vTube: Efficient Streaming of Virtual Appliances Over Last-Mile Networks

Yoshihisa Abe[†], Roxana Geambasu[‡], Kaustubh Joshi[•],
H. Andrés Lagar-Cavilla[⋆], Mahadev Satyanarayanan[†]
[†]Carnegie Mellon University, [‡]Columbia University, [•]AT&T Research, [⋆]GridCentric

Abstract

Cloud-sourced virtual appliances (VAs) have been touted as powerful solutions for many software maintenance, mobility, backward compatibility, and security challenges. In this paper, we ask whether it is possible to create a VA cloud service that supports fluid, interactive user experience even over mobile networks. More specifically, we wish to support a *YouTube-like streaming service for executable content*, such as games, interactive books, research artifacts, etc. Users should be able to post, browse through, and interact with executable content swiftly and without long interruptions. Intuitively, this seems impossible; the bandwidths, latencies, and costs of last-mile networks would be prohibitive given the sheer sizes of virtual machines! Yet, we show that a set of carefully crafted, novel prefetching and streaming techniques can bring this goal surprisingly close to reality. We show that *vTube*, a VA streaming system that incorporates our techniques, supports fluid interaction even in challenging network conditions, such as 4G LTE.

1 Introduction

Viewing cloud-sourced video over 3G or 4G mobile networks is a reality experienced today by millions of smartphone and tablet users. This is an impressive achievement considering the constraints of cellular networks (shared bandwidth, high latency, high jitter) and the sustained high volume of data transmission (roughly 2 GB per hour per user for HD video according to Netflix [7]). The key to successful video streaming is *aggressive prefetching*. By maintaining a sufficient number of prefetched frames in advance of demand, the video player is able to tolerate transient degradation of network quality due to factors such as congestion or brief signal loss.

Can we achieve a similar streaming capability for cloud-sourced virtual appliances? That is the question we ask in this paper. Since their introduction [34], *virtual appliances (VAs)* have been proposed for many use cases such as software deployment and maintenance [32], desktop administration [10], and management of heterogenous systems [19]. More recently, VAs have been proposed for archiving executable content such as scientific simulation models, interactive games, and expert systems for industrial troubleshooting [35]. Our vision is to create a YouTube-like cloud service for VAs, called *vTube*. On vTube, browsing VAs and click-launching some of them should be as seamless as browsing and launching from YouTube.

Alas, streaming VAs over last-mile networks is much harder than streaming videos. For crisp, low-latency user interaction, the launched VA should execute close to the user rather than executing in the cloud and just streaming its output. As we show in Section 2.2, no existing VA transfer mechanism has adequate agility for browsing. Prior work on VA streaming [29], has not addressed such networks.

In this paper, we investigate the problem of VA streaming over limited-bandwidth, high-latency last-mile networks such as broadband, Wi-Fi, and 4G/3G. The essence of our solution is a new VA streaming mechanism that uses prefetching hints that are obtained by a fine-grained analysis of disk and memory state access traces from previous executions. This complexity is necessary because the temporal order in which parts of VA state are accessed may vary widely from execution to ex-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SOCC '13, October 01 - 03 2013, Santa Clara, CA, USA
Copyright 2013 ACM 978-1-4503-2428-1/13/10$15.00.
http://dx.doi.org/10.1145/2523616.2523636

ecution, depending on user actions, network interactions and runtime data content.

The key insight we exploit is that despite these wide variances from execution to execution and from user to user, it is possible to identify short segments of state access that once activated, are exceptionally stable across multiple executions and can thus provide high-quality predictive hints. Furthermore, these state segments compose in many different ways to form the major building blocks through which long chains of VA state access are constructed. In effect, we extract a VA-specific notion of state access locality that can be reliably predicted and then exploited for prefetching.

Our experiments and usage experience confirm that vTube supports fluid interactions even over mobile networks. For example, on an AT&T 4G/LTE network, initial buffering delay is under 40 seconds even for an interactive game VA accessing several hundreds of megabytes of state. Subsequent execution proceeds at near-native performance with fewer than 12 buffering interruptions over a 15 minute period, about half of which are sub-second, and all but one of which are sub-10-seconds in duration. Compared to VMTorrent, a prior VA streaming system [29], we reduce the unnecessary state transfers by at least a factor of 2, and result in users having to wait for VA state 17 fewer minutes in a 30 minute session. Relative to VA streaming, we make three contributions:

- We motivate and introduce a VA streaming model that incorporates from the video streaming area not just the delivery model, but (uniquely) the user-interaction model and the evaluation metrics (Section 2).
- We describe the design of a VA streaming algorithm that focuses on improving user-perceived interactivity of the application (Sections 3 and 4). Our algorithm is rooted in novel observations about VA execution patterns and represents a significant departure from existing overly-simplistic VA streaming algorithms.
- We present the first evaluation of VM streaming interactivity over mobile networks (Section 5).

2 Motivation and Context

vTube represents a new kind of VA repository whose agility requirements far surpass those of existing VA repositories such as VMware's VA Marketplace [37]. We envision a lightweight user experience that is closer to browsing web pages than managing a cloud dashboard. Section 2.1 motivates the need for agility in VA transfer using several use cases. The agility required for a satisfactory browsing experience exposes limitations of well-known VA transfer mechanisms, as discussed in Section 2.2. These limitations motivate our new video-streaming-based model for VA transfer, described in Section 2.3. We make explicit vTube's specific goals and assumptions in Sections 2.4 and 2.5.

2.1 Motivating Use Cases

Our design of vTube was guided by a series of motivating use cases, of which one has played an essential role: long-term preservation of digital artifacts. We next describe this use case, after which we broaden the scope with other examples.

Archival executable content: Today, an increasing fraction of the world's intellectual output is in the form of *executable content.* This includes apps, games, interactive media, simulation models, tutoring systems, expert systems, data visualization tools, and so on. While short- to medium-term persistence of these digital artifacts has been addressed, long-term preservation – measured in decades or centuries – is an open problem that few have considered [35]. How can such artifacts be preserved over decades and reproduced in the form originally intended as file formats, applications, and operating systems come and go?

Precise reproduction of software execution, which we call *execution fidelity,* is a complex problem in which many moving parts must all be perfectly aligned – one needs the right versions of the hardware, the operating system, the application, dynamically linked libraries, configuration and user preferences, geographic location, and so on. While multiple approaches exist (see Section 2.2), virtual machines (VMs) are considered the highest-fidelity mechanism for precisely capturing execution contexts. Consequently, some recent executable content archival efforts, such as the *Olive project* (`http://olivearchive.org`), use VMs as their containers of choice.

vTube's design is guided by such archival library services, from which it draws its requirements. We argue that the true potential of such executable content libraries will be achieved only when their content is conveniently accessible from everywhere on the Internet, including WANs and cellular networks. For example, a user should be able to use his commute time to look through fun artifacts in the library. Moreover, a key requirement is to support *browsing*, an activity that is natural in libraries and bookstores. A user should be able to search for a broad topic of interest – such as tools for image processing or strategy games from the early 2000's – and then try out a number of VAs before deciding if she is interested in one. Hence, quick startup time, along with reasonable performance during execu-

tion, over last-mile networks are crucial design requirements for vTube.

Other use cases: While digital libraries form the core motivation for vTube's use of VM streaming over last mile networks, such a facility is useful for several other current and future applications as well.

Application distribution: VAs can allow software publishers to encapsulate applications with the needed OS and libraries, and publish them for use across multiple platforms - e.g., a Windows game being played on a Linux laptop. The large size of VAs such as games makes streaming an attractive option for delivery. Users can start playing without having to wait for everything to be downloaded (which can take a long time).

Media encapsulation: As has often been argued, VAs can also serve as all-in-one containers for media requiring special handling, such as a video file in an exotic format, or a movie requiring special DRM protections. A lightweight VA streaming service would facilitate quick access to such media without requiring users to install the prerequisite codecs and libraries that are needed.

