Towards Synchronization of Live Virtual Machines among Mobile Devices

Jeffrey Bickford
AT&T Security Research Center
New York, NY, USA
jbickford@att.com

Ramón Cáceres
AT&T Labs – Research
Florham Park, NJ, USA
ramon@research.att.com

ABSTRACT

The mobile computing experience would improve if users could switch seamlessly from one device to another, with both data and computation state preserved across the switch without apparent delay. This paper proposes *VMsync*, a system for synchronizing the state of live virtual machines (VMs) among mobile devices. VMsync seeks to incrementally transfer changes in an active VM on one device to standby VMs in other devices, so as to maintain a consistent VM image and minimize switching latency. However, constraints of the mobile environment make these goals difficult to achieve and raise many research questions. We present our preliminary design for VMsync and a feasibility study aimed at determining how much data would need to be transferred under different mobile workloads and synchronization policies. For example, through experiments with a Xen VM running Android and playing a YouTube video, we show that sending dirty memory pages transfers 3 times more data than sending only the bytes that actually changed in those pages. Overall, we conclude that VMsync is a feasible approach deserving of further research.

1. INTRODUCTION

People increasingly rely on mobile devices in their everyday lives, often multiple devices such as a smartphone and a tablet. The utility of these devices would improve if users could switch seamlessly from one device to another, in particular if they could continue using applications on the second device exactly where they left off on the first device, with both data and computation state preserved across the switch without apparent delay. For example, a user who starts watching a video on a smartphone may want to continue watching the video on the larger display provided by a tablet.

A limited form of such device switching is currently available through per-application data synchronization. For example, Apple's iCloud service synchronizes changes to calendars, address books, and a few other supported applications. Some other applications provide their own synchronization facilities. However, this approach requires separate and often specific support to be built into each application. A general solution that works for all applications would scale better to the rapidly growing set of mobile applications, which already number in the hundreds of thousands [9].

This paper proposes *VMsync*, a system for synchronizing the state of live virtual machines (VMs) among mobile devices. System-level VMs have been widely proposed to improve the security, manageability, and other aspects of mobile computing [3, 5, 6, 8, 12]. In the context of device switching, VMs encapsulate both data and computation state for a complete operating system and all its applications. Therefore, synchronizing VM state between mobile devices automatically synchronizes all application state.

VMsync seeks to incrementally transfer changes in an active VM on one device to standby VMs on other devices, so as to maintain a consistent VM image and minimize switching latency. This way, when a user switches between devices, there should only be a small amount of data left to transfer before the VMs on both devices become fully consistent, and the switch can be made quickly enough that the user will not notice any delay.

However, constraints of the mobile environment make these goals difficult to achieve and raise many research questions. For example, intermittent connectivity may delay dissemination of changes. Similarly, bandwidth, processing, storage, and energy limitations introduce challenging tradeoffs between simple schemes that transfer complete memory pages or disk blocks, and more sophisticated schemes that transfer only the portions of those pages and blocks that have actually changed.

This paper presents our early efforts towards a complete VMsync design, implementation, and evaluation. After discussing the most relevant prior work, we describe our preliminary design. The VMsync architecture involves a daemon running outside the guest VM on the active device. This daemon inspects the memory and file-system state of the VM and sends recent changes to a server in the cloud. The server then forwards the changes to standby devices.

Finding appropriate policies for how to represent changes and when to send them are central research issues in this work. For example, should the device send whole dirty pages and blocks, or only the changed portions? Should the device send periodic checkpoints or wait for contextual hints? Should the server forward changes as it receives them, or post-process them to reduce the amount of data sent to standby devices?

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HotMobile'13, February 26–27, 2013, Jekyll Island, Georgia, USA.
Copyright 2013 ACM 978-1-4503-1421-3/13/02 ...$15.00.

We also present a feasibility study aimed at determining how much data would need to be transferred to maintain a consistent VM image across devices, under different mobile workloads and synchronization policies. We use the size of these data transfers as a rough proxy of various costs incurred during VM synchronization: bandwidth, latency, and energy. We believe that bandwidth, latency, and energy costs will be to some degree proportional to data transfer size. We plan to explicitly measure these different costs in a future full-fledged implementation of VMsync.

