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, delays, and costs of mobile 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 interactions even over challenging networking 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 [6]). 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 [32], virtual appliances (VAs) have been proposed for many use cases such as software deployment and maintenance [30], desktop administration [9], and management of heterogeneous systems [18]. More recently, VAs have been proposed for archiving executable content such as scientific simulation models, interactive games, and expert systems for industrial troubleshooting [33]. 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 [27], has not addressed such networks.

In this paper, we investigate the problem of VA streaming over low-bandwidth, high-latency last-mile networks such as broadband, 4G, and 3G. The essence of our solution is a new VA streaming mechanism that uses prefetching hints that are obtained by machine learning on 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 execution, 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 spatial 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 ATT 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 [27], 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).

We next begin by motivating our work.

2. MOTIVATION AND CONTEXT

vTube represents a new kind of cloud service whose agility requirements far surpass those of existing VA repositories such as VMware’s VA Marketplace [35]. We envision a lightweight user experience that is closer to browsing web pages than managing a cloud dashboard. A VA is the ideal granularity at which to browse executable content because it is a pre-installed and pre-configured environment that includes all necessary software within its encapsulating virtual machine (VM). This includes the operating system and dynamically-linked libraries, as well as input data such as contents of interactive books and game scene data.

The agility of vTube allows the unique strengths of VAs to be leveraged in new ways, as suggested by the scenarios of Section 2.1. 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 Usage Model

We envision vTube to be a service that stores and serves executable content such as games, interactive books, pre-configured tool chains, simulation models for research, and archival software of historical interest. We illustrate our vision with two example scenarios.

Scenario 1: FunGames, a gaming company, encapsulates each of its games in a VA and publishes them all on its web site. It used to advertise by posting demo videos on YouTube. Sales have improved significantly since the adoption of vTube. Now, users can get live experience of trial game VAs. A user can search for, browse through, launch and interact with many VAs before finding one that interests him. This is the point at which a sale is made. The fluidity of interaction in browsing VAs, by trying out one briefly, then trying out another, then a third and so on is crucial to the improvement in sales. Experience has shown that slow VA launch, sluggish response, or choppy interaction all shorten user engagement and lower sales.

Scenario 2: Following a recently recognized need for software preservation, the Library of Congress seeks to expand its archival efforts from static content to interactive artifacts of cultural significance. These include old games, document- or photo-editing tools, interactive books, documents, unique datasets encapsulated with now-obsolete applications to interpret their content, and many other types of executable content. For long-term preservation and use of these artifacts, the library creates a VA for each in its vTube repository. This is a national resource that is open to the general public, allowing citizens to browse through and interact with artifacts of interest.

2.2 Design Alternatives

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 [29] or ThinC [28] 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 120-300 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 comes 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 [35] trans-
fers an entire VA before launching an instance. On mobile networks, 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 [21], downloading this image takes over nine minutes. Adding applications or data to the VA increases the file size of the VA, further slowing transfer before launch. To be fair, VMware Marketplace was not intended to support browsing, whereby users can try out many VAs seamlessly as they “shop around.”

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 [31] and free-list identification [20]. 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.

Avoid VMs: A VA is heavyweight because it includes an entire guest OS. The benefit is that a VA cleanly encapsulates the messy complexity of its internal state, thereby eliminating many of the dependencies that a target host must satisfy. Lighter-weight encapsulations are available if one is willing to compromise on ubiquity of execution. For example, a user-level packaging system such as CDEpack [14] allows some flexibility of targets by encapsulating user-level state such as libraries, application code, and data.

Such packaging has two well-understood limitations. 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 corresponding VA.

In our judgement, this modest increase in state transfer is a small price to pay for the greater confidence that the VA can be successfully launched on many hosts well into the future. Longevity and ubiquity are important considerations in archival preservation of executable content.

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 [11]: 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 condi-
tions. 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 [21, 25], while median RTTs range from 69.5 ms to 120 ms [16, 25]. 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.

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 kvm-qemu 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 are 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 [12].