Streaming applications to local data: Applications that work on large media files could be packaged in VAs and streamed to a user's personal computer. For instance, an application rental service could stream professional video editing software encapsulated in a VA to its customers whenever the customer wishes to edit video they have captured using their home video cameras.

2.2 Design Alternatives for VA Delivery

A variety of mechanisms exist today for accessing cloud-sourced VAs over last-mile networks. We examine the available choices below.

Use thin clients: VA state transfer over last mile networks can be completely avoided by executing the VA in the cloud and using a thin client protocol such as VNC [31] or ThinC [30] for user interaction. Unfortunately, this approach does not deliver adequate responsiveness in high-latency networks such as 3G or 4G, whose RTTs routinely range between 50-300 ms. Our anecdotal experience is that playing a game such as Riven over VNC with 200 ms RTT is frustrating. Every interaction resulting in non-trivial screen changes is delayed, leading to a very sluggish application and chopped animations. In contrast, the same game is quite usable over vTube under the same conditions; the only difference from local execution is about a minute's delay for initial buffering and a few, well-signaled buffering interruptions. Moreover, in scenarios where large amounts of content come from the client, such as editing a video with a proprietary app, thin clients are inefficient. We therefore focus on VA execution close to the user.

Download entire VAs: VMware's Marketplace [37] transfers an entire VA before launching an instance. Under limited bandwidth, this leads to unacceptable launch delays. For example, a plain Ubuntu 12.04 LTS image with an 80 GB disk is a 507-MB compressed file on VMware's Marketplace. At 7.2 Mbps, which is the nation-wide average bandwidth according to Akamai [22], downloading this image takes over nine minutes. Adding applications or data to the VA increases its image size, further slowing transfer before launch. To be fair, VMware Marketplace was not intended to support quick launch, whereby users can browse and try out VAs seamlessly.

Download only parts of VAs: VAs may contain state that is rarely accessed in typical executions. For example, the disk drivers in the guest OS may support bad block remapping and flash wear-leveling because these functions are useful on real hardware. This code is not needed on a virtual disk. Selectively avoiding these rarely-used parts of a VA (possibly by examining traces of previous executions) can reduce data transfer before VA launch. Our experience, however, is that the potential win on well-constructed VAs is limited. For the VAs studied in Section 5, the compressed state can reach up to 845 MB.

Shrink VAs and leverage cached state: To further cut down on VA state transferred, researchers have proposed deduplication [33] and free-list identification [21]. With deduplication plus caching, matching partial state that was downloaded in the past from any VA can be reused. With free-list identification, one avoids the futile transfer of unallocated pages in a guest's memory image. Both of these techniques are effective, and we incorporate them in vTube. However, their effectiveness depends significantly on the target workload, and our experience indicates that they are insufficient for demanding cases.

Don't use VAs: Instead of using VAs, one could choose a seemingly lighter-weight encapsulation such as CDEpack [15], a user-level packing system that allows encapsulation of an application's dependencies (e.g., libraries) along with it. In our opinion, such user-level system is a less desirable option than VMs. First, it restricts the OS and hardware on which a consumer can run a package (e.g., CDEpack claims that its packages run on any Linux distributions from the past 5 years). Second, it requires perfect prior profiling of all accessible state. Surprisingly, such packaging is not as lightweight as one might think; packing LibreOffice Writer, an open-source document editor, with the latest version of CDEpack (2011) leads to a 117 MB package. For comparison, vTube typically streams a total of 139 MB for the cor-

responding VA. In our judgement, this modest increase in state transfer is a small price to pay for a transparent streaming system that makes the fewest assumptions about its environment and target applications, and can thus be used by the most use cases.

2.3 vTube Streaming Model

We thus see that no existing mechanism is adequate for browsing cloud-source VAs. We have therefore created a new VA streaming model that is inspired by video streaming. We describe the parallels below.

Video streaming: Cloud-sourced video services such as YouTube and Netflix offer nearly instant gratification. The user experience is simple and effortless. To start, a user just clicks a URL on the service's web page. After a brief startup delay of a few tens of seconds for initial buffering, the video begins to play. It maintains its quality (frame rate and resolution) despite unpredictable variation in network quality. While playing the content, the video player continues to stream in data. Occasionally, when network quality degrades too much for too long, the video pauses for additional buffering and then resumes. Quick launch and sustained rates are preserved independent of total video size. The prefetch buffer is primed at startup, after which prefetching occurs continuously in the background.

Video streaming is typically characterized by two metrics [12]: *buffering rate* and *buffering ratio*. Buffering rate is the number of buffering events per unit of time. Buffering ratio is the cumulative time wasted in buffering events divided by total time of play. Together, these metrics define the performance of the streaming process when content quality is preserved. Zero is the ideal value for both.

VA streaming: In vTube, we seek a user experience for VA streaming that is loosely comparable to video streaming. There is a brief delay at VA launch, while a subset of the VA's initial working set is prefetched, and then execution begins. As VA execution proceeds, it may encounter missing state that has to be demand-fetched from the cloud. Each such event stalls VA execution. Although an occasional stall may be imperceptible, an accumulation of back-to-back stalls will be perceived as a sluggish or unresponsive system by the user. vTube avoids many of these stalls by prefetching in the background VA state that will likely be accessed in the near future. Such predictions are made based on knowledge accumulated through prior executions of this VA.

Occasionally, mispredictions or lack of prefetching knowledge may lead to stalls. On other occasions, there may be accurate prefetching knowledge but it may not be available soon enough to mask all stalls under current network conditions. In such situations, we alert the user to a pause while the missing state is prefetched. During the buffering pause, the user is free to turn his attention elsewhere and to use that brief span of time productively. This may be preferable to interacting with a sluggish VM over that span of time. To quantify interactivity in vTube, we adopt the buffering ratio and rate metrics mentioned above for video streaming.

2.4 Design Goals

To frame the vTube architecture presented in the next section, we make explicit our design goals below.

Good interactivity over last-mile networks: We target non-LAN networks such as broadband connections, 4G, or 3G. Typical bandwidths today for such networks range from 7 Mbps to 20 Mbps [22, 27], while median RTTs range from 69.5 ms to 120 ms [17, 27]. While last-mile networks will continue to improve, we posit that the need for clever prefetching in such networks will persist as the amount of VA state, such as incorporated datasets, game scene data, or media content, increases over time.

Bounded VA state transfer: The total size of transferred state should be within 2x of the accessed state size. In other words, there should not be a lot of wasted data transfer. This goal is motivated by our focus on cellular networks, which often have volume-sensitive pricing. Reducing state transfer also improves scalability.

No guest changes: We shun guest OS, library, and application changes in VA creation. This allows us to support the broadest possible range of VAs, across platforms, languages and applications. We make one exception: we require guest OSes to provide indication of free memory pages to the VMM, an option that is not always enabled by default. While Windows clears pages upon reboot, making them easy to identify by vTube, Linux requires a special compile-time option, PAX security, to clear freed pages. Without this customization, we believe that efficient streaming of RAM images would be beyond reach on Linux.

Simple VA construction: The creation of VAs must not rely on fine-tuning for efficient streaming. This reflects our belief that the success of our system depends on how easy it is to create VAs for it. While we provide a few simple rough guidelines to VA producers on how to prepare their VAs (see below), we reject fine-tuning of VAs by the providers for streaming efficiency. Such fine tuning is likely to prove fragile and hard to maintain over time. Instead, the algorithms in vTube learn and adapt to the uniqueness of each VA rather than imposing rigid constraints.

