You Can Teach Elephants to Dance:

Agile VM Handoff for Edge Computing

Kiryong Ha, Yoshihisa Abe*, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos,
Rohit Upadhyaya, Padmanabhan Pillai†, Mahadev Satyanarayanan

Carnegie Mellon University, †Intel Labs

ABSTRACT

VM handoff enables rapid and transparent placement changes to executing code in edge computing use cases where the safety and management attributes of VM encapsulation are important. This versatile primitive offers the functionality of classic live migration but is highly optimized for the edge. Over WAN bandwidths ranging from 5 to 25 Mbps, VM handoff migrates a running 8 GB VM in about a minute, with a downtime of a few tens of seconds. By dynamically adapting to varying network bandwidth and processing load, VM handoff is more than an order of magnitude faster than live migration at those bandwidths.

1 Introduction

Edge computing involves the execution of untrusted application code on computing platforms that are located close to users, mobile devices, and sensors. We refer to these platforms as *cloudlets.* A wide range of futuristic use cases that span low-latency mobile device offload, scalable video analytics, IoT privacy, and failure resiliency are enabled by edge computing [37]. For legacy applications, cloudlets can be used to extend virtual desktop infrastructure (VDI) to mobile users via a thin client protocol such as VNC [33].

To encapsulate application code for edge computing, mechanisms such as Docker are attractive because of their small memory footprint, rapid launch, and low I/O overhead. However, safety and management attributes such as platform integrity, multi-tenant isolation, software compatibility, and ease of software provisioning can also be important, as discussed in Section 2. When these concerns are dominant, classic virtual machine (VM) encapsulation prevails.

In this paper, we describe a mechanism called *VM handoff* that supports *agility* for cloudlet-based applications. This refers to rapid reaction when operating conditions change, thereby rendering suboptimal the current choice of a cloudlet. There are many edge computing situations in which agility is valuable. For example, an unexpected flash crowd may overload a small cloudlet and make it necessary to temporarily move some parts of the current workload to another cloudlet or the cloud. A second example is when advance knowledge is received of impending cloudlet failure due to a site catastrophe such as rising flood water, a spreading fire, or approaching enemy: the currently-executing applications can be moved to a safer cloudlet without disrupting service. Site failures are more likely at a vulnerable edge location than in a cloud data center. A third example arises in the context of a mobile user offloading a stateful latency-sensitive application such as wearable cognitive assistance [16]. The user's physical movement may increase end-to-end latency to an unacceptable level. Offloading to a closer cloudlet could fix this, provided application-specific volatile state is preserved.

VM handoff bears superficial resemblance to live migration in data centers [8, 27]. However, the turbulent operational environment of VM handoff is far more challenging than the benign and stable environment assumed for live migration. Connectivity between cloudlets is subject to widely-varying WAN latency, bandwidth, and jitter. In this paper, we use the range from 5 to 25 Mbps for our experiments, with the US average broadband Internet connectivity of 13 Mbps in 2015 falling in the middle of this range [2]. This is far from the 1–40 Gbps, low-latency and low-jitter connectivity that is typically available within data centers. VM handoff also differs from live migration in the primary performance metric of interest. *Down time,* which refers to the brief period towards the end when the VM is unresponsive, is the primary metric in live migration. In contrast, it is *total completion time* rather than down time that matters for VM handoff. In most use cases, prolonged total completion time defeats the original motivation for triggering the operation. Abe et al [1]

SEC '17, San Jose / Silicon Valley, CA, USA

 978-1-4503-5087-7/17/10…$15.00
DOI: 10.1145/3132211.3134453

*Now at Nokia Bell Labs. All the contributions to this work were made while at Carnegie Mellon University.

App1 periodically sends accelerometer readings from a mobile device to a Linux back-end that performs a compute-intensive physics simulation [39], and returns an image to be rendered.

App2 ships an image from a mobile device to a Windows back-end, where face recognition is performed [42] and labels corresponding to identified faces are returned.

App3 [41] is an augmented reality application that ships an image from a mobile device to a Windows back-end that identifies landmarks, and returns a modified image with these annotations.

App4 ships an image from a mobile device to a Linux back-end that performs object detection [9], and returns the labels of identified objects.

Figure 1: Offload Benchmark Suite for Edge Computing

have shown that a narrow focus on down time can lead to excessive total completion time.

The essence of our design is *preferential substitution of cloudlet computation for data transmission volume.* VM handoff dynamically retunes this balance in the face of frequent bottleneck shifts between cloudlet processing and network transmission. Using a parallelized computational pipeline to maximize throughput, it leverages a variety of data reduction mechanisms. We develop an analytic model of our pipeline, and derive an adaptation heuristic that is sensitive to varying bandwidth and cloudlet load. Our experiments confirm that VM handoff is agile, and reduces total completion time by one to two orders of magnitude relative to live migration.

2 VM Encapsulation for Edge Computing

There are opposing tensions in the choice of an encapsulation mechanism for edge computing. One key consideration is the memory footprint, launch speed and I/O performance degradation induced by the container. Lightweight mechanisms such as Docker minimally burden a cloudlet, and hence offer good scalability and return on hardware investment. Even lighterweight encapsulation is possible by simply using the Unix process abstraction as a container.

However, scalability and performance are not the only attributes of interest in edge computing. There are at least four other important attributes to consider. The first is *safety:* protecting the integrity of cloudlet infrastructure from potentially malicious application software. The second is *isolation:* hiding the actions of mutually untrusting executions from each other on a multi-tenant cloudlet. The third is *transparency:* the ability to run unmodified application code without recompiling or relinking. Transparency lowers the barrier to entry of cloudlet-based applications because it allows reuse of existing software for rapid initial deployment and prototyping. Refactoring or rewriting the application software to use lighterweight encapsulation can be done at leisure, after initial validation of the application in an edge computing context. A huge body of computer vision and machine learning software thus becomes immediately usable at the edge, close to sensors and video cameras. Transparency is especially valuable in use cases such as VDI, where the source code to legacy software may not be available. A fourth attribute is *deployability:* the ability to easily maintain cloudlets in the field, and to create mobile applications that have a high likelihood of finding a software-compatible cloudlet anywhere in the world [17]. Lighterweight encapsulation typically comes at the cost of deployability. A Docker-encapsulated application, for example, uses the services of the underlying host operating system and therefore has to be compatible with it. Process migration is an example of a lightweight service that has proven to be brittle and difficult to maintain in the field, even though there have been many excellent experimental implementations [4, 14, 30, 48].

For these four attributes, classic VM encapsulation is superior to lighterweight encapsulation techniques. Clearly, the optimal choice of encapsulation technique will be context-specific. It will depend on the importance of these attributes relative to memory footprint and CPU overhead. A hybrid approach, such as running many Docker containers within an outer encapsulating VM, is also possible.

In addition to these four attributes, the attribute of agility that was introduced in Section 1 is especially relevant. VM handoff is a mechanism that scores well on all five of these safety and management attributes for edge computing.

3 Poor Agility of Live Migration

Within a data center, *live migration* [8, 27] is widely used to provide the functionality targeted by VM handoff. Its design is optimized to take full advantage of LANs. Efforts to extend live migration to work over long distances [3, 26, 47] typically rely on dedicated high-bandwidth links between end points. The few works targeting low bandwidth migration [6, 49] either slow the running VM, or use a post-copy approach which may result in erratic application performance that is unacceptable for latency-sensitive use cases.