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 VA 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 traces of multiple executions of a VA will be identical. Rather, it suggests that short stretches of the traces can be dynamically predicted during VA execution with sufficient accuracy for prefetching. Section 4 presents the details of the algorithms that make this possible. 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 executions of the VA and abstracting from them can lead to accurate and robust prefetching hints.

**Client structure:** A simple client, with a lightly modified *qemu-kvm* VMM and a hash-based persistent cache, accesses VA state from a cloud-based streaming server. Our modifications to *kvm-qemu* 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 4-KB chunks and stored in 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 those. 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, which is obtained by inspecting the VA for zeroed-out pages. The server submits this list to the client at the start of a session.

**VA execution** proceeds as follows:

**Step 1:** When the user initiates VA execution, the client fetches VA metadata (VCPU state, device state, etc.) and a list of free memory chunks from the server. These chunks can be recreated on the client and need not be transferred.

**Step 2:** The server determines what initial state the client must buffer before it can begin VA execution based on the prefetching algorithm described in Section 4. It compares the initial chunk set with those already cached on the client from previous executions of other VAs (using SHA1 hashes), and transfers chunks the client does not have.

**Step 3:** Once VA execution has begun, the client issues demand-fetches to the server if (a) it does not have a chunk that was accessed by the VA, (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. Once it has returned the requested chunk to the client, the server also determines what other chunks should be predictively pushed to the client, and whether the VA 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 each memory and disk chunk. The server modifies 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.

**Controlling prefetching:** 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 VA for buffering during execution. 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 a VA 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 the VA is doing right now, and what network bandwidth is available for fetching state. This dynamic decision-making is in stark contrast to previous approaches such as VMTorrent [27] 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 fetched.

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 an illustrative but typical example of state accessed by two executions of the 2D role-playing game Arcanum [2] in which the user starts playing the game at different levels. While the dashed lines show a substantial amount of common state between the traces, 30-50% of the state accessed by each execution is unique. This insight suggests that a static approach that decides what to prefetch before VA 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 bounded VA state transfer and minimizing waits induced due to buffering.

2) **VA state access occurs in bursts that may appear at unpredictable times, or in different orders across different histories, but are predictable once they begin.**

Figure 3 shows two execution histories for Arcanum in which each set of memory accesses that occurs within a second interval and is identical across both traces is assigned a unique “cluster ID”, and is marked as a black dot. Two second bursts unique to a trace are shown as grey dots. The plots demonstrate that while there are a large number of common identical bursts in the two histories, these bursts appear at different
times and different orders in each history. Such behavior is easily explained using the well known concept of a working set. Whenever the VA 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 – are 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 versus 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 VA 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 VA. The histories 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 VA 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 the occur based on available bandwidth, and pause the VA in anticipation.

4.2 The Prefetching Algorithm

The general outline of vTube’s prefetching algorithm follows our insights: a) The algorithm derives units of VA state called clusters that are likely to be accessed in their own bursts. b) When the VA accesses state belonging to a certain cluster, the algorithm then 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 virtual appliance, 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 groups consisting of those 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 an individual trace by grouping chunks that are accessed one after another within a clustering interval (2 seconds in our prototype, chosen experimentally), as shown in Figure 4. Next, we merge common clusters that appear in multiple traces, replacing each pair of overlapping clusters with three newly generated disjoint ones: the intersection and the complements of the original clusters. If any of the newly created clusters contain chunks that are separated by an interval longer than the clustering interval, they are further split into independent clusters.

Besides its role in delineating distinct user activities, clustering also makes it computationally feasible to analyze VA traces by allowing efficient analysis at a course granularity while still allowing for transfers at a fairly fine-grained granularity when needed. 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 more than tens of GB of RAM and substantial computational resources.