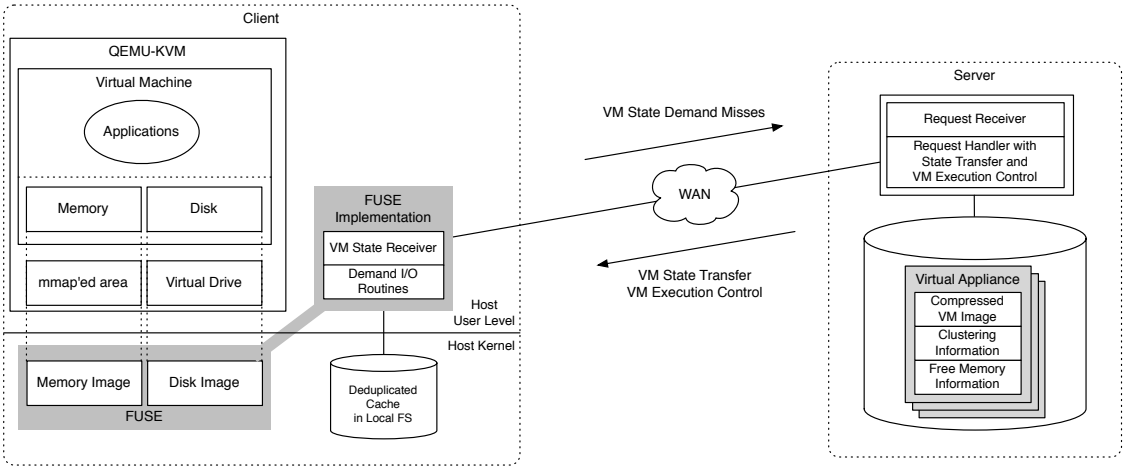


Figure 1: The vTube architecture. Two components: (1) modified `qemu-kvm` on the client side and (2) a streaming server on the VA repository side. The server collects VA access traces and orchestrates streaming decisions.

2.5 Design Assumptions

We next list the key assumptions behind vTube's design.

Task-specific VAs: We assume that producers encapsulate a focused and task-specific workflow in each VA. This single-task assumption constrains to some degree the kinds of interactions that a user will have with the VA. However, VAs can still exhibit significant variability of execution paths. For example, a complex game might have many different scenarios, each with its own scene graphics or audio data. Moreover, some VAs may use multiple processes to implement an task, and some of these processes may be multi-threaded. The asynchrony embodied in this structure adds to the challenges of prefetching VA state.

Resume rather than cold boot: When constructing the VA, the owner has a choice of including either just the disk image or both the RAM and disk images. Our experiments show that in many cases, including a RAM image with a booted OS results in less state being accessed than not including a RAM image at all. Moreover, posting a suspended VA, as opposed to one that requires booting, is more in-line with our agility goals. For these reasons, we recommend that producers always include both RAM and disk images.

Big VAs: We target VAs that range in size from hundreds of MBs to GBs of compressed state. Barring significant VA-minimization efforts by their creators, we expect VAs to grow in size over time.

Agility focus: Creating a production-quality vTube requires attention to many issues that are beyond the scope of this paper. For example, power consumption on mobile devices when streaming VAs is an important issue. Service scalability is another important issue. In this paper, we focus on the core issues of agility and interactive user experience and leave other important issues to future work.

3 The vTube Architecture

vTube leverages the `qemu-kvm` client virtualization software on Linux, and integrates it into a client-server architecture as shown in Figure 1. The client uses hash-based persistent caching and aggressive prefetching as the crucial mechanisms for agility when browsing VAs over a last-mile network. Both the memory and disk state of VAs is cached at 4KB granularity, with compression applied to network transfers. Persistent caching reduces data transfers by exploiting temporal locality of access patterns across current and previous user sessions. The hash-based cache design further reduces data transfers by reusing identical content cached from other VAs during previous executions.

Prefetching: Relative to LANs, both the high latency and the low bandwidth of last-mile networks make cache misses expensive. Prefetching helps in two ways. First, all or part of the cost of cache miss servicing is overlapped with client execution prior to the miss. Second, prefetching in bulk allows TCP windows to grow to optimal size and thus reduces the per-byte transfer cost. Unfortunately, it is well known that prefetching is a double-edged sword; acting on incorrect prefetching hints can clog a network with junk, thereby hurting overall performance. Erroneous prefetching can also exacerbate the problem of buffer bloat [13].

In light of these fears of prefetching, the key observation from our work can be stated as follows: *Despite all the variability and non-determinism of VM execution, prefetching hints of sufficient accuracy and robustness can be extracted and applied to make VA browsing over last-mile networks interactive.* This observation does not

suggest that entire disk and memory access traces of multiple uses of a VA will be identical. Rather, it suggests that short stretches of the traces can be dynamically predicted during VM execution with sufficient accuracy for prefetching. Section 4 presents the details of the algorithms that enable this prediction. Intuitively, it is the single-task nature of VAs that makes prefetching possible. Since a VA enables its users to only perform a limited number of specific activities, examining previous execution traces for the VA and abstracting from them can lead to accurate and robust prefetching hints.

Client structure: A simple client, with lightly modified `qemu-kvm` and hash-based persistent cache, accesses VA state from a cloud-based streaming server. Our modifications to `qemu-kvm` enable the client to demand-page the memory snapshot of a VA from the server, rather than having to fetch it completely at instance creation. The disk image is mounted via the FUSE file system, and its I/O is redirected to user-level code that handles memory and disk cache misses, as well as prefetching directives from the server. When the client is paused for buffering, the user is notified via a client GUI.

Server structure: The algorithmic sophistication and prefetching control are embodied in the server. It maintains three data structures for each VA. First, it maintains compressed and deduplicated VM state. Memory and disk images are split into 4KB *chunks* and stored in a compressed form. Associated with each chunk is its SHA1 value, which is used for deduplication within or across VAs. Second, the server maintains coarse access patterns to control prefetching. Using trace analysis, we organize chunks into coarse-grained *clusters*, and derive, store, and use access patterns for them. As shown in Section 4, this removes a lot of uncertainty associated with finer-grained access patterns. Third, the server maintains a list of free memory chunks (a technique used in previous work such as [9]), which is obtained by inspecting the VA for zeroed-out pages or with the help of a tiny guest kernel module. It submits this list to the client at the start of a session, through which it avoids requesting the contents of those free chunks unnecessarily.

VM execution flow: VA use proceeds as follows:
Step 1: When the user initiates a session, the client fetches VA metadata (virtualized hardware configurations etc.) and a list of free memory chunks from the server. The server receives from the client a hash list of its cached chunks, if any, from previous sessions.

Step 2: The server determines what initial state the client must buffer before it can begin VM execution based on the prefetching algorithm described in Section 4. It compares the initial chunk set with those already cached on the client, and transfers chunks it does not have.

Step 3: Once VM execution has started, the client issues demand fetches to the server if (a) it has not retrieved the content of a chunk being accessed by the VM, (b) the chunk is not present in the local cache, and (c) the chunk is not on the free list.

Step 4: Each demand fetch from the client triggers a predictive analysis algorithm on the server. When it returns the requested chunk to the client, the server also determines what other chunks should be predictively pushed to the client, and whether the VM must be paused in the process.

Step 5: After the user has finished using the VA, the client uploads a timestamped trace of all first-time accesses to memory and disk chunks. The server processes the trace to remove the effects of poor network conditions by eliminating time spent waiting for data to arrive from the network, and uses the resulting "zero-latency idealized trace" to update its models for predictive streaming.

Prefetch control: The biggest challenge in vTube is dynamic control of prefetching so that it helps as much as possible, but never hurts. In practice, this translates into two key decisions: (1) choosing what state to predictively stream to minimize application performance hiccups during execution; and (2) choosing when and for how long to pause a VM for buffering. These decisions must factor in several criteria such as the VA's historical behavior, the VA's current behavior, current network bandwidth, and the nonlinear behavior of human users.