To illustrate the suboptimal behavior of live migration over WANs, we briefly preview results from experiments that are reported later in Section 5.3. Figure 1 describes the benchmark suite that is used for all experimental results reported in this paper. This suite is representative of latency-sensitive workloads in edge computing. Each application in the suite runs on an Android mobile device, and offloads computation in the critical path of user interaction to a VM-encapsulated application backend on a cloudlet. That VM is configured

VM	Total time	Down time	Transfer Size
App3	3126s (39%)	7.63s (11%)	3.45 GB (39%)
App4	726s (1%)	1.54s (20%)	0.80 GB (1%)

Average and Relative standard deviation of 3 runs. These results are extracted as a preview of Figure 7 in Section 5.3 See Figure 7 for full details.

Figure 2: Total Completion Time of Live Migration (10 Mbps)

with an 8 GB disk and 1 GB of memory. The VMM on the cloudlet is QEMU/KVM 1.1.1 on Ubuntu Linux.

Figure 2 presents the total completion times of live migration at a bandwidth of 10 Mbps. App4 takes 726 seconds to complete, which is hardly agile relative to the timeframes of cloudlet overload and imminent site failures discussed in Section 1. App3 suffers even worse. It requires a total completion time of 3126 seconds, with high variance. This is due to background activity in the Windows 7 guest that modifies memory fast enough to inordinately prolong live migration. We show later that VM handoff completes these operations much faster: 66 seconds and 258 seconds respectively.

The poor agility of live migration for edge computing is not specific to QEMU/KVM. Abe et al. [1] have shown that long total completion times are also seen with live migration implementations on other hypervisors such as Xen, VMware, and VirtualBox. In principle, one could retune live migration parameters to eliminate a specific pathological behavior such as App3 above. However, these parameter settings would have to be retuned for other workloads and operating conditions. Classic live migration is a fine mechanism within a bandwidth-rich data center, but suboptimal across cloudlets.

4 Basic Design and Implementation

VM handoff builds on three simple principles:

- *Every non-transfer is a win.* Use deduplication, compression and delta-encoding to ruthlessly eliminate avoidable transfers.
- *Keep the network busy.* Network bandwidth is a precious resource, and should kept at the highest possible level of utilization.
- *Go with the flow.* Adapt at fine time granularity to network bandwidth and cloudlet compute resources.

Figure 3 shows the overall design of VM handoff. Pipelined processing is used to efficiently find and encode the differences between current VM state at the source and already-present VM state at the destination (Section 4.1 and 4.2). This encoding is then deduplicated and compressed using multicore-friendly parallel code, and then transferred (Section 4.3). We describe these aspects of VM handoff in the sections below. For dynamic adaptation, the algorithms and parameters used in these stages are dynamically selected to match current processing resources and network bandwidth. We defer discussion of these mechanisms until Section 6.

4.1 Leveraging Base VM Images

VM handoff extends previous work on optimizations for content similarity on disk [29, 31], memory [15, 43], and rapid VM provisioning [17]. It leverages the presence of a base VM at the destination: data blocks already present there do not have to be transferred. Even a modestly similar VM is sufficient to provide many matching blocks. A handful of such base VMs are typically in widespread use at any given time. It would be straightforward to publish a list of such base VMs, and to precache all of them on every cloudlet. Our experiments use Windows 7 and Ubuntu 12.04 as base VM images. It should be noted that the correctness of VM handoff is preserved even though its performance will be affected by the absence of a base VM image at the destination.

4.2 Tracking Changes

To determine which blocks to transmit, we need to track differences between a VM instance and its base image. When a VM instance is launched, we first identify all the blocks that are different from the base VM. We cache this information in case of future launches of same image. It is also possible to preprocess the image and warm the cache even before the first launch. To track changes to a running VM disk state, VM handoff uses the Linux FUSE interface to implement a user-level filesystem on which the VM disk image is stored. All VM disk accesses pass through the FUSE layer, which can efficiently and accurately track modified blocks. A list of modified disk blocks relative to the base image is maintained, and is immediately available when migration is triggered. Since FUSE is used only for hooking, it does not break compatibility with the existing virtual disk image. In our implementation, we use raw virtual disk format but other formats can also be supported if needed. Also, as in [28], we have found that FUSE has minimal impact on virtual disk accesses, despite the fact that it is on the critical read and write paths from the VM to its disk.

Tracking VM memory modifications is more difficult. A FUSE-like approach would incur too much overhead on every memory write. Instead, we capture the memory snapshot at migration, and determine the changed blocks in our code. To get the memory state, we leverage QEMU/KVM's built-in live migration mechanism. When we trigger this mechanism, it marks all VM pages as read-only to trap and track any further modifications. It then starts a complete transfer of the memory state. We redirect this transfer to new VM handoff stages in the processing pipeline. As described in the following subsections, these stages filter out unmodified pages relative to the base VM, and then delta-encode, deduplicate, and compress the remaining data before transmission.

After this initial step, the standard iterative process for live migration takes over. On each iteration, the modified pages identified by QEMU/KVM are passed through the

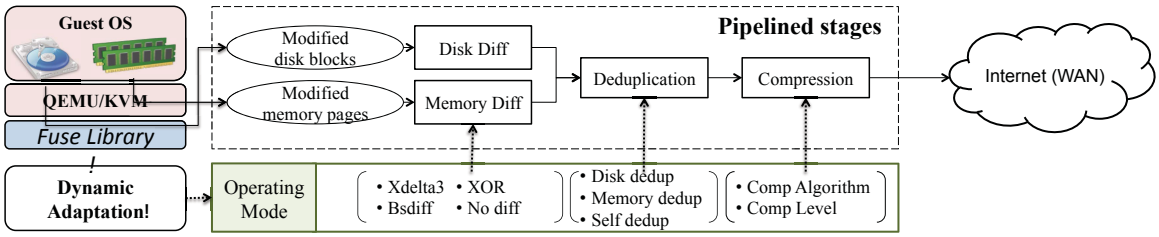


Figure 3: Overall System Diagram for VM handoff

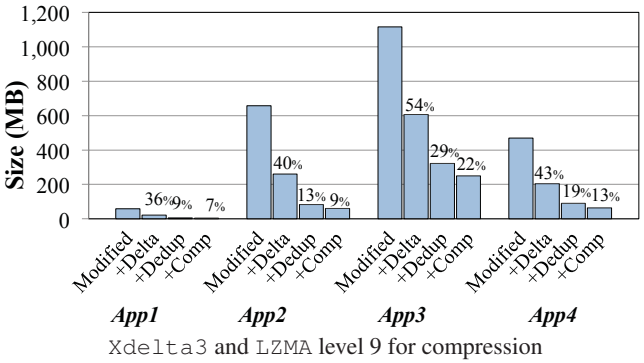


Figure 4: Cumulative Reductions in VM State Transfer

delta-encoding, deduplication and compression stages of our pipeline. During this process, it tries to leverage the base VM's memory as much as it can if the base VM's memory is found at the destination. To limit repeated transmission of hot pages, VM handoff regulates the start of these iterations and limits how many iterations are performed.