In the current work, we measure changes to the memory and file-system images of a Xen virtual machine running the Android operating system. We drive the experiments with popular Android applications, and report how many bytes would be transferred for a range of policies. For example, when playing a YouTube video, we show that sending dirty memory pages transfers 3 times more data than sending only the bytes that actually changed in those pages. These measurement results constitute a modest research contribution, as we are not aware of previous measurements of VM-image changes using such mobile-specific workloads.

Overall, we conclude that VMsync is a feasible approach deserving of further research. Our measurements show that there are significant opportunities to save costs by choosing certain synchronization policies over others. At the same time, many questions remain to be answered before we know which policies are most appropriate in which situations.

2. RELATED WORK

System-level virtualization of mobile devices has been proposed and implemented by both researchers [5, 8, 3] and commercial entities [6, 12]. We agree with their conclusions that virtualization improves the security and manageability of mobile computing, and add our insight that virtualization would also enable seamless switching between devices.

There are several established techniques for migrating VM state between hardware hosts, but we find them unsuitable for our purposes. For example, *live migration* [4] transfers a VM image while the VM continues to run in the originating host, only suspending the VM for an imperceptible period while control is finally switched to the receiving host. However, live migration transfers the complete VM memory image each time, an operation that generally involves hundreds of megabytes if not gigabytes of data, which would be prohibitive over a slow wireless link. In addition, live migration assumes a high-speed shared storage medium between the hosts involved, so that file-system state need not be transferred at migration time. Mobile devices do not enjoy such high-speed shared storage.

A recent refinement on live migration uses delta compression to reduce the amount of data transferred in the later stages of migration [10]. However, it still sends the complete contents of memory at least once before beginning to apply differencing techniques to the pages that have changed in the course of the migration. It also still assumes a high-speed shared storage medium.

Work on *opportunistic replay* [2] proposes an approach for decreasing the amount of data transferred during VM migration in low-bandwidth environments. This approach logs user-input events (e.g., keyboard presses and mouse clicks) during VM execution, then transmits and replays this log on a second identically-configured VM to produce nearly the same VM state. Because only user events are logged, events

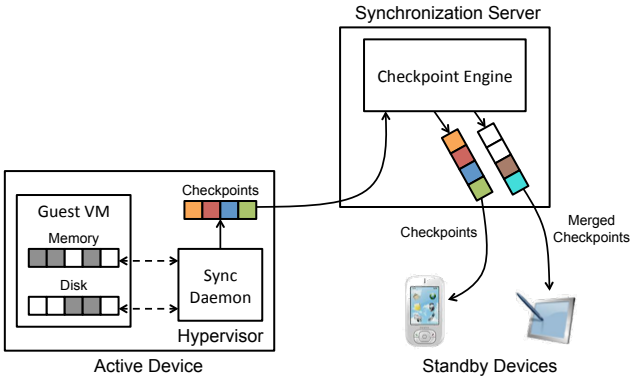

Figure 1: VMsync Architecture

triggered by additional hardware devices or background network connections may produce unmonitored state changes. These changes remaining after replay are also transferred and applied, resulting in a final identical VM. Though opportunistic replay is a potential mechanism for synchronizing multiple VMs in the VMsync scenario, there are many policy decisions left to explore to create a mobile VM synchronization solution that is imperceptible to the user. Further studies on opportunistic replay using today's network-intensive mobile applications and operating systems would also be needed to determine how well the approach would work in the VMsync scenario.

The Kimberley system [13] introduced the concept of pre-distributing a base VM image to relevant hosts, then sending only the differences from the base when wanting to move a VM from one host to another. It calls for suspending the VM on the originating host, calculating differences, transferring them, and applying them, before resuming the VM at the destination host. We adopt the idea of pre-distributing a base VM, but go further in pursuing incremental synchronization of live VM state among multiple devices without perceptibly suspending the VM.

3. PRELIMINARY DESIGN

The goal of VMsync is to maintain a consistent VM image across multiple devices while minimizing the time it takes for users to switch between devices. We consider this time the *switch penalty*. The simplest approach would perform a live migration of the guest VM at the time the user would like to switch devices, though this would incur a data transfer cost on the order of the VM size. For example, live migration of a Xen VM with 800 MB of memory can transfer as much as 960 MB (1.2x) [4].

In the case of wireless networks, VM state changes could occur faster than bandwidth allows, leading to higher switch penalties. Even with delta compression [10], live migration transfers on the order of the VM memory size during its initial stages, limiting its use in networks confined by data caps and bandwidth limitations. Therefore, along with reducing the switch penalty, we must also minimize the total amount of data transferred between devices. In VMsync we propose an incremental synchronization method to migrate a VM across mobile devices that attempts to minimize both the switch penalty and the data transferred.