Tolerating variability and uncertainty is a prerequisite. Even though each VA is single-task, it is still an ensemble of software that may include multiple processes, threads, and code paths interacting in non-deterministic ways. Moreover, different "paths" users take in interacting with the same application might lead to widely different access traces. Wide variation in networking conditions further adds to variability and uncertainty. We describe our approach to addressing these challenges in the next section.

4 VA Prefetching and Streaming

The algorithms vTube uses for extracting prefetching hints, for deciding (1) what VA state is to be streamed and (2) when the VM should be paused, are inherently *dynamic* in nature. They do not just rely on historical traces of how a VA has been used in the past, but more critically, on what its VM is currently doing, and what network bandwidth is available for streaming state. This dynamic decision-making is in stark contrast to previous approaches such as VMTorrent [29] that rely on *statically* constructing an "average" trace of historical VA behavior and using it to inform the order in which disk chunks must be prefetched.

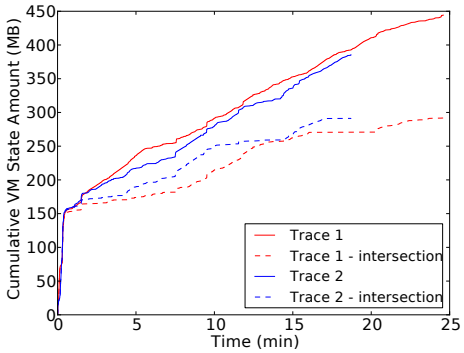


Figure 2: Amounts of VM state accessed by a pair of traces for the game Riven over time (solid lines). Dashed lines show the amount accessed in one trace that was also accessed in the other trace.

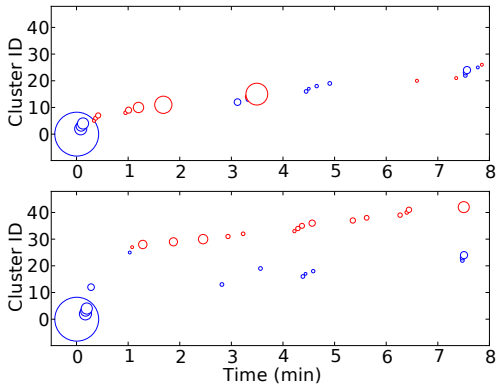


Figure 3: Examples of VM state access behavior for Riven. Each circle represents a VM state cluster, with its radius reflecting the cluster size (ranging from 0.5 to 131 MB). Those clusters in blue are common between the two traces. For clarity, clusters smaller than 0.5 MB are omitted.

4.1 The Case for Dynamic Prefetching

The following key insights demonstrate why static approaches are not sufficient, and why dynamic algorithms are needed to construct robust prefetching hints.

1) VA traces contain many similarities, but pairs of traces can have significant differences in the VA state they access. Figure 2 shows examples of state accesses by two executions of the 2D adventure game Riven, in which the user started a new game. While the dashed lines show a substantial amount of state common between the traces, 24-34% of the state accessed by each execution is unique.

This insight suggests that a static approach that decides what to prefetch before VM execution begins must either fetch too much state or too little. It would fetch too much unnecessary state if, like VMTorrent, it chooses to prefetch the union of all traces; it will fetch too little if it prefetches only their intersection. Dynamic decision making is needed if we are to meet our goals of bounding VA state transfer and minimizing waits induced due to buffering.

2) VM state access occurs in bursts that may appear at unpredictable times, or in different orders across different traces, but is predictable once they begin. Figure 3 illustrates VM state access behavior extracted from the same two traces of Riven used for Figure 2. It shows that common clusters (variable-size groups of state chunks, as described in Section 4.2) can appear at different times in the two executions, partially maintaining the same ordering, while interleaved with unique clusters. Such behavior is easily explained using the well known concept of a *working set*. Whenever the user initiates a new activity, e.g., moving into a new game area or invoking a print operation on a document, a lot of new state – binaries, libraries, and data – is often needed, resulting in a flurry of activity.

This insight suggests that while it may not be possible to accurately predict what the user is going to do next, and thus the next burst, the first few accesses in a burst may be enough to adequately predict everything else that follows. To leverage this insight, vTube needs two mechanisms: 1) a way to identify clusters of memory accesses that correspond to distinct user-activity-driven bursts, and 2) a way to distinguish between clusters whose occurrence is truly unpredictable and clusters that are likely to follow other clusters, e.g., an add-on library that is often loaded after the main binary of an application.

3) Poor performance occurs when memory access bursts overwhelm the network. Bursty access patterns have yet another consequence; even if it is possible to predict the next chunks that will be accessed by a burst based on the first few accesses, the network must still be fast enough to deliver these chunks before they are needed by the VM. The traces in Figure 2 show several examples of bursts (the near-vertical jumps in the solid lines), in which the instantaneous demand for VA state would exceed any last-mile bandwidth available today. If none of the required state has been prefetched by the client before the burst occurs, the network will not be able to prefetch state at the needed rate, leading to expensive demand misses that the VMM must service while the VM is stalled. In our experience, multiple demand misses in quick succession lead to serious choppiness of execution and affect usability. Hence, vTube includes techniques to dynamically detect such situations before they occur based on available bandwidth, and pause the VM in anticipation.

4.2 The Prefetching Algorithm

The general outline of vTube's prefetching algorithm follows our insights. a) The algorithm derives units of

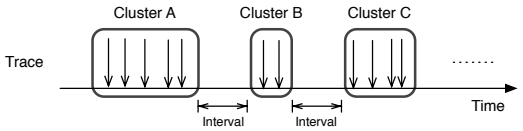


Figure 4: Clustering in individual traces. Clusters are formed by grouping chunks accessed one after another within an interval.

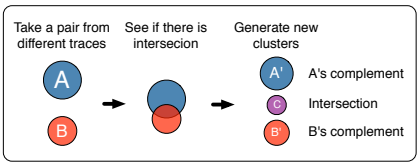


Figure 5: Generating clusters out of two traces. Overlapping clusters from distinct traces are split into disjoint clusters.

VA state called *clusters* that are likely to be accessed in their own bursts. b) When the VM accesses state belonging to a certain cluster, the algorithm determines a set of clusters to be transferred together. c) Finally, the algorithm uses current network bandwidth to decide how this transfer is overlapped with VM execution, minimizing the user wait time. The first step is performed offline for each VA, using its collected execution traces. The other two steps are done online, as the VM executes.

4.2.1 Clustering VM State (offline)

Clusters are variable-size units of VM state transfer, and consist of 4KB disk or memory chunks that are accessed together. Figures 4 and 5 illustrate the process of generating clusters from traces. We construct clusters for each individual trace by grouping chunks accessed one after another within a *clustering interval* (which is 2 seconds in our prototype, selected empirically). Next, we merge common clusters across different traces, replacing each pair of overlapping clusters with three disjoint clusters: the intersection and the complements of the original clusters. Finally, the newly generated clusters are checked again with the clustering interval, and split further into separate clusters as appropriate.

The rationale behind clustering is two-fold. As we describe next, we exploit properties associated with the clusters for making efficient VM state transfer decisions. Clustering also makes VM state access analysis coarse enough to be computationally affordable for virtual appliances with well-focused target workloads, such as those described in Section 5, while still allowing for fairly fine-grained state transfer. Without clustering, for example, tens of thousands of 4KB chunks are typically accessed in one session, and analyzing billions of resulting pairs would easily require substantial computational resources, especially more than tens of GB of memory.

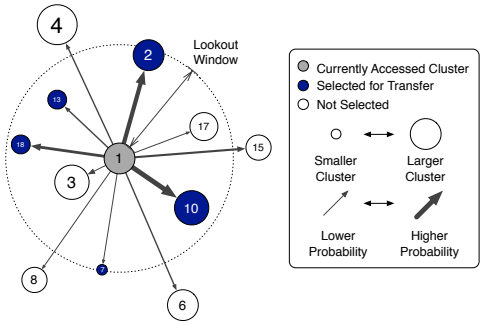


Figure 6: Cluster selection for buffering and streaming. Cluster 1 contains a chunk being accessed, and blue clusters are selected for transfer together with the cluster.