4.3 Reducing the Size of Transmitted Data

VM handoff implements a pipeline of processing stages to shrink data before it reaches the network. The cumulative reduction achieved by this pipeline can be substantial. For the four applications listed in Figure 1, the volume of data transferred is typically reduced to between 1/5 and 1/10 of the total modified data blocks. Figure 4 presents these results in more detail, showing the effect of individual stages in the pipeline. We describe these stages 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 3). The data streams are split into 4KB chunks, and are compared to the corresponding chunks in the base VM using their (possibly cached) SHA-256 hash values. Chunks that are identical to those in the base VM are omitted.

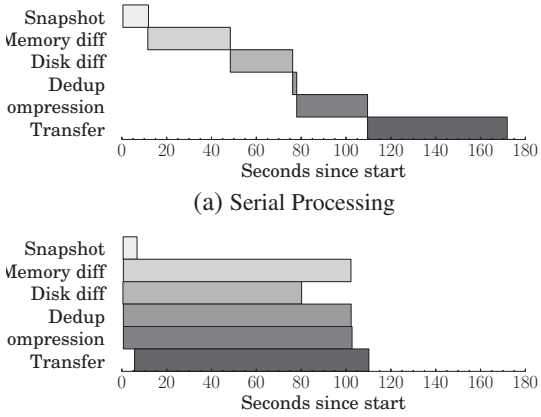
For each modified chunk, we a use binary delta algorithm to encode the difference between the chunk and its counterpart in the base VM image. If the encoding is smaller than the chunk, we transmit the encoding. The idea here is that small or partial modifications are common, and there may be significant overlap between the modified and original block when viewed at fine granularities. VM handoff can dynamically choose between `xdelta3`, `bsdiff4`, or `xor` to perform the binary delta encoding, or to skip delta encoding. We parallelize the compute-intensive hash computations and the delta encoding steps using multiple threads.

Deduplication: The streams of modified disk and memory chunks, along with the computed hash values, are merged and passed to the deduplication stage. Multiple copies of the same data commonly occur in a running VM. For example, 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 reduces to fast hash lookups operations and can therefore run as a single thread.

Compression: Compression is the final stage of the data reduction pipeline. We apply one of several off-the-shelf compression algorithms, including `GZIP` (deflate algorithm) [12], `BZIP2` [7], and `LZMA` [45]. These algorithms vary significantly in the compression achieved and processing speed. As compression works best on bulk data, we aggregate the modified chunk stream into approximately 1 MB segments before applying compression. We leverage multicore parallelism in this processing-intensive stage by running multiple instances of the compression algorithms in separate threads, and sending data segments to them round-robin.

4.4 Pipelined Execution

The data reduction stages described in the preceding sections ensure that only high-value data reaches the network. However, because of the processing-intensive nature of these stages, their serial execution can result in significant delay before the network receives any data to transmit. The network



(a) Serial Processing

(b) Pipelined Processing

App4: `xdelta3` and `LZMA` level 5

Figure 5: Serial versus Pipelined Data Reduction

is idle during this delay, and this represents a lost opportunity. Under conditions of low bandwidth, keeping the network filled with high-value data is crucial to good performance.

VM handoff pipelines the data reduction stages in order to fill the network as quickly as possible. This has the added benefit of reducing the amount of memory needed to buffer the intermediate data generated by the upstream stages, as they are consumed quickly by downstream stages. Figure 5 illustrates the benefit of VM handoff's pipelined implementation. In this example, total migration time is reduced by 36% (from 171.9 s down to 110.2 s).

5 Evaluation of Basic Design

In this section, we evaluate the performance of VM handoff under stable conditions that do not require dynamic adaptation. We defer evaluation under variable conditions until the adaptation mechanisms of VM handoff have been presented in Section 6. We answer the following questions here:

- Is VM handoff able to provide short total migration time on a slow WAN? (Section 5.2)
- Does VM handoff provide a significant performance win over live migration? How does it compare with non-VM approaches such as `Docker`? (Section 5.3)
- Is VM handoff able to scale well by making good use of extra cores on a cloudlet? (Section 5.4)

5.1 Experimental Setup

Our experiments emulate WAN-like conditions using the Linux Traffic Control (`tc` [25] 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 [2, 46], 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 source and destination cloudlet machines 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. All experiments use the benchmark suite in Figure 1, and start with the VM instance already running on the source cloudlet. Since these experiments focus on the non-adaptive aspects of VM handoff, the internal parameter settings remain stable throughout each experiment. Their initial values are chosen as described for the adaptive cases of Section 6.

5.2 Performance on WANs

Figure 6 presents the overall performance of VM handoff over a range of network bandwidths. *Total 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 total time, is the duration for which the VM is suspended. Even at 5 Mbps, total 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, total time and down time both shrink. At 15 Mbps using two cores, VM handoff completes within one minute for all of the workloads except App3, which is an outlier in terms of size of modified memory state (over 1 GB, see Figure 4). The other outlier is App1, whose modified state is too small for effective adaptation.

More bandwidth clearly helps speed up total completion time. Note, however, that the relationship of completion time to bandwidth is not a simple linear inverse. Two factors make it more complex. First, at low bandwidths, slow data transfers give the VM more time to dirty pages, increasing the total data volume and hence transfer time. On the other hand, these quantities are reduced by the fact that VM handoff has more time to compress data. The relative effects of these opposing factors can vary greatly across workloads.

5.3 Comparison to Alternatives

What is the agility of VM handoff relative to alternatives? Figure 7 contrasts VM handoff and QEMU/KVM live migration (version 1.1.1) between two cloudlets connected by a 10 Mbps, 50 ms RTT WAN. The WAN is emulated by Linktropy hardware on a gigabit Ethernet. There is no shared storage between cloudlets. To ensure a fair comparison, live migration is configured so that the destination cloudlet already has a copy of the base VM image into which the application and its supporting toolchain were installed to construct the benchmark VM. These "pre-copied" parts of the benchmark VM state do not have to be transferred by live migration.

BW	5 Mbps		10 Mbps		15 Mbps		20 Mbps		25 Mbps	
Time (s)	Total	Down	Total	Down	Total	Down	Total	Down	Total	Down
App 1	25.1	4.0 (5%)	24.6	3.2 (29%)	23.9	2.9 (38%)	23.9	3.0 (38%)	24.0	2.9 (43%)
App 2	247.0	24.3 (3%)	87.4	15.1 (10%)	60.3	11.4 (8%)	46.9	7.0 (14%)	39.3	5.7 (25%)
App 3	494.4	24.0 (4%)	257.9	13.7 (25%)	178.2	8.8 (19%)	142.1	7.1 (24%)	121.4	7.8 (22%)
App 4	113.9	15.8 (6%)	66.9	7.3 (42%)	52.8	5.3 (12%)	49.1	6.9 (12%)	45.0	7.1 (30%)

(a) 1 CPU core

BW	5 Mbps		10 Mbps		15 Mbps		20 Mbps		25 Mbps	
Time (s)	Total	Down	Total	Down	Total	Down	Total	Down	Total	Down
App 1	17.3	4.1 (6%)	15.7	2.5 (4%)	15.6	2.2 (14%)	15.4	2.0 (19%)	15.2	1.9 (20%)
App 2	245.5	26.5 (7%)	77.4	14.7 (24%)	48.5	6.7 (15%)	36.1	3.6 (12%)	31.3	4.1 (17%)
App 3	493.4	24.5 (10%)	250.8	12.6 (13%)	170.4	9.0 (17%)	132.3	7.3 (20%)	109.8	6.5 (22%)
App 4	111.6	17.2 (7%)	58.6	5.5 (5%)	43.6	5.5 (31%)	34.1	2.1 (22%)	30.2	2.1 (26%)

