaptation performs better than two distinctive static modes. Note that it is not trivial to determine *a priori* whether either of these static modes, or one from the many different possible modes, would work well for a particular combination of workload, bandwidth, and processing resources. For example, our adaptation heuristic selects 15 different operating modes for the `OBJECT` workload as bandwidth is varied between 5 Mbps and 30 Mbps. Furthermore, the selections of the best static operating mode at particular resource levels is unlikely to be applicable to other workloads, as the processing speed and compression ratios are likely to be very different.



(a) System throughput trace

(b) P and R trace

Figure 18: Adaptation trace using 5 Mbps and 1 CPU core (OBJECT)

In spite of this, suppose we could somehow find the best operating mode for the workload and resource conditions. How well does our adaptation mechanism compare to this optimal static operating mode? To answer this question, we exhaustively measure the VM handoff times for all possible operating modes. Figure 17 compares the best static operating mode with adaptation for the OBJECT workload at varying network bandwidth. The 10th percentile performance among the static modes for each condition is also shown. The adaptation results are nearly as good as the best static mode for each case. Specifically, the adaptation results always rank within the top 5 among the 108 possible operating modes in most of the cases (ranked top 17th at 20 Mbps network bandwidth). Therefore, our adaptation mechanism manages to select good operating modes across a wide range of conditions, with little loss compared to the best static operating mode for each condition.

5.3 Dynamics of Adaptation

Our VM handoff system uses dynamic adaptation to both select an ideal operating mode for a static set of of resources, and also to adjust the modes as conditions change. To evaluate how well this process works, we study traces of execution under both static conditions and varying conditions.

5.3.1 Adapting to Available Resources

We first demonstrate how well our system adapts to the available resources and network bandwidth. Figure 18-(a) is an execution trace of our system, showing various throughputs achieved at different points in the system: output throughput, potential output throughput, and input throughput. Output throughput is the actual rate of data output generated by the processing pipeline to the network (solid blue line). Ideally, this line should stay right at the available bandwidth level, which indicates that the system is fully utilizing the network resource. If the rate stays above the available bandwidth level for a long time, the output queues will fill up and the processing stages will stall. If it drops below that level, the system is processing bound and cannot fully utilize the network. In the figure, we see that the output rate closely tracks the available network bandwidth.

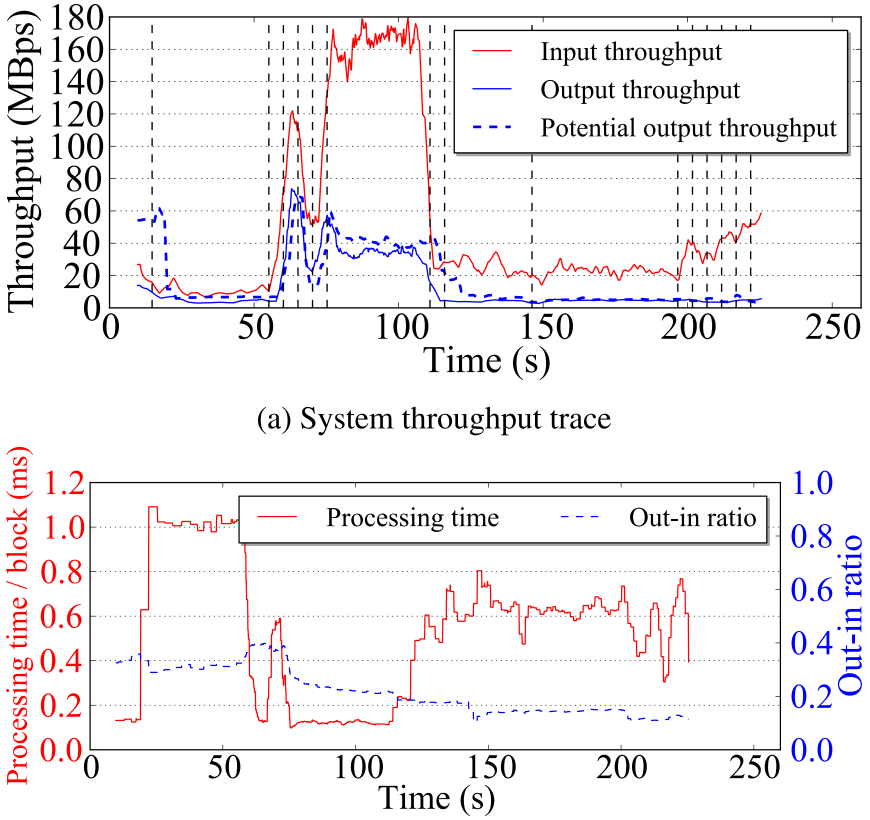
The second curve in the figure represents the potential output rate (blue dashed line). This shows what the output rate would be given the current configurations of the pipeline stages, if it were not limited by the network bandwidth. When using more expensive compression, the potential output rate drops. Ideally, this curve stays above the bandwidth line (so we do not under utilize the network), but as low as possible, indicating the system is using the most aggressive data reduction techniques without being CPU-bound. Here, too, we see that the system keeps this metric close to the network bandwidth limit.