4.2.2 Cluster Predictions (online)

When the VM demand fetches a chunk contained in a particular cluster, we identify a set of clusters deemed necessary in the near future and send it together with the accessed cluster. This selection is based on relations between clusters reflecting VM access demands and transitions observed in prior traces. Specifically, each relation involves two measures: an *interval* and a *probability*. The *interval* defines how soon a cluster is expected to be accessed after another. We use a conservative metric of the minimal interval observed in the traces. The *probability* represents how likely this succession of accesses occurs. For each ordered cluster pair, these two measures are obtained from the traces.

The process of identifying the cluster set is illustrated in Figure 6, in which Cluster 1 is being accessed. First, from a practical perspective of not trying to estimate the infinite future, we have a time frame called *lookout window*, and consider those clusters whose interval falls within this window from the currently accessed cluster. Then, we select a subset of these clusters based on their probability and size. To do so, we exploit a key intuition that accesses to small clusters often appear random and are hard to predict, while the cost of retrieving them is low. Taking this into account, we retrieve smaller clusters more opportunistically and larger clusters more conservatively. We perform this by reflecting the distribution of cluster sizes in a threshold applied to probability. Let us consider two clusters, with identifiers X and Y. Upon the demand miss of a chunk in X, we select Y for retrieval if the following condition is satisfied:

$$P(Y|X) > Percentile(Size(Y))$$

$P(Y|X)$ is the probability of Y following X. $Percentile(Size(Y))$ is the percentile of the size of Y; in other words, it represents the the fraction of all chunks that belong to a cluster whose size is smaller than that of Y. Figure 7 shows this percentile for varied size of Y, for one of the VAs used in our evaluation,

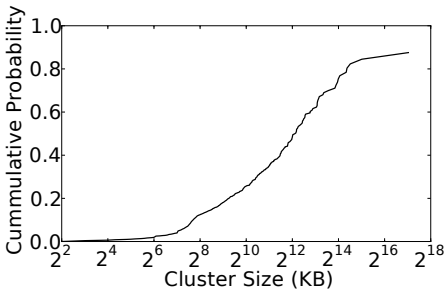


Figure 7: Example of cluster size distribution (for a VA with the game Riven). The x-axis is on a log scale.

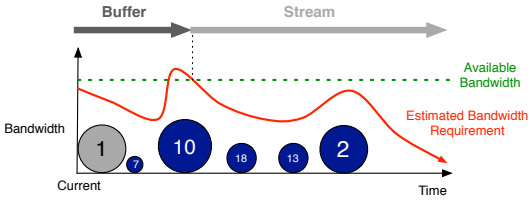


Figure 8: Buffering/streaming decision of selected clusters. Circles represent the set of clusters selected for transfer.

Riven. In this example, if the size of Y is 4 MB, $P(Y|X)$ needs to be greater than approximately 0.5 for it to be retrieved with X.

4.2.3 Cluster Buffering and Streaming (online)

Finally, with the set of clusters for retrieval decided as described in the previous section, we create an estimate of VM state access demands using cluster intervals. Based on this estimate, we decide part of the set that is buffered, while streaming the rest in the background of VM execution. Figure 8 summarizes how we derive access demands. We order the clusters in the set according to their intervals to the currently accessed cluster (Cluster 1). Then, we calculate a point after which currently available bandwidth can transfer the remaining clusters without expecting to incur demand misses. We buffer the clusters up to this point, while suspending VM execution. Once the available bandwidth appears to be sufficient for ensuring retrieval of chunks before their accesses, we switch to streaming the remaining chunks.

An overall flow of the algorithm is illustrated in Figure 9. Upon each state demand miss, the process described above is triggered, resulting in a buffering period followed by streaming of VM state. Note that those demand misses are the first read access to a certain chunk, and not subsequent reads or any writes, which do not require VM state transfer. In particular, we keep track of writes at the disk block granularity and cache the written contents, avoiding the retrieval of a chunk even when partial writes occur to it.

Additionally, we use an optimization in which the client notifies the server of a currently accessed chunk

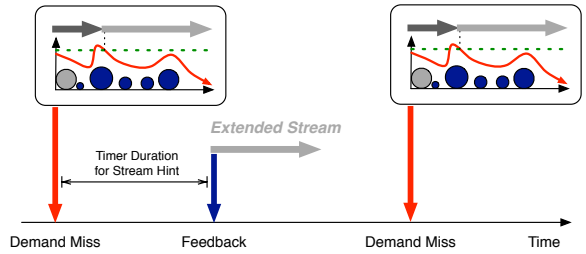


Figure 9: Overall flow of the vTube algorithm with streaming feedback from the client.

periodically. For this chunk, the server triggers streaming of a set of clusters derived in the same manner, with the exception that VM execution is not suspended for buffering. This helps extending streaming when there is no demand miss for an extended period of time to trigger further VM state transfer.

5 Evaluation

We focus our evaluation on three critical questions, which evaluate the core contributions of our paper:

Q1: *How interactive is vTube in real-use scenarios, such as real applications, networks, and users*? (Section 5.2)

Q2: *How accurate is vTube's prefetching*? (Section 5.3)

Q3: *How does vTube's performance compare to that of other VA streaming systems*? (Section 5.4)

We next describe our methodology to answer these questions.

5.1 Experimental Methodology

We evaluate vTube in multiple network and workload settings. All experiments were conducted with a server running in Pittsburgh PA, and a client laptop in various locations with different network conditions. The server was equipped with an 8-core Intel Core i7 at 3.40GHz and 32GB RAM, while the client laptop had a dual-core Intel Core 2 Duo CPU at 2.40GHz and 4GB RAM. All measurements were performed with no cached VA state on the client side. Also, we set the lookout window size and timer duration for stream hints to 16 and 2 minutes, respectively.

Experimental networks: Figure 10(a) shows the various network environments we tested. For controlled experiments, we used Linktropy [1], a dedicated network emulator, to construct the two emulated networks labeled "14.4 Mbps" and "7.2 Mbps." We chose the characteristics of these networks based on reports on US nation-wide average bandwidth from Akamai [22] and

Label	Type	Bandwidth	RTT
7.2 Mbps	emulated	7.2 Mbps	120 ms
14.4 Mbps	emulated	14.4 Mbps	60 ms
3G	real	8-12 Mbps	52-113 ms
4G	real	1-29 Mbps	96 ms
Taiwan (good)	real	7-25 Mbps	220 ms
Taiwan (fair)	real	4-9 Mbps	220 ms
Coffee Shop	real	5-7 Mbps	14-175 ms

(a) Experimental Networks

VA	Description	OS	Total Size (compressed)	Download Time (7.2 / 14.4 Mbps)
Mplayer	video payback	Linux	4.6 GB	87 min / 44 min
Avidemux	video edit	Linux	5.1 GB	97 min / 49 min
Arcanum	game	Win XP	6.9 GB	131 min / 66 min
Riven	game	Win 7	8.0 GB	152 min / 76 min
HoMM4	game	Win 7	5.6 GB	106 min / 53 min
Selenium	browsing	Linux	4.4 GB	83 min / 42 min
Make	compilation	Linux	4.6 GB	88 min / 44 min

(b) Experimental VAs

Figure 10: Summary of our network set-ups (a) and virtual appliances (b).

bandwidth/latency studies over mobile networks [17, 27]. In addition to controlled experiments, we also report results over real networks, including: (1) domestic AT&T 3G and 4G LTE (labeled "3G" and "4G" in the figure), (2) overseas broadband connections through two public Wi-Fi's (labeled "Taiwan"), and (3) a public Wi-Fi at a local coffee shop.