(b) 2 CPU cores

Average of 5 runs and relative standard deviations (RSDs, in parentheses) are reported. For total migration 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 6: Total Completion Time and Down Time of VM handoff on WANs

VM	Approach	Total time (s)	Down time (s)	Transfer size (MB)
App1	VM handoff	16 (3 %)	2.5 (4%)	7
	Live Migration	229 (1 %)	2.4 (22 %)	235
App2	VM handoff	77 (4 %)	14.7 (24 %)	84
	Live Migration	6243 (68 %)	5.5 (17 %)	6899
App3	VM handoff	251 (1 %)	12.6 (13 %)	276
	Live Migration	3126 (39 %)	7.6 (11 %)	3533
App4	VM handoff	59 (1 %)	5.5 (5 %)	61
	Live Migration	726 (1 %)	1.5 (20 %)	82

Figure 7: VM handoff vs. KVM/QEMU Live Migration at 10 Mbps

VM	Approach	Total time	Down time	Transfer size
App1	VM handoff	15.7s	2.5s	7.0 MB
	Docker	6.9s	6.9s	6.5 MB
App4	VM handoff	58.6s	5.5s	61 MB
	Docker	118s	118s	98 MB

Only App1 and App4 are shown because `Docker-CRIU` works for only Linux Apps.

Figure 8: VM handoff vs. Docker at 10 Mbps

In other words, the comparison already factors into live migration a component source of efficiency for VM handoff. Despite this, live migration performs poorly relative to VM handoff. For every app in the benchmark suite, VM handoff improves total completion time by an order of magnitude.

Live migration would perform even worse if the base VM were not present at the destination. For example, our experiments take over two hours for App4, and two and a half hours for App3. We omit the detailed results to save space.

How much would lighterweight encapsulation improve agility? To explore this question, we compare VM handoff with `Docker` [13]. Although Docker does not natively support migration of a running container, a form of migration can be achieved with Checkpoint/Restore in Userspace (`CRIU`) [10]. This involves suspending the container and copying memory and disk state to the destination. So, unlike VM handoff, down time equals total completion time.

Figure 8 compares VM handoff to Docker migration for the two Linux applications in our benchmark suite (`Docker-CRIU` only works for Linux apps). Given the reputation of VM encapsulation as being heavyweight and unwieldy, we would expect the agility of VM handoff to be horrible relative to Docker migration. In fact, the results are far more reassuring. For App4, VM handoff is actually *twice as fast* as Docker. For App1, the total state is so small that the VM handoff optimizations do not really kick in. Though Docker takes roughly half the total completion time, its down time is significantly longer than that of VM handoff.

From the viewpoint of agility, VM encapsulation is thus surprisingly inexpensive in light of its safety and management benefits that were discussed in Section 2. The issue of large memory footprint and CPU overhead continue to be concerns for VM encapsulation, but agility need not be a concern.

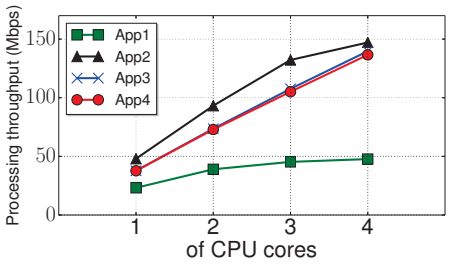


Figure 9: Processing Scalability of the System

5.4 Multicore Scalability

The pipelining in VM handoff enables good use to be made of extra cloudlet cores. Figure 9 shows the processing throughput for different workloads as the number of CPU cores increases. For Apps 2–4, the improvement is almost linear as we use more cores. For App1, the total volume of data processed is too small to benefit from multiple cores.

5.5 Importance of Memory State

As mentioned in Section 4.1, VM handoff aggressively leverages available state at the destination cloudlet to minimize transfer size. Our system can leverage both disk and memory state at the destination. Disk state clearly helps, but as memory state is much more volatile and dynamic, how much does the memory state at the destination cloudlet really help?

When the base VM memory snapshot is available, total completion time is reduced significantly: 17% (App3 and App4), 36% (App2) and 40% (App1). This indicates that there are indeed many identical memory pages between a running application VM and a freshly-booted VM instance of its base VM image. Note that the base VM image does not have to be an exact match. A VM image of a roughly similar guest operating system is good enough, since it is likely to contain many similar 4 KB chunks of data.

6 Dynamic Adaptation

Network bandwidth and processing capacity can vary considerably in edge computing, even over timescales of a few seconds to tens of seconds. Cloudlet load can change due to new users arriving or departing; available network bandwidth can change due to competing traffic; and, the compressibility of VM state can change as its contents change. It is thus important to dynamically re-optimize the compute-transmit tradeoff during VM handoff.

The effect of imbalance is visible in the example shown earlier in Figure 5. Due to the particular choices of algorithms and parameters used, the system is clearly CPU bound: processing takes 109.6 s (from the start of memory snapshotting to the end of compression), while network transfer only requires 62.3 s. After pipelining, the larger of these two values determines total completion time. A better choice of parameters would have selected less aggressive compression to reduce processing demand, even at the expense of increased data transfer size and hence longer transmit time.

VM handoff continuously monitors relevant parameters at the timescale of tens of milliseconds, and uses this information to dynamically adapt settings even during the course of a single migration. We refer to a specific selection of algorithms and parameters as an *operating mode* of VM handoff.

6.1 Pipeline Performance Model

To select the right operating mode, we model VM handoff's processing pipeline as shown in Figure 10. Each stage in the pipeline outputs a lossless but smaller version of its input data. For each stage i, we define:

$$P_i = \text{processing time},\ R_i = \frac{\text{output size}}{\text{input size}} \quad \text{at stage i} \tag{1}$$

Because of pipelining, the total migration time is the larger of the processing time and the network transfer time:

$$Time_{processing} = \sum_{1 \le i \le n} P_i \tag{2}$$

$$Time_{network} = \frac{Size_{migration}}{\text{Network bandwidth}} \tag{3}$$

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

$$Time_{migration} = max(Time_{processing}, Time_{network}) \tag{4}$$

In the implementation, we use measurable short-term quantities (processing throughput and network throughput) rather than parameters that depend on indeterminate information such as total input size. Total system throughput is then:

$$Thru_{system} = min(Thru_{processing}, Thru_{network}) \tag{5}$$

$$Thru_{processing} = \frac{1}{\sum_{1 \le i \le n} P_i}$$
$$Thru_{network} = \frac{\text{Network Bandwidth}}{(R_1 \times \cdots \times R_n)} \tag{6}$$

Intuitively, (5) and (6) show that system throughput can be estimated by measuring processing time (P), current network bandwidth, and out-in ratio (R).