Figure 1 represents the architecture of VMsync, a system made up of multiple host devices with virtualization support

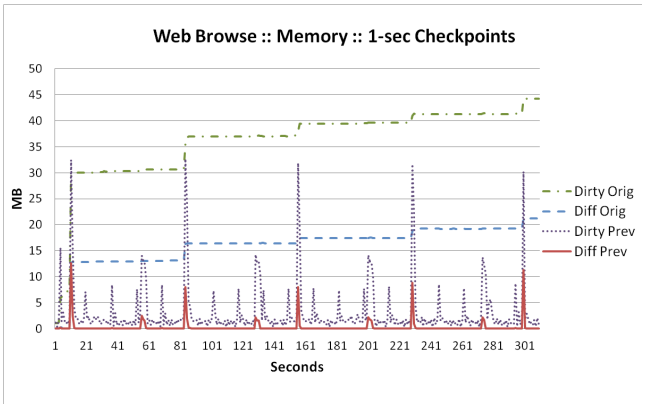

Figure 2: Memory contents change significantly with each new web page loaded.

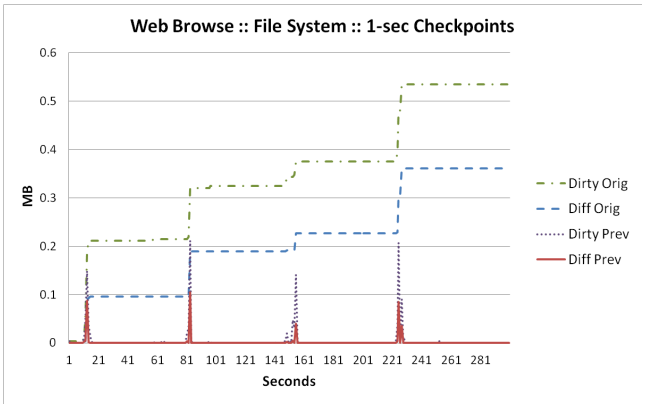

Figure 3: File-system contents change when the web browser caches data.

and a resource-rich server in the cloud, used as a synchronization point between devices. Devices registered with a VMsync instance are provisioned with a single base guest VM, like in Kimberley [13], containing a typical mobile device operating system such as Android. The hypervisor of each device handles syncing operations through a privileged daemon which monitors the guest VM state.

In our initial design of VMsync, only one *active VM* will be running at any given point in time to ensure that VMs do not diverge. This *active device* will propagate changes of both memory and file system state to the synchronization server over the period of time in which the device is active. We consider this process a *checkpoint*. Every other VM, considered *standby VMs*, will be *paused* and periodically updated via the synchronization server if the device is online. Devices which are not connected to the network or have not been synchronized will be considered to be in a *stale state* and must be synchronized before a user can switch to that device. The longer a device is offline or not updated, due to limited bandwidth or policy decisions, the higher the switch penalty.

The synchronization daemon running on the end device, which monitors the guest VM for changes, must be designed in such a way that balances the tradeoff between data transferred and computational overhead. For example, a naive low-computation approach would be similar to live migration, e.g., simply propagate every memory page and disk block that has been changed since the last checkpoint. During our feasibility study in the subsequent section, we show that this method would propagate a large amount of data that has not actually changed. This raises the question of how to efficiently synchronize only the bytes that have changed since the previous checkpoint.

Since mobile devices contain an increasing amount of file-system storage, 64 GB or more, it would be feasible to maintain a snapshot of the previous memory checkpoint on disk. Today's mobile devices, such as smart phones and tablets, contain a limited amount of memory, typically maxing out at around 2 GB. This would allow a byte by byte comparison or an on-device differencing algorithm to identify the bytes that have changed since the last checkpoint. Alternatively, a network-based differencing algorithm such as *rysnc* could be used, though this may require additional network and computational overhead. Due to the large size of the file system, a copy-on-write disk image or customized block driver could be used to efficiently monitor file-system changes and synchronize file-system state. Previous work on opportunistic replay [2] and delta compression [10] could also be adapted for use during VMsync's checkpointing step.