Virtual appliances: We constructed seven VAs highlighting different use cases, workloads, and evaluation metrics. All VAs had 1GB memory and a 10-20GB disk. The VAs and their metrics are described below, while some of their properties are shown in Figure 10(b):

1. *Mplayer*: `mplayer` on Linux plays a 5-minute, 84-MB AVI video file. Evaluate frames per second (FPS).

2. *Avidemux*: `Avidemux`, a Linux video editing application, batch-converts eight MP4 files, locally supplied and 698 MB in total, to AVI format. Evaluate conversion times.

3. *Arcanum*: Contains Arcanum [2], a 2D role-playing game on Windows XP. Report manual experience.

4. *Riven*: Contains Riven [5], a 2D adventure game on Windows 7. Report manual experience.

5. *HoMM4*: Contains Heroes of Might and Magic IV [3], a 2D strategy game on Windows 7. Report manual experience.

6. *Selenium*: `Firefox`, along with an automation Selenium script [6], browses through an HTML copy of Python documentation. Evaluate the speed of page traversal over time.

7. *Make*: Compiles `Apache` 2.4.4 source code, locally supplied, on Linux. Evaluate compilation time.

Figure 10(b) shows the VAs' compressed sizes and the times for full download over our emulated networks. All VAs are large and while some might be optimizable, the three games Riven, Arcanum, and HoMM4, have inherently large installation package sizes: 1.6, 1.1, and 0.9 GB, respectively. The full download of a VA would thus take 40 minutes to 2.5 hours, an aspect that vTube dramatically improves.

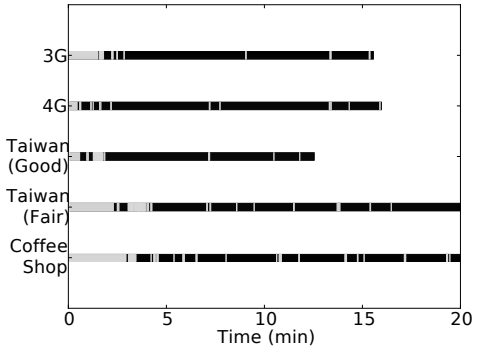


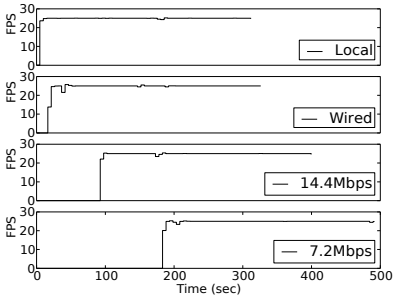
Figure 11: Visualization of sample Riven runs on real networks. Note that each workload is a manual game play, and thus is not directly comparable to another.

VA histories: To create histories for these VAs, we manually collected 10 traces for all except for Make and Riven, for which we collected 6 and 16 traces, respectively. We intentionally introduced variability in the workloads. For example, Mplayer traces include various play modes, such as skipping, pausing, and speed-up in addition to regular playback. Avidemux traces involve converting video and audio to different formats. Arcanum and Riven include game plays from the beginning and from different save data files. HoMM4 traces consist of tutorial plays with different difficulty levels. Selenium traces have different behaviors such as looking at different content, searching, or clicking on links. Make, a particularly low-variance workload, requires only few traces.

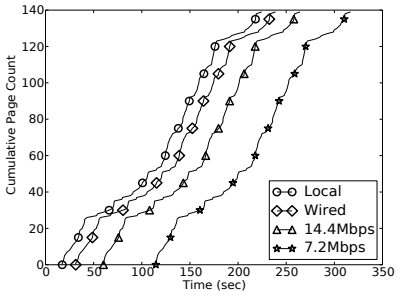
5.2 Interactivity Evaluation (Q1)

To evaluate vTube's interactivity, we first show a few sample runs, which help build up intuition about its behavior in practice, after which we evaluate application performance more systematically.

Sample runs: Figure 11 shows sample runs of the Riven VA over various real networks. For each run, black continuous lines indicate periods of uninterrupted execution, while gray interruptions indicate when VA execution is stalled either due to explicit buffering periods or



(a) Mplayer Frames Per Second (FPS)



(b) Selenium Browsing Rate

VA	Network	Launch Time (sec)	Workload Completion Time (sec)	Overhead on Local (no net)
Avidemux	Wired	6	75	54.6%
	14.4 Mbps	26	67	39.5%
	7.2 Mbps	52	65	33.8%
Make	Wired	7	420	2.8%
	14.4 Mbps	17	407	-0.4%
	7.2 Mbps	32	411	0.4%

(c) Avidemux, Make Runtimes

Figure 12: Evaluation of application-level metrics. The figures compare execution over vTube to execution out of a local VA without vTube ("Local") and execution over a direct-wire link with pure demand fetching and no prefetch ("Wired").

memory demand fetches. Most demand-fetch stalls are short, approximately equal to the RTT and on their own, and typically not visible on the graph (nor to the user, in our experience). Durations of buffering periods vary widely. All executions follow a similar pattern; after a relatively long initial buffering period (25-175 seconds, depending on bandwidth), vTube occasionally pauses executions for periods up to 20 seconds. For example, the 4G run has one pause longer than 10 sec, five 1-10 sec pauses, and five sub-second pauses.

Application performance evaluation: Figure 12 shows application-level performance metrics for the subset of VAs that support scripted workloads and metrics as described in Section 5.1. For each VA, we report two timings: (1) *launch time*, measured from VM resume until the workload can start and (2) *workload completion time* excluding the launch time. We report averages over three runs for Avidemux/Make and individual runs for Mplayer/Selenium. We compare the results of using vTube over the two emulated networks with two baselines: a) execution from a locally supplied VA without vTube, and b) execution using a pure demand-fetch system over a fast direct-wire connection.

Figure 12(a) shows the evolution of Mplayer's FPS over time as a video is played. The video's native FPS is 25.0, and local VM execution preserves it. In the demand-fetch case, video playback starts with a delay of 16.3 seconds. On vTube, video playback starts with a delay of 92.5 and 183.3 seconds to launch for the 14.4 Mbps and 7.2 Mbps cases, respectively. However, once the playback has launched, vTube maintains its FPS close to native, with only one or two short interruptions depending on the network. Figure 12(b) shows a similar effect for Selenium: after 60.2 and 114.6-second launch times for the 14.4Mbps and 7.2Mbps networks, respectively, the automated browsing script sustains a progress rate that is close to the ideal.

Finally, Figure 12(c) shows the results for Avidemux and Make, and the increase in completion time compared to local VM execution (overhead). After some tens of seconds of initial buffering, vTube's workload-runtime overheads remain tiny for the Apache compilation (under 0.4%) and reasonable for the Avidemux workload (under 40%) across both networks. The difference in overheads is driven by differences in vTube's prefetching accuracy, which in turn is driven by the difference in the degree of variance inherent in the two workloads. Section 5.3 expands on this.

Placed in the context of VA sizes and download statistics from Figure 10(b), vTube's interactivity improvements are dramatic. Instead of having to wait over 44 minutes to fetch a VA over a 14.4 Mbps network – a bandwidth twice the US national average [22] – a vTube user waits between some tens of seconds to a couple of minutes to launch the VA, after which his interactions are close to ideal, except for a few explicit but short interruptions. This is, in our opinion, a significant achievement. However, a key question remains: is this good enough for end-users, have we achieved a "usable" system yet? While we lack a formal answer to this question, we address it anecdotally next.