6.2 Adaptation Heuristic

Based on this model, we develop a heuristic for dynamically adapting the operating mode to maximize $Thru_{system}$. A

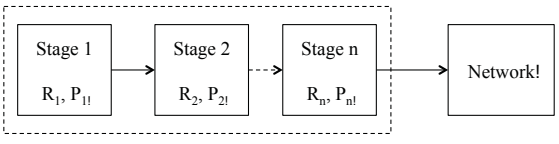
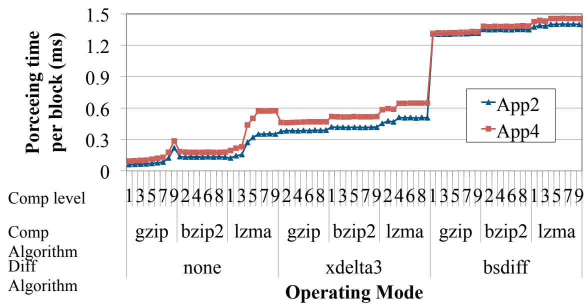
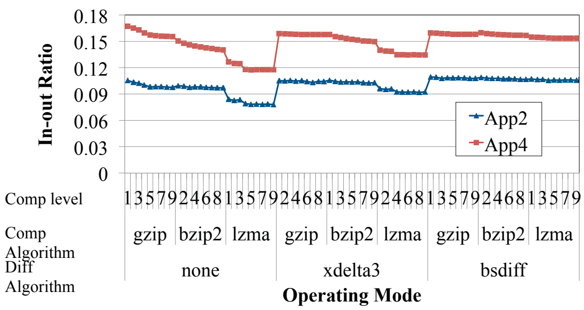


Figure 10: Model of VM handoff Pipeline



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



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

(G: Gzip, B: Bzip2, L: LZMA)

Figure 11: Trends of P and R Across Workloads

concern is that the quantities P and R are highly specific to workload, even at the same compression settings. Figure 11 illustrates this with measured P and R values for varying operating modes of App2 and App4. Each data point is P or R per block (i.e. modified memory page or disk blocks), for a particular mode. Ideally, we would like an adaptation mechanism that does not require advance profiling of a workload to determine P and R. A critical insight is that although P and R can vary significantly across workloads, the ratio of P (or R) values between different operating modes are relatively stable across workloads. In other words, the operating mode trends of P and R are workload-independent. From our measurement, the average correlation values of all pairs of 4 workloads are 0.99 for P and 0.84 for R. From this insight, the heuristic iterates as follows:

(1) First, measure the current P and R values ($P_{current}$ and $R_{current}$) of the ongoing handoff. Also measure the current network bandwidth by observing the rate of acknowledgments from the destination cloudlet.
(2) From a lookup table that was constructed offline from a reference workload (not necessarily the current workload), find P_{lookup} and R_{lookup} for proposed operating mode, M, and then compute these scaling factors:

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

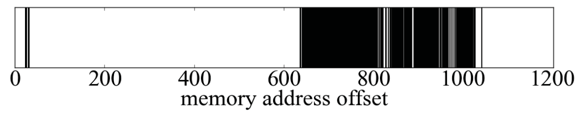


Figure 12: Modified Memory Regions for App4 (Black)

(3) Apply these scaling factors to adjust the measured P and R values of the ongoing migration. Then, use (6), to calculate $Thru_{processing}$ and $Thru_{network}$ for each operating mode being considered.
(4) Using (5), select the operating mode that maximizes $Thru_{system}$.

This heuristic is reactive to changes in network bandwidth, cloudlet load, and compressibility of modified VM state. The adaptation loop is repeated every 100 ms to ensure agility. An updated operating mode will last for at least 5 s to provide hysteresis in adaptation.

6.3 Workload Leveling

Guest operating systems often manage memory in a way that clusters allocations, and therefore modifications. As Figure 12 illustrates, the set of modified pages for a typical VM tends to be highly clustered rather than uniformly distributed over physical memory. Sending this memory snapshot to our processing pipeline would result in a 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.

We address this problem by randomizing the order of page frames passed to the processing pipeline. Now, 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. Neither processing nor network resources are left idling for long. More importantly, the ratio of modified and unmodified pages arriving at the processing pipeline is less bursty.

Figure 13 shows how randomization of page order reduces the burstiness of processing demand. The spikes correspond to CPU-bound conditions, causing network underutilization; while the troughs result in CPU under-utilization due to network bottlenecks. Randomization avoids the extremes and helps VM handoff to efficiently use both resources. Note that no adaptation is performed here (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.

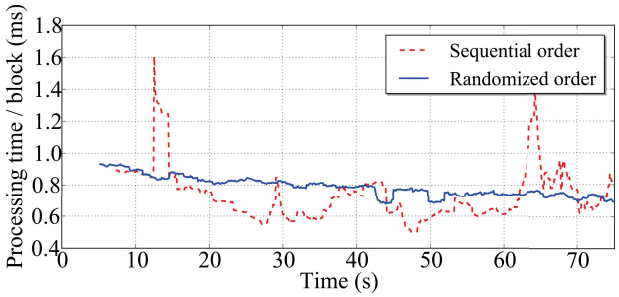


Figure 13: Randomized vs Sequential memory order (1-second moving average)

6.4 Iterative Transfer for Liveness

VM handoff has to strike a delicate balance between minimizing service disruption and unacceptably extending completion time. If total completion 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 be too disruptive for good user experience (many minutes or more of cloudlet service outage on a slow WAN). 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 total completion time unacceptably. We achieve balance by using an input queue threshold of 10 MB of modified state to trigger the next iteration, and an iteration time of 2 s or less to trigger termination. These values are based on our empirical observations of VM handoff.

7 Evaluation of Dynamic Adaptation

In this section, we evaluate how well the adaptive mechanisms of VM handoff respond to dynamically changing conditions of network bandwidth and cloudlet load. Using the same experimental setup described in Section 5.1, we answer the following questions:

- How effective is VM handoff's dynamic selection of operating modes? How does this adaptation compare to the static modes? (Section 7.1)
- How rapidly does VM handoff adapt to changing conditions? (Section 7.2)

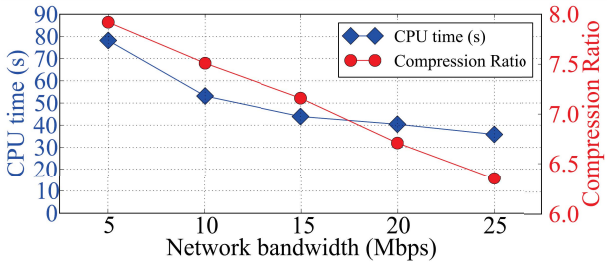


Figure 14: Operating Mode Selection (App4, 1 core)

7.1 Operating Mode Selection

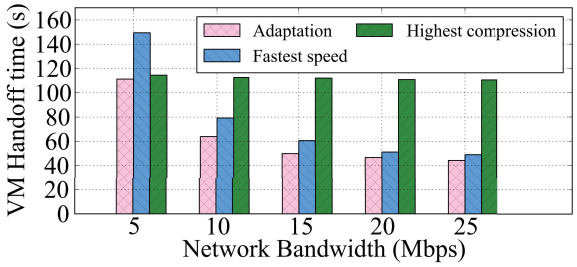
For App4, Figure 14 shows the processing time and compression ratios of VM handoff as bandwidth varies. "CPU time" is the absolute CPU usage, in seconds, by VM handoff. "Compression ratio" is the ratio of the input data size (i.e., modified VM state) to the output data size (i.e., final data shipped over the network). When bandwidth is low, more CPU cycles are used to aggressively compress VM state and thus reduce the volume of data transmitted. At higher bandwidth, fewer CPU cycles are consumed so that processing does not become the bottleneck. The average CPU utilization remains high (between 80% and 90%) even though absolute CPU usage, in seconds, drops as the network bandwidth increases. This confirms that VM handoff successfully balances processing and transmission, while using all available resources.