4.2.2 Cluster Predictions (online)

When the VM faults in state contained in a particular cluster, we identify a set of clusters deemed necessary in the near future and send it together with the faulted-in 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 faulted in. 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 faulted-in cluster. Then, we select a subset of these clusters based on the clusters’ probability and size. To do so, we exploit a key intuition: accesses to small clusters are hard to predict, while the cost of retrieving them is less. 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 the threshold over probability. Specifically, upon the fault-in of cluster X, we select given cluster Y for retrieval if the following condition is satisfied:

\[
P(Y|X) \geq P(\text{ClusterSize} \geq \text{size}(Y))
\]

where \(P(Y|X)\) is the probability of Y following X and \(P(\text{ClusterSize} \geq \text{size}(Y))\) is the probability of any given chunk belonging to a cluster at least as large as Y.

### 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 7 summarizes how we derive access demands. We order the clusters in the set according to their interval 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 fault-ins. We buffer the clusters up to this point, while suspending VM execution. Once the available bandwidth appears to be sufficient to ensure retrieval of chunks before their accesses, we switch to streaming the remaining chunks.

An overall flow of the algorithm is illustrated in Figure 8. Upon each state fault-in, the process described above is triggered, resulting in a buffering period followed by streaming of VM state. Additionally, we use an optimization in which the client notifies the server of a currently accessed chunk periodically, for which 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 fault-in 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 the 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 running 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.

**Experimental Networks.** Figure 9(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 [21] and bandwidth/latency studies over mobile networks [16, 25]. In addition to controlled experiments, we also report results over real networks,
Virtual Appliances. We constructed six VAs highlighting different use cases, workloads, and evaluation metrics. All VAs had 1GB memory and 10-20GB disk. The VAs and their metrics are described below, while some of their properties are shown in Figure 9(b):

1. Mplayer: mp4/layer on Linux plays a 5-minute, 84-MB AVI video file. Evaluate frames per second (FPS).

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


5. Selenium: Firefox, along with an automation Selenium script [5], randomly browses through an HTML copy of Python docs. Evaluate pages rendered over time.

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

Figure 9(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 two games Riven and Arcanum, have inherently large installation package sizes: 1.1 and 1.6 GB, respectively. The full download of a VA would thus take 40 minutes to 2.5 hours, an aspect that vTube dramatically improves.

VA Histories. To create histories for these VAs, we manually collected 10 traces for all except Make, for which we only collected 6 traces. 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. 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 10 shows sample runs of the Riven VA over various real networks. For each run, thick continuous lines indicate periods of uninterrupted execution, while blank interruptions indicate when VA execution is stalled either due to memory demand fetches or explicit buffering periods. Demand-fetch stalls are short, approximately equal to the RTT and on their own, are 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 (25-275 seconds, depending on bandwidth), vTube occasionally pauses executions for periods between 1-20 seconds. For example, the 4G run has one pause longer than 10 sec, 5 1-10 sec pauses, and 5 sub-second pauses.

Application Performance Evaluation. Figure 11 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 VA resumption 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 in a local VA without vTube, and b) execution using a pure demand-fetch system over a fast direct-wire connection.
Figure 11(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 VA execution preserves it. In the demand-fetch case, video playback starts with a delay of 16.8 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 11(b) shows a similar effect for Selenium: after 61 and 114-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 11(c) shows the results for Avidemux and Make, and the increase in completion time compared to local VA 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 37%) 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 9(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 [21] — 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 independent from the project; the two non-authors knew nothing about the project, its goals, or its mechanisms. All users played our two game VAs – Riven and Arcanum – over two public wi-fi networks in Taiwan (see Figure 9(a)), and were asked to rate their interactions as {	extquoteleft}bad', 	extquoteleft fair', 	extquoteleft good', or 	extquoteleft very good'.

Three out of the four users (the two co-authors and one non-author) ran over the “Taiwan (good)” network. All three rated their executions with both games as either 	extquoteleft good' or 	extquoteleft very good'. In particular, the non-author rated his executions of both games as 	extquoteleft very good' and said “he didn’t mind the buffering periods, [and that] execution seemed as good as what [he] would expect in local execution.” The fourth user (the other non-author) ran over the “Taiwan (fair)” network and rated his executions for Arcanum and Riven as 	extquoteleft good' and 	extquoteleft fair', respectively. For Riven, he said that the “initial buffering was fine, but [that] he didn’t have experience using a VM that gets suspended while in use, and that made it 	extquoteleft fair' overall.” For Arcanum, he said that he “didn’t really notice any sluggishness.” For visualization, traces labeled “Taiwan (Good)” and “Taiwan (Fair)” in Figure 10 represent the executions rated 	extquoteleft very good' and 	extquoteleft fair', respectively, by the two non-authors.