The final curve is input throughput, which is the actual rate at which the modified memory and disk state emitted by QEMU/KVM is consumed by the pipeline (thick red line). This is the metric that ultimately determines how fast the handoff completes, and it depends on the actual output rate and the data compression. The system maximizes this metric, given the network and processing constraints.

The vertical dashed lines in the trace indicate the points at which the current operating mode is adjusted. As described in Section 4.4, the system bases the decision on measurements of P and R values (depicted in Figure 18-(b)) made every 100 ms. A decision is made every 5 seconds to let the effects of changing modes propagate through the system before the next decision point. In Figure 18-(a), our heuristic updates the operating mode 3 times during the VM handoff.

During the first 10 seconds of the trace, we observe high peaks of input and output throughput. During this period, the empty network buffers in the kernel and the inter-stage buffers in our pipeline absorb large volumes of data without hitting the network. Thus, the transient behavior is not limited by the network bottleneck. However, once the buffers fill, the system immediately adapts to this constraint.

The first decision, which occurs at approximately 10 seconds, changes the compression algorithm from its starting mode (GZIP level 1) to a much more compressive mode (LZMA level 5), adapting to low bandwidth. The effects of the mode change are evident in the traces of P and R (Figure 18-(b)), where processing time per block suddenly increases, and the out-in ratio drops after switching to the more expensive, but more compressive algorithm. In general, when the potential output is very high (with excess processing resources), the operating mode is shifted to a more aggressive technique that reduces the potential output rate closer to the bandwidth level, while increasing the actual input rate. Our system manages to find a good operating mode for this trace at this first decision point; the following two changes are only minor updates to the compression level.



(a) System throughput trace

(b) P and R trace

Figure 19: Dynamic adaptation for varying network bandwidth (MAR)

5.3.2 Adapting to Changing Conditions

Finally, we evaluate our adaptation mechanism in a case where conditions change during handoff. Figure 19-(a) shows a system throughput trace for the MAR workload, where network bandwidth is initially limited to 5 Mbps, but increases to 35 Mbps at 50 seconds, and reverts back to 5 Mbps at 100 seconds. In such a situation, no single static mode can do well – the ones that work well at high network bandwidth are ill-suited for low bandwidth, and vice versa.

Our adaptive system, however, reacts to these changes, and ensures a good operating mode is used throughout the trace. At the first decision point, our system selects high processing, high compression settings (LZMA, level 9) to deal with the very low network bandwidth. The output rate is limited by network, but the input rate is kept higher due to the greater level of compression (as shown in the P and R traces in Figure 19-(b)). After the bandwidth increases at 50 s, a decision is made at 58 s to switch back to GZIP compression to avoid being processing bound (as potential output is below the new network throughput). After a few minor mode changes, the system settles on a mode that fully utilizes the higher bandwidth (GZIP, level 7). Finally, a few seconds after bandwidth drops at time 100, our system once again switches to high compression (LZMA, level 9). The other mode changes are minor changes in compression level, which do not significantly affect P or R. Throughout the trace, the system manages to keep output throughput close to the network bandwidth, and potential output rate not much higher, thus maximally using processing resources. Through this dynamic adaptation, we complete VM handoff in 251 s, 31 s faster than with the best static operating mode.

6 Conclusion

The creation of a *Tactile Internet* [13], in which mobile devices leverage offload infrastructure at such extremely low latencies that it becomes possible to augment human perception and cognition in real time, is a vision that has captured the imagination of 5G wireless pioneers. Unfortunately, translating low first-hop wireless latency into low end-to-end latency is a difficult challenge today. Adding dispersed infrastructure in the form of cloudlets is an important step towards meeting this challenge [3, 24].