This behavior of VM handoff is consistent across all applications except App1, which has too little modified state for for adaptation to be effective. Generally, adaptation is more effective when larger amounts of data are involved. So App2 and App4 with 500–800 MB of modified data show median improvements, while App3 (> 1.3 GB) and App1 (< 70 MB) are two extreme ends. Due to space limitations, we only present the results for App4. To avoid any workload-specific bias, the adaptation lookup table (mentioned in Section 6.2) is created using App2, but tested using App4.

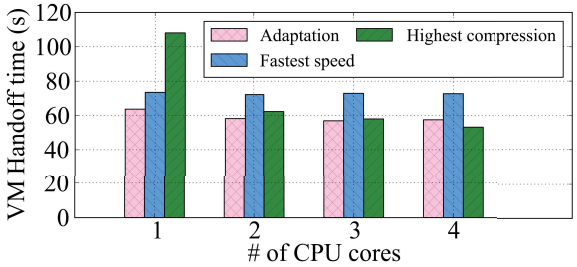
Comparison with Static Modes: How effective is adaptation over picking a static configuration? To evaluate this, we first compare against two operating modes: HIGHEST_COMPRESSION and FASTEST_SPEED. For FASTEST_SPEED, each stage is tuned to use the least processing resources. In HIGHEST_COMPRESSION, we exhaustively run all the combinations and choose the option that minimizes data transfer size. Note that the most CPU-intensive mode might not be the highest compression mode. Some configurations incur high processing costs, yet only achieve low compression rates.

Figure 15(a) shows that 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 migration time when bandwidth is low, but is worse than the other approaches at higher bandwidth. This is because its speed becomes limited by computation, so bandwidth is not fully utilized, and it is largely unaffected by bandwidth changes. Unlike the two static cases, 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 15(b) shows the total completion time for differing numbers of CPU cores, with fixed 10 Mbps bandwidth.



(a) Varying Network Bandwidth (1 CPU core)



(b) Varying CPU Cores (10 Mbps bandwidth)

Figure 15: Adaptation vs Static Modes (App4)

FASTEST_SPEED shows constant total completion time regardless of available cores, because it is limited by bandwidth not processing. HIGHEST_COMPRESSION improves as we assign more CPU cores. Again, adaptation equals or improves upon the static operating modes in all cases.

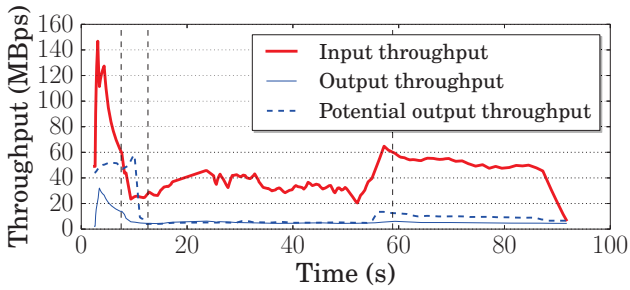
Exhaustive Evaluation of Static Modes: We have shown that adaptation performs better than two distinctive static modes. Note that it is not trivial to determine *a priori* whether either of these static modes, or one from the many different possible modes, would work well for a particular combination of workload, bandwidth, and processing resources. For example, our adaptation heuristic selects 15 different operating modes for App4 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.

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 total migration times for all possible operating modes. Figure 16 compares the best static operating mode with adaptation for App4 at varying network bandwidth. To compare adaptation results with the top tier of static operating modes, we also present the 10th percentile performance among the static modes for each condition. The adaptation results are nearly as good as the best static mode for each case. Specifically, adaptation almost always ranks within the top 10 among the 108 possible operating modes. This confirms the efficacy of adaptation in VM handoff.

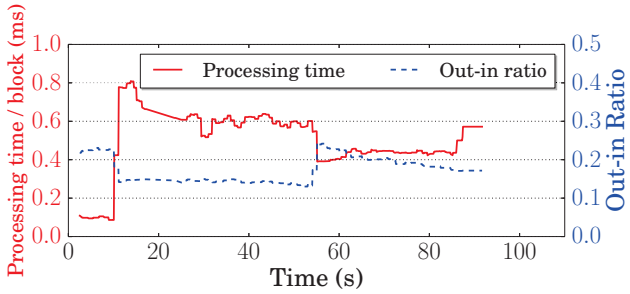
BW (mbps)	Approach	Total time	Down time
5	Adaptation	113.9s (3 %)	15.8s (6 %)
	Best static	111.5s (1 %)	15.9s (12 %)
	Top 10%	128.3s (2 %)	20.7s (9 %)
10	Adaptation	66.9s (6 %)	7.3s (42 %)
	Best static	62.0s (1 %)	5.0s (11 %)
	Top 10%	72.1s (1 %)	4.8s (3 %)
20	Adaptation	49.1s (8 %)	6.9s (12 %)
	Best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

Relative standard deviation in parentheses

Figure 16: Adaptation vs Static Mode for App4 (1 core)



(a) System throughput trace



(b) P and R trace

Figure 17: Adaptation Trace (App4, 5 Mbps and 1 CPU)

7.2 Dynamics of Adaptation

VM handoff uses dynamic adaptation to adjust operating modes as conditions change. To evaluate this process, we study traces of execution under static and dynamic conditions.

Adapting to Available Resources: Figure 17(a) is an execution trace of VM handoff, showing various throughputs achieved at different points in the system. 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 CPU bound and cannot fully utilize the network. The figure shows that the output rate closely tracks bandwidth (5 Mbps).

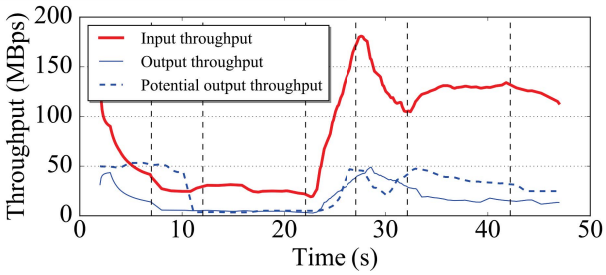
The second curve in the figure represents the potential output rate (blue dashed line). This shows what the output rate would be given at the current operating mode of the pipeline stages, if it were not limited by the network bandwidth. When using more expensive compression, the potential output rate drops. This curve stays above the bandwidth line (so we do not underutilize the network), but as low as possible, indicating the system is using the most aggressive data reduction without being CPU-bound.

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

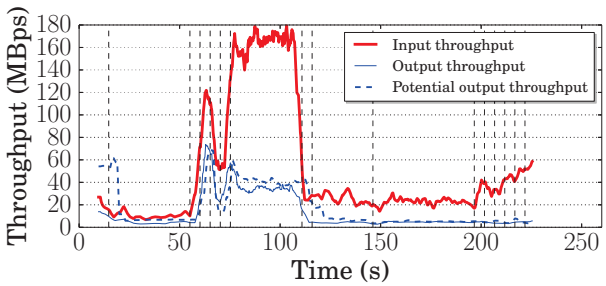
The vertical dashed lines in the trace indicate the points at which the current operating mode is adjusted. As described in Section 6, the system bases the decision on measurements of P and R values (depicted in Figure 17(b)) made every 100 ms. A decision is made every 5 seconds to let the effects of changing modes propagate through the system before the next decision point. In Figure 17(a), our heuristic updates the operating mode 3 times during the migration.