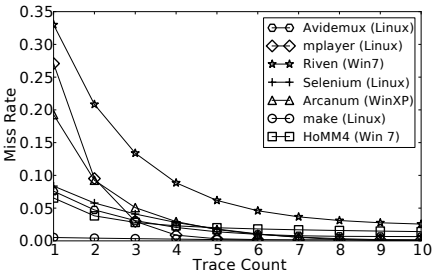
Anecdotal user experience: To gain some context about interactivity, we informally tried out vTube with four users: two co-authors and two Ph.D. students outside of the project. All users played our two game VAs – Riven and Arcanum – over two public Wi-Fi networks in Taiwan (see Figure 10(a)), and were asked to rate their interactions as $\{bad — fair — good — very\ good\}$.

Three out of the four users (the two co-authors and one non-author) rated their executions with both games as either *good* or *very good*. In particular, the non-author rated his executions of both games as *very good* and did not mind the buffering periods on the "Taiwan (good)" network. The fourth user (the other non-author) ran over the "Taiwan (fair)" network and rated his executions for Arcanum and Riven as *good* and *fair*, respectively. For visualization, traces labeled "Taiwan (Good)" and "Taiwan (Fair)" in Figure 11 represent the executions rated *very good* and *fair*, respectively, by the two non-authors.

Summary: Our application evaluation and anecdotal ex-

VA	Workload Duration (sec)	Accessed State (MB)	Fetch Ratio (fetched / accessed state)	Buffering Ratio (buf time / total time)	Buffering Rate (# buf events / minute)	Miss Rate (# misses / # accesses)
Mplayer	500	316 (2.4%)	1.03	0.33	0.24	0.20%
Avidemux	127	103 (0.8%)	1.33	0.39	1.88	0.82%
Arcanum	920	87 (0.8%)	1.51	0.04	0.41	0.40%
Riven	1223	379 (1.8%)	1.49	0.21	0.46	0.70%
HoMM4	926	256 (1.2%)	1.41	0.15	0.37	1.96%
Selenium	332	155 (1.2%)	1.45	0.31	0.96	0.12%
Make	451	76 (0.6%)	1.01	0.06	0.35	0.66%

(a) Whole-System Accuracy.



(b) Miss Rates with Number of Traces.

Figure 13: Accuracy evaluation. Results are over a 7.2Mbps/120ms emulated network. In (a), the numbers are the average of three runs. State amounts are those that require transfer, and reported as raw values (uncompressed, non-deduplicated).

perience suggest that vTube provides interactive access to cloud-sourced VAs over WAN and mobile networks. Acknowledging that a rigorous user study is required to prove this aspect, we next turn to evaluating prefetch accuracy.

5.3 Prefetching Accuracy Evaluation (Q2)

To measure the accuracy of vTube's prefetching predictions, we rely on several standard metrics: fetch ratio, buffering ratio, buffering rate, and miss rate. Together, these metrics reflect how close our system's access predictions are to reality. Figure 13(a) evaluates these metrics against our seven VAs. The graph proves two points. First, independent of vTube, the amount of accessed state in any trace is tiny compared to the total VA state (no greater than 2.4% across all VAs), which justifies the need for accurate prefetching. Second, with vTube, the four metrics indicate very good accuracy for low-variance workloads and reasonable accuracy for higher-variance ones. We analyze each metric in turn.

Fetch ratio: The fetch ratio – the ratio of the amount of state fetched to the amount of state accessed – quantifies how much state is fetched needlessly. While some over-fetching is unavoidable, too much can affect interactivity and network load. Figure 13(a) shows that fetch ratio in vTube remains under 2.0, a value that we deem reasonable. For fixed workloads, such as Mplayer and Make, fetch ratios are particularly low (under 3% overhead). For higher-variance workloads, such as games, the amount of over-fetch is highest (up to 51%). Overall, vTube meets our goal of bounded VA state transfer (Section 2.4).

Buffering ratio and rate: Buffering ratio and rate quantify the total time the user wastes waiting for buffering and the frequency of buffering interruptions, respectively. Lower numbers are better. With the exception of Avidemux, the user wastes little time, between 6 and 33%, during his interaction with the VM. The Avidemux workload has a particularly short running time, which justifies the higher buffering ratio (39%). As a general rule, the longer the user's interaction with the VM, the lower the overall buffering overhead will be, as the investment in an initial, long buffering period amortizes over time. Buffering events are also relatively rare across all VAs, as shown by the "Buffering Rate" column: less than one buffering period per minute except for Avidemux.

Miss rates: The *miss rate* – the ratio of the number of chunks demand-fetched to the number of chunks accessed – remains under 2% for all traces. While the absolute number of misses is non-negligible, our experience suggests that individual misses do not significantly degrade user experience, unless they occur one after another. Our algorithm makes state transfer decisions based on previous VM executions, and hence miss rates are heavily influenced by the coverage of accessed chunks in the available trace set. We examine this influence next.

Impact of trace sets on miss rates: Figure 13(b) shows how the miss rate changes with different subsets of traces included in the training set. We compute the miss rate using results of our emulated network experiments. With 10 traces, the coverage is high enough to achieve low miss rates for our VAs: under 0.65% for all VAs other than Riven and HoMM4, for which the miss rates are 2.6% and 1.4%, respectively. With fewer traces, miss rates vary dramatically depending on the VA. For Make, which has a fixed workload, the miss rate is particularly low even with 6 traces (0.12%). (Note that these numbers are statically computed from traces, whereas the miss rates in Figure 13(b) (a) are affected by access timings.)

Summary: These results indicate that, given sufficient trace sets, vTube's prefetching is accurate enough to limit wasteful transfers and curtail excessive user delays due to buffering or missed predictions. We next show that existing techniques do not even come close to these achievements.

VA	System	Workload Duration (sec)	Accessed State (MB)	Fetched State (MB)	Total Buffering Time (sec)	Miss Rate (# misses / # accesses)
Arcanum	vTube	920	87	131	39	0.40%
	VMTorrent	924	84	230	12	0.89%
Riven	vTube	1223	379	566	255	0.70%
	VMTorrent	1222	222	1051	980	0.02%
HoMM4	vTube	926	256	360	136	1.96%
	VMTorrent	927	262	384	54	1.31%

Figure 14: Comparison to VMTorrent. The numbers are the average of three runs. State amounts are those that require transfer, and reported as raw values (uncompressed, non-deduplicated).

5.4 Comparison to VMTorrent (Q3)

We compare against VMTorrent, the closest related work, by implementing its prefetching logic in vTube while retaining our largely orthogonal optimizations related to compression, deduplication, and streaming. For each VA, the new system creates a static streaming plan that includes all chunks that have been accessed in at least one previous trace ordered by the average time at which they were first accessed in those traces (see Section 2.2.3 of [29]).

Figure 14 shows the comparison of the two systems for the Arcanum, Riven, and HoMM4 VAs executed over our 7.2Mbps/120ms network. In the comparisons, we keep the total wall-clock time that the user interacts with each system roughly constant: 15 min for Arcanum and HoMM4, and 20 min for Riven (column "Workload Duration"). Overall, the results demonstrate that VMTorrent is much less precise in its prefetching than vTube for both VAs. For example, in Arcanum, while the amount of state accessed by each execution is similar (≈85MB), VMTorrent fetches 1.8 times more state as vTube, resulting in a fetch ratio of 2.76 for VMTorrent vs. 1.51 for vTube. For Riven, the difference is even more dramatic: 4.73 for VMtorrent vs. 1.49 for vTube. HoMM4 has a smaller degree of workload variety in its traces, resulting in comparable fetch ratios of 1.41 for vTube vs. 1.47 for VMTorrent.

These overheads have two effects: (1) they strain the network and (2) they force users to wait, thus hampering the fluidity of VA access. For example, as shown in the column "Total Buffering Time" for Riven, VMTorrent forces the user to wait for over 16 minutes out of a 20-minute session (80%). In contrast, vTube limits user waiting to only about 4 minutes out of 20 minutes (or 21%). An effect of this aspect is visible in the figure: because the user spent a lot more time playing Riven rather than waiting for buffering, the amount of accessed state by vTube (379MB) is much higher than the amount of state accessed by VMTorrent (222MB). Thus, vTube's precise prefetching decreases wasted time and (arguably) increases the user's productivity.