Though the above update mechanisms are not novel, VMsync introduces many policy questions that can only be answered with a thorough design, implementation, and evaluation of the system. For example, in order to minimize the switch penalty, should the active device propagate changes periodically over time, use specific operating system events to infer the best time to propagate changes, or use a mix of these two policies? An event, such as putting the phone to sleep via the hardware power button, may be a good indicator that the device will no longer be used for some time and the user could potentially switch to another device. On the other hand, if the user switches devices before this event, the state change from the last checkpoint may be large and will thus increase the switch penalty. In this case, a periodic checkpoint would have helped. We can also use other factors such as location, nearby device presence, battery life, CPU utilization, network bandwidth, bandwidth caps, etc., as triggers for state propagation.

There are also various policy decisions that must be made on the synchronization server. For example, when should devices be updated with the latest state information? One policy may decide that only devices connected to a network with sufficient bandwidth can receive changes. For devices that have been offline for some time, the server can merge multiple checkpoints from the active VM to minimize the amount of data transferred. In some cases it may also be possible to bypass the synchronization server by using local wireless links such as Wi-Fi Direct, Bluetooth, or NFC to migrate changes directly between devices.

Finally, the variety of hardware configurations in mobile devices introduces challenges when migrating a VM from one device to another. We have not fully addressed this device heterogeneity issue, but we note that it is common to other mobility schemes based on VM migration [7, 11]. On the positive side, modern mobile operating systems such as Android and Windows Phone 8 are designed for extensibility, thus providing support for many types of hardware built by different manufacturers. Therefore, it seems feasible in the future to extend these operating systems to detect and adapt to hardware changes at runtime.

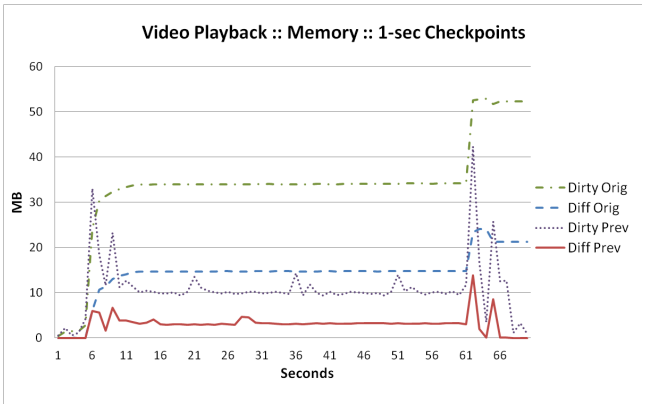

Figure 4: Most changes to memory occur during the initial loading of the video, with some final changes when the application closes.

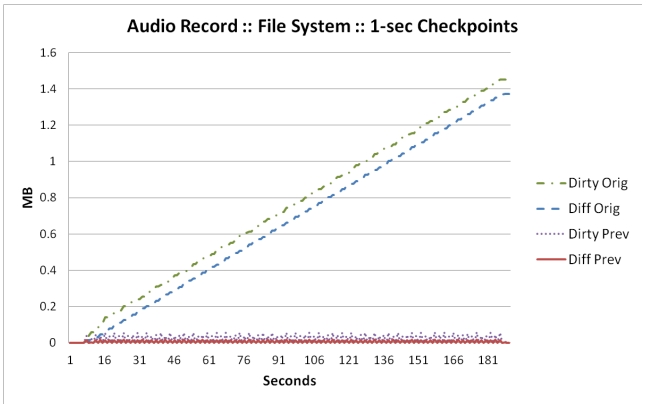

Figure 5: File-system contents change continuously during audio recording.

4. FEASIBILITY STUDY

To measure the feasibility of VMsync, we analyzed changes to memory and the file system under various mobile workloads. Our goal was to determine how much data would be required to maintain a consistent VM state across multiple devices. As workloads, we chose applications that we believe are representative of current mobile phone use: web browsing, video playback, audio playback, and audio recording.

We performed our study on the Android platform. Our VM is an Android-x86 4.0.4 (Ice Cream Sandwich) [1] guest domain running above the Xen 4.1.1 hypervisor. The Android VM uses an Android Open Source Project (AOSP) 3.0.8 kernel compiled for x86 and with Xen paravirtualization support enabled. The guest domain is allocated with 512 MB of memory, 1 virtual CPU, a 512 MB read-only system image that contains the Android software stack and is pre-distributed to all devices, and a 512 MB read/write data partition that is monitored for changes during experiments. The host machine is a desktop-class machine with a quad-core Intel Core i7 860 processor executing at 2.8 GHZ and 12 GB of RAM. Though this host system is in no way representative of a mobile device, we are measuring changes to memory and file-system usage under mobile workloads, not computation overheads or other effects influenced by host capacity.