During the first 10 seconds, we observe high peaks of input and output throughput because the empty network buffers in the kernel and 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 17(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. VM handoff manages to find a good operating mode at this first decision point; the following two changes are only minor updates to the compression level.



(a) App4: 5 Mbps → 35 Mbps at 20 s



(b) App3: 5 Mbps → 35 Mbps at 20 s → 5 Mbps at 100 s

Figure 18: Adaptation for Network Bandwidth

Adapting to Changing Conditions: Finally, we evaluate VM handoff in a case where conditions change during migration. Figure 18(a) shows a system throughput trace for App4, where network bandwidth is initially 5 Mbps, but increases to 35 Mbps at 20 seconds. VM handoff reacts quickly to ensure that a good operating mode is used throughout the trace. At the first decision point (vertical dashed line), the mechanism 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. When the bandwidth increases at 20 s, VM handoff switches to the most lightweight operating mode (GZIP, level 1, No diff) to avoid being processing bound. The other mode changes are minor.

We also evaluate with App3, which provides longer migration times and allows us to test multiple bandwidth changes. In this case, bandwidth starts at 5 Mbps, then increases to 35 Mbps at 50 seconds, and finally reverts back to 5 Mbps at 100 seconds. Figure 18(b) shows how the various system throughputs change over time. As for the App4 trace, at the first decision point, VM handoff selects high processing, high compression settings (LZMA, level 9) to deal with the very low network bandwidth. At 58 s, a major decision is made 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, VM handoff again switches to high compression (LZMA, level 9).

		Total time (s)	Downtime (s)
Live Migration	1 Gbps	946.4 (0 %)	0.2 (52%)
VM handoff	1 Gbps	87.8 (24 %)	6.6 (20 %)
	100 Mbps	111.0 (29 %)	7.4 (20 %)
	25 Mbps	101.8 (3 %)	10.8 (4 %)
	20 Mbps	114.6 (2 %)	13.2 (6 %)
	15 Mbps	584.8 (7 %)	16.2 (5 %)
	10 Mbps	656.4 (1 %)	21.4 (10 %)

Average of 5 runs, with RSD in parentheses.
Windows 7 guest OS, 32 GB disk image, 1 GB memory
Added latency: 0 for 1 Gbps; 25 ms for 100 Mbps; 50 ms for all other bandwidths

Figure 19: VM handoff for WAN VDI

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.

When the network bandwidth changes, 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. We verify that by comparing our result with all possible static operating modes under these varying network conditions. For the App3 experiment, the best static operating mode completes VM handoff in 282 s, which is 31 s slower than our dynamic adaptation result of 251 s.

VM handoff adaptation also works well as the available number of CPU cores changes (e.g. due to load), and outperforms the best static operating mode. Details of the study have been omitted for space reasons.

8 Legacy Applications

Our experimental results until now have focused on forward-looking applications such as augmented reality and wearable cognitive assistance. The benchmark suite in Figure 1 is representative of the backends of such applications, and the associated VM images are relatively small. However, VM handoff can also be useful for VDI-based legacy applications. A user could crisply interact with his virtual desktop on a nearby cloudlet via a remote desktop protocol such as VNC [33]. When the user travels to distant locations, VM handoff can ensure that his virtual desktop "follows him around" from cloudlet to cloudlet. It could also move the desktop to a cloudlet on an aircraft while it is tethered on the ground, and then allow the user to continue working during flight even without Internet connectivity.

To explore how well VM handoff works for VDI use cases, we installed Windows 7 as the guest OS in a VM with a 32 GB disk and 1 GB memory. At 1 Gbps, Figure 19 shows that live migration takes over 15 minutes to complete. This is due to background activity from Windows Update that dirties a considerable amount of memory state, thereby prolonging live migration. During this entire period, the remote user will continue interacting with the (presumably suboptimal) source cloudlet and thus suffer poor user experience. By design, live migration has very low down time (barely 200 milliseconds in this case). However, this does not translate into a significant improvement in user experience in the VDI context. A user would much rather reduce the 15-minute period of poor interactive experience, even at the cost of a few seconds of down time. Figure 19 confirms that this is indeed the case with VM handoff. At 1 Gbps, the total time is close to 1.5 minutes which is an order of magnitude shorter than the 15 minutes for live migration. As bandwidth drops, total time rises but remains under two minutes even down to 20 Mbps. The increase in down time is modest, just under 15 seconds. Even at 10 Mbps, the total time of just over ten minutes is well below the 15-minute figure for live migration at 1 Gbps. The down time of just over 20 seconds is acceptable for VDI.

9 Related Work

The use of VM encapsulation to move live execution state across machines was introduced in 2002 in the context of Internet Suspend/Resume [22]. That work, as well as others [35, 44], used what is now known as a *stop-and-copy* approach that requires a VM instance to be suspended during transfer. Augmented with partial demand fetch and other optimizations, this approach has been applied to VDI use cases in projects such as Internet Suspend/Resume [21, 38] and the Collective [34, 36].

Live migration, where a VM instance is moved without disrupting its execution, was first introduced in 2005 [8, 27] and has since undergone many refinements and optimizations [5, 11]. It has always been viewed as a technique for the stable and bandwidth-rich environment of a data center. Even refinements to span dispersed data centers have assumed high-bandwidth data paths. These approaches are characterized as *pre-copy* approaches since control is not transferred to the destination until all VM state has been copied there. The opposite approach is *post-copy* migration, where VMs are resumed on their destination first, and then their state is retrieved [6, 19, 20, 23, 47]. Hybrid approaches utilizing pre-copy and post-copy have also been proposed [24]. Finally, various optimizations to migration have been proposed, including page compression [18, 40] and guest throttling [6, 26]. Approaches such as VM distribution networks [29, 31, 32] have also been proposed. *Enlightened post-copy* has been proposed as an approach that leverages higher-level knowledge from within the guest environment [1].

Unfortunately, post-copy approaches incur unpredictable delays when missing state is accessed by the VM, and has to be fetched from the source. This makes post-copy approaches unacceptable for low-latency applications such as augmented

reality, where edge computing offers the promise of low average latency as well as low jitter. VM handoff therefore preserves the pre-copy approach of classic live migration while extensively optimizing its implementation for WANs and the unique challenges of edge computing.

10 Conclusion

As edge computing gains momentum, there is growing debate on the form of encapsulation to used for application code that is executed on cloudlets. Considerations of small memory footprint, rapid launch, and low I/O overhead suggest that lightweight encapsulations such as Docker are a natural fit for edge computing. However, as the early sections of this paper have discussed, these are not the only attributes of importance in edge computing. Safety and management attributes such as platform integrity, multi-tenant isolation, software compatibility, and ease of software provisioning can also be important in many edge computing use cases. When those concerns are dominant, classic VM encapsulation is superior.