5.5 Summary

We have shown that vTube provides interactive access to cloud-sourced VAs even over some of today's most challenging networks, such as 3G, 4G, and WAN. Its prefetching mechanism is more precise than existing systems. The key assumption in vTube is the availability of suitable trace sets. For best user experience, a vTube deployment could refrain from streaming new VAs to low-bandwidth users until trace sets with sufficient coverage have been gathered from high-bandwidth users.

6 Related Work

We have already covered some related work in Section 2.2. We first summarize our contributions vis-a-vis that work, after which we discuss other related work. Overall, vTube is unique in two ways. First, it focuses on a new class of agile interactive workloads, namely browsing a large collection of VAs. "Browsing" here includes rapid launch of a VA instance at the whim of the user, some period of interaction with that instance, followed by abandonment of that instance and change of focus to another VA. This is completely different from today's workloads, where VA instances are long-lived and launching one on impulse is rare. Second, vTube aspires to support this new style of browsing over cellular wireless networks that have challenging bandwidth, latency and jitter characteristics. We are not aware of any other work that addresses these two major challenges.

Closest in spirit to our work is VMTorrent [29], which uses P2P streaming techniques to deliver VAs on demand. It also uses profile-based prefetching and block prioritization, along with a number of other well-known I/O mechanisms, to speed VA launch. Our focus on last-mile networks implies that a P2P strategy is not useful: it can only help if the bottlenecks are on individual WAN paths from the cloud and replica sites to the edge. A related approach that uses a hash-based content distribution network is described by Peng et al [28], but it is also inadequate for last-mile networks.

More broadly, we have benefited from the rich body of work on efficient state transfer and rapid launch of VMs that has been published over the last decade. Our work has been influenced by the Collective [10, 32, 33, 34], Moka5 [4], Snowflock [20], Kaleidoscope [9], Xen live migration [11], and the Internet Suspend/Resume system [18, 36]. From these systems, we have leveraged techniques such as demand-driven incremental VM state transfer, a FUSE-based implementation strategy, hash-based persistent caching of VM state, and transfer of live execution state. vTube extends these mechanisms with prefetching and streaming to achieve agile access to a cloud-based VA repository.

We have also benefited from the lessons learned about caching and prefetching in the context of distributed file systems for bandwidth-challenged environments [14, 16, 25, 26], high performance I/O [8], and multiprocessor systems [24]. Previous work in the area of file systems has also investigated prefetching based on access pattern analysis [14, 23], which could potentially apply to and augment our scheme in a complementary manner. Besides actual analysis procedures, our work also differs from such work in the scope of application, considering the notion of VM execution together with prefetching. vTube's browsing model was inspired by the Olive project's vision of archiving executable content [35].

7 Conclusion

Cloud-sourced executable content has grown in significance as the benefits of VM encapsulation have become more apparent. Unfortunately, the large size of VAs has constrained their use in environments where network quality is constrained. The mobile edge of the Internet is the most obvious example of such an environment. Accessing large VAs over such networks, especially in the highly agile manner required for browsing, appears to be a fool's dream.

In this paper, we have shown that a set of carefully-crafted, novel prefetching and streaming techniques can bring this dream close to reality. We have constructed vTube, a VA repository that supports fluid interactions. On vTube, a user can browse and try out VAs seamlessly and without much wait, just like he would browse and try out for videos on YouTube. The essence of our techniques boils down to a key observation: that despite all uncertainty and variability inherent in VA executions, prior runs of the same VA bear sufficient predictive power to allow for efficient buffering and streaming of VA state. With controlled experiments and personal experience, we show that vTube can achieve high interactivity even over challenging networking conditions such as 3G, and in doing so it far outperforms prior systems.

8 Acknowledgements

We thank the Olive team for inspiring this work, and especially Benjamin Gilbert and Jan Harkes for their valuable technical discussions on Olive implementation. We thank Da-Yoon Chung for his help with trace collection, Athula Balachandran for her guidance on video streaming performance metrics, and Kiryong Ha and Zhuo Chen for their feedback on the usability of our system. We thank our shepherd Ken Yocum and the anonymous reviewers for their valuable comments and suggestions for improving the presentation of the paper.

This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0833882 and IIS-1065336, by an Intel Science and Technology Center grant, by DARPA Contract No. FA8650-11-C-7190, and by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), a federally funded research and development center. This material has been approved for public release and unlimited distribution (DM-0000276). Additional support was provided by IBM, Google, and Bosch. 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.

References

[1] Apposite Technologies :: Linktropy 5500 WAN Emulator. `http://www.apposite-tech.com/products/5500.html`.

[2] Arcanum product page (archived). `http://www.gog.com/game/arcanum_of_steamworks_and_magick_obscura`.

[3] Heroes of Might and Magic IV. `http://www.gog.com/game/heroes_of_might_and_magic_4_complete`.

[4] MokaFive Home Page. `http://www.mokafive.com`.

[5] Riven: the sequel to Myst. `http://www.gog.com/gamecard/riven_the_sequel_to_myst`.

[6] Selenium - Web Browser Automation. `http://docs.seleniumhq.org`.

[7] Netflix Streaming Bandwidth Levels/Requirements. `http://watchingnetflix.com/home/2012/08/streaming-bandwidth-requirement/`, Aug. 2012.

[8] A. Brown, T. Mowry, and O. Krieger. Compiler-based I/O Prefetching for Out-of-Core Applications. *ACM Transactions on Computer Systems*, 19(2), May 2001.

[9] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: Cloud Micro-elasticity via VM State Coloring. In *Proceedings of the Sixth*

Conference on Computer Systems (EuroSys 2011), Salzburg, Austria, April 2011.

[10] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, May 2005.

[11] 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.

[12] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, ON, 2011.

[13] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *Communications of the ACM*, 55(1), January 2012.

[14] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, Boston, MA, 1994.

[15] P. J. Guo and D. Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, June 2011.

[16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[17] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

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

[19] D. Kreutz and A. Charao. FlexVAPs: a system for managing virtual appliances in heterogeneous virtualized environments. In *Latin American Network Operations and Management Symposium (LANOMS)*, 2009.

[20] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of EuroSys 2009*, Nuremberg, Germany, March 2009.

[21] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[22] F. Lardinois. Akamai: Average U.S. Internet Speed Up 28% YoY, Now At 7.4 Mbps, But South Korea, Japan And Hong Kong Still Far Ahead. *TechCrunch*, April 23 2013.

[23] Z. Li, Z. Chen, and Y. Zhou. Mining block correlations to improve storage performance. *Trans. Storage*, 1(2):213–245, May 2005.

[24] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions in Computer Systems*, 16(1), February 1998.

[25] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, December 1995.

[26] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, October 1995.

[27] PCMag. Fastest mobile networks 2013. http://www.pcmag.com/article2/0,2817,2420333,00.asp, 2013.

[28] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proceedings of Infocom 2012*, Orlando, FL, March 2012.

[29] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. VMTorrent: Scalable P2P Virtual Machine Streaming. In *Proceedings of CoNEXT12*, Nice, France, December 2012.

[30] Ricardo A. Baratto and Jason Nieh and Leo Kim. THINC: A Remote Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

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

[32] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[33] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec 2002.

[34] C. Sapuntzakis and M. Lam. Virtual Appliances in the Collective: A Road to Hassle-free Computing. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, May 2003.

[35] M. Satyanarayanan, V. Bala, G. S. Clair, and E. Linke. Collaborating with Executable Content Across Space and Time. In *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom2011)*, Orlando, FL, October 2011.

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

[37] S. Shankland. VMware opens virtual-appliance marketplace. *CNET News*, November 7 2006.