Our synchronization daemon executes outside of the guest domain within Domain0, and uses xenctrl APIs to map the guest domain's memory prior to starting a workload. At the beginning of the workload, we pause the VM and save a copy of both memory and the data partition. This represents the original state of the VM prior to running a workload. Over the course of a workload, we analyze the memory and file system states across various *checkpoints*. For each checkpoint, we pause the VM and compare the current memory and file system state with the original copy we saved in the beginning of the workload. We also save the state of each previous checkpoint to measure changes over specified intervals of time. This procedure simulates an implementation of VMsync, where a device would periodically sync the current changes with all devices. We can thus understand the volume of data changed across an entire workload for different VMsync policies.

Measuring Checkpoint Sizes

In each of Figures 2–5, we show four curves, each corresponding to a different checkpointing policy. The top curve, labeled *Dirty Orig*, represents the case where each memory page or disk block that was changed from the beginning of the workload is synced. The second curve, labeled *Diff Orig*, only counts the bytes changed from the beginning of the workload, simulating a differencing syncing mechanism. The third and bottom curves, *Dirty Prev* and *Diff Prev*, follow similarly, but are measured with respect to the system state of the previous checkpoint. For example, Figures 2 and 3 shows the number of megabytes that would be transferred while executing a checkpoint every second during a web browsing session using the default Android browser.

Our web browsing workload navigates through a popular news article on m.cnn.com. The workload sleeps a predetermined number of seconds (45) before scrolling down the page as a normal user would. Then the workload sleeps again (25 seconds) to simulate reading the end of a page before moving to the next page of the article. Each time the browser loads a new web page, we see a significant change in both memory and file system activity. The changes in the file system are due to the fact that the Android browser maintains a cache of previously viewed web pages and thus there are additional changes each time a new page is loaded.

Figure 4 shows the checkpointing effects during a streaming video using the YouTube app for Android. In the case of YouTube, the application is launched and the video is buffered during the beginning of the workload. During video playback, many pages are modified but the overall change in the system remains steady. An important observation is that throughout all our workloads, the most significant overall change to memory (Diff Orig curve) also occurs at the beginning of the workload. After this initial change, the changes following are fairly minor. For this reason, if there is enough time to propagate the changes caused by starting an application, migrating to a different device later in the use of an application should only incur a small switch penalty. On the other hand, if a user switches to a second device within a few seconds of starting an application, the amount of time needed to sync the system state would be much longer and possibly noticeable by the user.

To observe the effects of a file-system intensive workload, we used the Hi-Q MP3 Voice Recorder app to record 3 min-

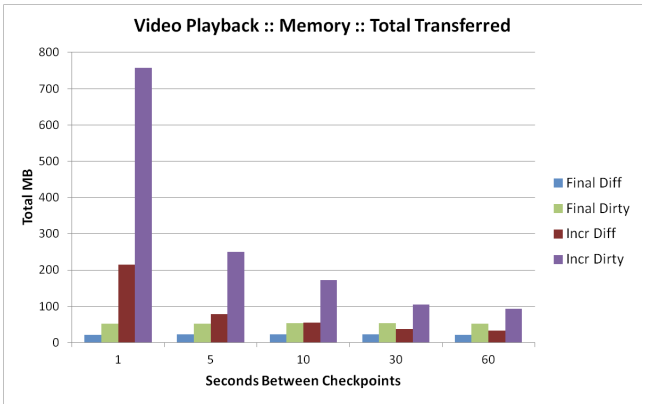

Figure 6: Transferring complete dirty memory pages involves three times more data than necessary.