In the context of cloudlet offload, one can think of a small physical region around each cloudlet as its "cell size." Within this region, end-to-end latency is low enough to meet the demands of emerging Tactile Internet applications. When a user moves outside this region, end-to-end latency degrades unacceptably. Inspired by the success of cellular handoff in 3G/4G wireless networks, we have introduced the concept of VM handoff in this paper. We have shown how this mechanism can be implemented in today's Internet, even with inter-cloudlet bandwidths that are as low as 5 Mbps. We have also shown how adaptation can play an important role in VM handoff in order to cope with dynamic application behavior, and with dynamic variation in WAN bandwidth and cloudlet load. Our validation experiments confirm that the resulting mechanism is a promising building block for enabling user mobility in the Tactile Internet.

References

[1] AKAMAI. State of the Internet. `http://www.akamai.com/dl/akamai/akamai-soti-q114.pdf`.

[2] S. Akoush, R. Sohan, B. Roman, A. Rice, and A. Hopper. Activity based sector synchronisation: Efficient transfer of disk-state for wan live migration. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 22–31, 2011.

[3] P. Bahl, R. Y. Han, E. Li, and M. Satyanarayanan. Advancing the State of Mobile Cloud Computing. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, Low Wood Bay, Lake District, UK, 2012.

[4] S. Banerjee and D. O. Wu. Final report from the NSF Workshop on Future Directions in Wireless Networking. National Science Foundation, November 2013.

[5] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: Efficient idle desktop consolidation with partial vm migration. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 211–224, 2012.

[6] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 169–179, 2007.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[9] A. Collet, M. Martinez, and S. S. Srinivasa. The MOPED framework: Object Recognition and Pose Estimation for Manipulation. *The International Journal of Robotics Research*, 2011.

[10] P. Deutsch. DEFLATE Compressed Data Format Specification, 1996. `http://tools.ietf.org/html/rfc1951`.

[11] S. R. Ellis, K. Mania, B. D. Adelstein, and M. I. Hill. Generalizeability of Latency Detection in a Variety of Virtual Environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, 2004.

[12] ESTI. Mobile Edge Computing. `http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing`.

[13] G. P. Fettweis. A 5G Wireless Communication Vision. *Microwave Journal*, 55(12), December 2012.

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

[15] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-Time Provisioning for Cyber Foraging. In *Proceedings of MobSys 2013*, Taipei, Taiwan, June 2013.

[16] Kyungmin Lee and David Chu and Eduardo Cuervo and Johannes Kopf and Sergey Grizan and Alec Wolman and Jason Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Cloud Gaming. Technical Report MSR-TR-2014-115, August 2014.

[17] H.-F. Lai, Y.-S. Wu, and Y.-J. Cheng. Exploiting neighborhood similarity for virtual machine migration over wide-area network. In *Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability*, SERE '13, pages 149–158, 2013.

[18] Martin A. Brown. Traffic Control HOWTO. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html`.

[19] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty. Xvmotion: Unified virtual machine migration over long distance. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 97–108, 2014.

[20] OpenStack. `http://www.openstack.org/`, February 2015.

[21] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proceedings of INFOCOM 2012*, INFOCOM 2012, pages 181–189, 2012.

[22] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. Vmtorrent: virtual appliances on-demand. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 453–454, 2010.

[23] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. Vmtorrent: Scalable p2p virtual machine streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 289–300, 2012.

[24] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009.

[25] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible SPH. *ACM Trans. Graph.*, 28(3):40:1–40:6, July 2009.

[26] Sundaresan, S., De Donato, W., Feamster, N., Teixeira, R., Crawford, S., Pescape, A. Broadband internet performance: a view from the gateway. *ACM SIGCOMM computer communication review*, 41(4):134–145, 2011.

[27] 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.

[28] B. Taylor, Y. Abe, A. Dey, M. Satyanarayanan, D. Siewiorek, and A. Smailagic. The Usability Impact of Prefetching and Caching on Cloud-sourced Applications. *under review*, 2014.

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

[30] Wikipedia. List of countries by Internet connection speeds. `http://en.wikipedia.org/wiki/List_of_countries_by_Internet_connection_speeds`.

[31] Wikipedia. Lempel-Ziv-Markov chain algorithm, 2008. `http://en.wikipedia.org/w/index.php?title=Lempel-Ziv-Markov_chain_algorithm&oldid=206469040`.

[32] W. Zhang, K. T. Lam, and C. L. Wang. Adaptive live vm migration over a wan: Modeling and implementation. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 368–375, 2014.