An agile edge computing system reacts rapidly when operating conditions change. When agility is important, this paper shows that the large size of VM images need not be a deterrent to their use. We have presented the design, implementation and evaluation of VM handoff, a mechanism that preserves the many attractive properties of classic live migration for data centers while optimizing for the turbulent operational environment of edge computing. Rather than requiring bandwidths in the 1-40 Gbps typical of data centers, VM handoff operates successfully at WAN bandwidths as low as 5-25 Mbps. VM handoff achieves this improvement through preferential substitution of cloudlet computation for data transmission volume. It dynamically retunes this balance in the face of frequent bottleneck shifts betweeen cloudlet processing and network transmission. It uses a parallelized computational pipeline to achieve maximal throughput while leveraging a variety of data reduction mechanisms. Our experimental results validate the efficacy of these mechanisms.

Acknowledgements

This research was supported in part by the National Science Foundation (NSF) under grant number CNS-1518865, and by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0051. Additional support was provided by Intel, Google, Vodafone, Deutsche Telekom, Verizon, Crown Castle, NTT, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view(s) of their employers or the above-mentioned funding sources.

REFERENCES

[1] Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, and Mahadev Satyanarayanan. 2016. Urgent Virtual Machine Eviction with Enlightened Post-Copy. In *Proceedings of VEE 2016*.

[2] AKAMAI. 2015. State of the Internet. https://www.akamai.com/us/en/multimedia/documents/report/q3-2015-soti-connectivity-final.pdf, (2015).

[3] Sherif Akoush, Ripduman Sohan, Bogdan Roman, Andrew Rice, and Andy Hopper. 2011. 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)*. 22–31.

[4] Y. Artsy and R. Finkel. 1989. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer* 22, 9 (1989).

[5] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. 2012. Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. 211–224.

[6] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. 2007. Live Wide-area Migration of Virtual Machines Including Local Persistent State. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. 169–179.

[7] Michael Burrows and David Wheeler. 1994. A block-sorting lossless data compression algorithm. In *DIGITAL SRC RESEARCH REPORT*. Citeseer.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 273–286.

[9] Alvaro Collet, Manuel Martinez, and Siddhartha S. Srinivasa. 2011. The MOPED framework: Object Recognition and Pose Estimation for Manipulation. *The International Journal of Robotics Research* (2011).

[10] CRIU. Dec. 2015. Checkpoint/Restore in Userspace. https://criu.org/Main_Page, (Dec. 2015).

[11] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, 14. http://dl.acm.org/citation.cfm?id=1387589.1387601

[12] P. Deutsch. 1996. DEFLATE Compressed Data Format Specification. (1996). http://tools.ietf.org/html/rfc1951.

[13] Inc Docker. 2016. Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/. (2016).

[14] Fred Douglis and John K. Ousterhout. 1991. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience* 21, 8 (1991).

[15] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association. http://dl.acm.org/citation.cfm?id=1855741.1855763

[16] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards Wearable Cognitive Assistance. In *Proceedings of MobSys 2014*. Bretton Woods, NH.

[17] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. 2013. Just-in-Time Provisioning for

Cyber Foraging. In *Proceedings of MobSys 2013*.

[18] Stuart Hacking and Benoit Hudzia. 2009. Improving the Live Migration Process of Large Enterprise Applications. In *Proceedings of the Third International Workshop on Virtualization Technologies in Distributed Computing*. Barcelona, Spain.

[19] Michael Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines. *SIGOPS Operating Systems Review* 43, 3 (July 2009).

[20] Michael R. Hines and Kartik Gopalan. 2009. Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Washington, DC.

[21] Michael Kozuch, M. Satyanarayanan, Thomas Bressoud, and Yan Ke. 2002. *Efficient State Transfer for Internet Suspend/Resume*. Technical Report Technical Report IRP-TR-02-03. Intel Research Pittsburgh.

[22] Michael A. Kozuch and Mahadev Satyanarayanan. 2002. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*. Callicoon, NY.

[23] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. 2009. Live Migration of Virtual Machine Based on Full System Trace and Replay. In *Proceedings of the Eighteenth ACM International Symposium on High Performance Distributed Computing*. Garching, Germany.

[24] Peng Lu, Antonio Barbalace, and Binoy Ravindran. 2013. HSG-LM: Hybrid-copy Speculative Guest OS Live Migration Without Hypervisor. In *Proceedings of the Sixth International Systems and Storage Conference*. Haifa, Israel.

[25] Martin A. Brown. 2015. Traffic Control HOWTO. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html. (2015).

[26] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. 2014. XvMotion: Unified Virtual Machine Migration over Long Distance. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*.

[27] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. 2005. Fast Transparent Migration for Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC '05)*. USENIX Association, 1. http://dl.acm.org/citation.cfm?id=1247360.1247385

[28] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. 2011. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. ACM, San Jose, California, USA. DOI: http://dx.doi.org/10.1145/1996130.1996152

[29] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. 2012. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proceedings of INFOCOM 2012*.

[30] M.L. Powell and B.P. Miller. 1983. Process Migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*. Bretton Woods, NH.

[31] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. 2010. VMtorrent: virtual appliances on-demand. In *Proceedings of the ACM SIGCOMM 2010 conference*.

[32] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. 2012. VMTorrent: Scalable P2P Virtual Machine Streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. 289–300.

[33] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. 1998. Virtual Network Computing. *IEEE Internet Computing* 2, 1 (Jan/Feb 1998).

[34] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica Lam, and Mendel Rosenblum. 2003. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the Seventeenth USENIX Conference on System Administration*. San Diego, CA.

[35] Constantine Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica Lam, and Mendel Rosenblum. 2002. Optimizing the Migration of Virtual Computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*. Boston, MA.

[36] Constantine Sapuntzakis and Monica Lam. 2003. Virtual Appliances in the Collective: A Road to Hassle-free Computing. In *Proceedings of the Ninth Conference on Hot Topics in Operating Systems*. Lihue, HI.

[37] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *IEEE Computer* 50, 1 (January 2017).

[38] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andrés Lagar-Cavilla. 2007. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing* 11, 2 (March 2007).

[39] B. Solenthaler and R. Pajarola. 2009. Predictive-corrective incompressible SPH. *ACM Trans. Graph.* 28, 3, Article 40 (July 2009), 6 pages. DOI: http://dx.doi.org/10.1145/1531326.1531346

[40] Petter Svard, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. 2011. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In *Proceedings of the Seventh ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Newport Beach, CA.

[41] Gabriel Takacs, Maha El Choubassi, Yi Wu, and Igor Kozintsev. 2011. 3D mobile augmented reality in urban scenes. In *Proceedings of IEEE International Conference on Multimedia and Expo*.

[42] Matthew Turk and Alex Pentland. 1991. Eigenfaces for Recognition. *Journal of Cognitive Neuroscience* 3, 1 (1991), 71–86.

[43] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating System Design and Implementation, SOSP 2002*.

[44] Andrew Whitaker, Richard Cox, Marianne Shaw, and Steven Gribble. 2004. Constructing Services with Interposable Virtual Hardware. In *Proceedings of the First Conference on Symposium on Networked Systems Design and Implementation*. San Francisco, CA.

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

[46] Wikipedia. 2015. List of countries by Internet connection speeds. http://en.wikipedia.org/wiki/List_of_countries_by_Internet_connection_speeds, (2015).

[47] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. 2011. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proceedings of the Seventh ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Newport Beach, CA.

[48] Edward Zayas. 1987. Attacking the Process Migration Bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles*. Austin, TX.

[49] Weida Zhang, King Tin Lam, and Cho Li Wang. 2014. Adaptive Live VM Migration over a WAN: Modeling and Implementation. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing (CLOUD '14)*.