**Summary.** Our application evaluation and anecdotal experience 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 12(a) evaluates these metrics against our six 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 (up to 2.37% 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 12(a) shows that fetch ratio in vTube remains under 2x, a value that we deem reasonable. For fixed workloads, such as Mplayer and Make, fetch ratios are particularly low (under 1.03x, or 3% overhead). For higher-variance workloads, such as games, the amount of over-fetch is highest (up to 81%). 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 a VA. The Avidemux workload has a particularly short running time, which justifies the higher buffering ratio (49%). As a general rule, the longer the user’s interaction with a VA, 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.

**Miss Rates.** The miss rate – the ratio of the number of chunks demand-fetched to the number of chunks accessed – remains under 0.028 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 VM 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.

![Figure 12: Accuracy Evaluation. Results are over 7.2-Mbps emulated network. In (a), lower numbers are better.](image)

Figure 12: Accuracy Evaluation. Results are over 7.2-Mbps emulated network. In (a), lower numbers are better.

---

**Buffering Ratio and Rate**

<table>
<thead>
<tr>
<th>VA</th>
<th>System</th>
<th>Workload Duration (sec)</th>
<th>Accessed State (MB)</th>
<th>Fetched State (MB)</th>
<th>Total Buffering Time (sec)</th>
<th>Miss Rate (# misses / # accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mplayer</td>
<td>vTube</td>
<td>907</td>
<td>85</td>
<td>117</td>
<td>79</td>
<td>0.51%</td>
</tr>
<tr>
<td>Avidemux</td>
<td>VMThread</td>
<td>908</td>
<td>86</td>
<td>230</td>
<td>79</td>
<td>1.42%</td>
</tr>
<tr>
<td>Riven</td>
<td>vTube</td>
<td>1872</td>
<td>407</td>
<td>610</td>
<td>390</td>
<td>0.96%</td>
</tr>
</tbody>
</table>

![Figure 13: Comparison to VMThread (Q3)](image)

Figure 13: Comparison to VMThread. For fetched state, buffering time, miss rate, lower values are better.

We examine this influence next.

**Impact of Trace Sets on Miss Rates.** Figure 12(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 ratios for our VAs: under 0.65% for all VAs other than Riven, for which the miss rate is 1.3%. 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%).

**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.

### 5.4 Comparison to VMThread (Q3)

We compare against VMThread, 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 [27]).

Figure 13 shows the comparison of the two systems for the Arcanum and Riven VAs executed over a 4.8Mbps/200ms 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 30 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 almost twice as much state as vTube, resulting in a fetch ratio of 2.67x for VMTorrent vs. 1.36x for vTube. For Riven, the difference is even more dramatic: 4.22x for VMTorrent vs. 1.45x for vTube!

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 almost 24 minutes out of a 30 minute session (79%). In contrast, vTube limits user waiting to only 6.5 minutes out of 31 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 (407MB) is much higher than the amount of state accessed by VMTorrent (249MB). 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. It relies on a prefetching mechanism that is far 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.

Closet in spirit to our work is VMTorrent [27], 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 [26], 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 [9, 30, 31, 32], Moka5 [3], Snowflock [19], Kaleidoscope [8], Xen live migration [10], and the Internet Suspend/Resume system [17, 34]. 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 [13, 15, 23, 24], high performance I/O [7], and multiprocessor systems [22]. vTube’s browsing model was inspired by the Olive project’s vision of archiving executable content [33].

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 use experience, we show that vTube can achieve high interactivity even over challenging networking conditions such as 4G LTE, and in doing so it far outperforms prior systems.
8. REFERENCES