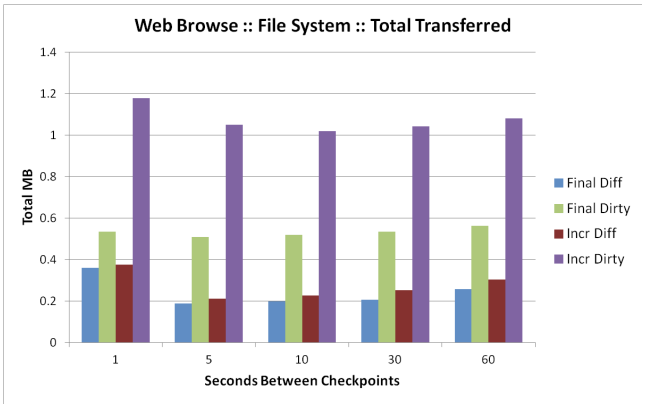

Figure 7: Transferring only the bytes that have changed is also advantageous in the file-system case.

utes of audio within our VM. This workload continuously writes to the file system and finishes with a final 1.3 MB change to the file system. Figure 5 shows the results when executing a file system checkpoint every 1 second. When compared to the original system image, the changes over the course of the workload are continuously increasing over time. Every checkpoint, the changes between the previous state are consistently minimal, but add up to the final total change at the end of the workload. Though a user may not switch devices while recording an audio session, this represents the feasibility of switching between devices when data is being constantly written to the file system.

Measuring the Total Bytes Required for Sync

For each workload, we vary the time between checkpoints from 1 second to 5, 10, 30, and 60 seconds. Due to the fact that each workload runs for a fixed amount of time, the number of checkpoints decreases as the amount of time between checkpoints increases. Because of this, the overall shape of each checkpoint curve is similar, but smoother and less fine grained over time. Figures 6, 7, 8, and 9 compare the total amount of data required for each checkpointing policy against the time interval between checkpoints. While we are not measuring the switch penalty directly in units of time, the size of data transfers under various policies gives us some indication of which policies would have higher or lower switch penalties. *Final Diff* represents the exact number of megabytes changed at the end of the workload, where *Final Dirty* is the total number of megabytes required if you synced each dirty page or block. When using a policy that compares against each previous checkpoint, we sum the number of megabytes changed for each checkpointing round and report the result as the total number of megabytes changed for that policy. *Incr Diff* represents the total number of bytes changed, where *Incr Dirty* represents the total number of complete pages that have changed over incremental checkpoints.

In all cases, the Final Diff and Final Dirty results remain constant (within a standard deviation of the workload) across varying time intervals. As the time interval between checkpoints increases, the number of checkpoints decreases, lowering the total number of megabytes for incremental checkpoints. Because memory state changes very frequently while a workload is running, checkpointing memory state often, such as every second, transfers large amounts of data. In the case of video playback, shown in Figure 6, propagating each memory page that has changed would require three times the amount of data compared to propagating the difference of each page. In fact, propagating dirty pages could use up to 35 times the amount of data than necessary in the worst case, measured from our audio recording workload. In all cases, propagating only the bytes that have changed at the end of the workload uses the least amount of data, but could cause a severe delay if the user decides to switch devices while running an application. It is also important to note that both differencing policies, final and incremental, may incur significant time and space overhead. We plan to measure these overheads and explore the trade off between each in a full-fledged implementation of VMsync.

Compared to memory state, the changes in file system state are fairly minimal, even in a file-system intensive workload such as audio recording (Figure 8). As the time between checkpoints increases, propagating dirty blocks every checkpoint converges closely to the actual number of megabytes changed. This is because during sequential writes, data is written to a single block at a time and the number of dirty blocks ends up being proportional to the total number of bytes changed. For writes spread out across multiple different blocks, such as in the web browsing workload in Figure 7, propagating dirty blocks incurs a cost about five times higher than performing a final diff.

Since mobile devices users can perform multiple tasks at once, we performed a combined workload which added background music, using the default Android music player, to our web browsing workload. In general, playing audio in the background has minimal effect on the system state. As shown in Figure 9, the size of changes to both the memory and file system state are nearly the same as web browsing alone, leading us to infer that audio playback continuously modifies a fixed number of memory pages and does not change the file system.

Following are some overall findings from this study. One, naive policies that only propagate dirty pages or blocks transfer large amounts of data and would not be feasible for mobile devices and networks. Two, large savings result from performing differences of previous checkpoints in order to propagate only the bytes that have changed. We also see that increasing the amount of time between checkpoints will save bytes but may require a higher switch latency if a user

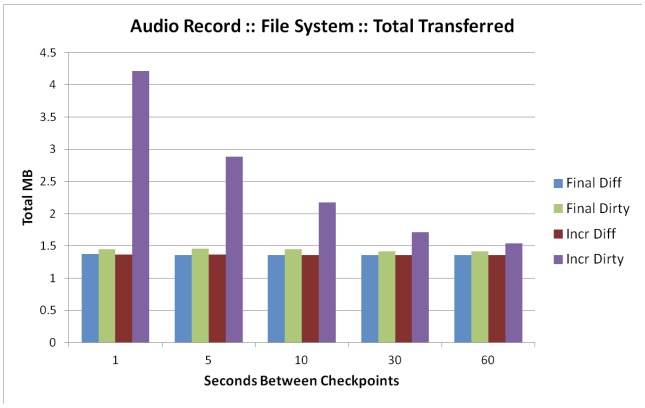

Figure 8: Waiting before propagating dirty file-system blocks approaches the cost of a differencing policy for apps that perform sequential writes.

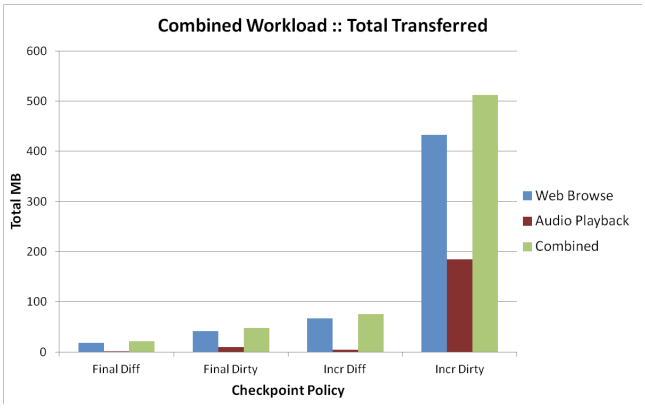

Figure 9: Playing music in the background of a web browsing session has minimal effect on the overall system state.

switches devices during that time. With regards to memory, the most significant changes occur in the beginning of a workload and subsequent changes are somewhat minimal.

5. CONCLUSION

We have presented our early work on VMsync, a system for incrementally synchronizing live virtual machine state among mobile devices. VMsync aims to enable users to seamlessly switch between devices with both data and computation state preserved across the switch without apparent delay. We described our initial design for identifying changes to the memory and file-system images of an active VM on one device, then propagating those changes to standby VMs on other devices via a synchronization server in the cloud. We also presented our measurements study of how much data would need to be transferred to maintain a consistent VM image across devices under different workloads and synchronization policies. From our efforts to date, we conclude that VMsync is a feasible approach with many open issues deserving of further research.

6. REFERENCES

[1] Android-x86 - Porting Android to x86. http://www.android-x86.org/.

[2] A. Surie and H. A. Lagar-Cavilla and E. de Lara and M. Satyanarayanan. Low-bandwidth VM Migration via Opportunistic Replay. In *Proc. 9th workshop on Mobile Computing Systems and Applications*, 2008.

[3] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proc. 23rd Symposium on Operating Systems Principles*, 2011.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. 2nd Symposium on Networked Systems Design & Implementation*, 2005.

[5] L. P. Cox and P. M. Chen. Pocket Hypervisors: Opportunities and Challenges. In *Proc. 8th IEEE Workshop on Mobile Computing Systems and Applications*, 2007.

[6] K. Gudeth, M. Pirretti, K. Hoeper, and R. Buskey. Short Paper: Delivering Secure Applications on Commercial Mobile Devices: The Case for Bare Metal Hypervisors. In *Proc. 1st ACM Workshop on Security and Privacy in Mobile Devices*, 2011.

[7] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. 4th IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[8] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proc. ACM Workshop on Security and Privacy in Mobile Devices*, 2011.

[9] Mashable. App Store Stats: 400 Million Accounts, 650,000 Apps. `http://mashable.com/2012/06/11/wwdc-2012-app-store-stats/`, June 2012.

[10] P. Svärd and B. Hudzia and J. Tordsson and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proc. 7th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2011.

[11] R. Cáceres and C. Carter and C. Narayanaswami and M. Raghunath. Reincarnating PCs with Portable SoulPads. In *Proc. 3rd International Conference on Mobile Systems, Applications, and Services*, 2005.

[12] VMware. Verizon Wireless and VMware Securely Mix the Professional and Personal Mobile Experience with Dual Persona Android Devices. `http://www.vmware.com/company/news/releases/vmw-vmworld-emea-verizon-joint-10-19-11.html`, October 2011.

[13] A. Wolbach, J. Harkes, S. Chellappa, and M. Satyanarayanan. Transient customization of mobile computing infrastructure. In *Proc. 1st Workshop on Virtualization in Mobile Computing*, 2008.