Efficient State Transfer for Internet Suspend/Resume

Michael Kozuch, M. Satyanarayanan, Thomas Bressoud, Yan Ke

IRP-TR-02-03
May 2002

Intel Research Pittsburgh

Efficient State Transfer for Internet Suspend/Resume

Michael Kozuch[‡], M. Satyanarayanan[†‡], Thomas Bressoud[‡⋆], Yan Ke[†‡]
[‡]Intel Research Pittsburgh, [†]Carnegie Mellon University, and [⋆]Denison University

Abstract

We report on a new capability for mobile computing called Internet Suspend/Resume. *This mechanism mimics the opening and closing of a laptop, but avoids physical transport of hardware. We show that this capability can be implemented by layering virtual machine technology on a distributed file system. We also show how the key obstacle of large VM state size can be overcome by exploiting proactivity and by using incremental state transfer to the resume site. Our experiments confirm that these techniques are successful in reducing resume latency to just a few seconds for a typical present-day machine configuration. The paper describes a number of state transfer policies and quantifies their relative merits using an industry-standard benchmark.*

1 Introduction

Internet Suspend/Resume (ISR) is a new capability for mobile computing that mimics the opening and closing of a laptop, but avoids physical transport of hardware. Through rapid and easy personalization and depersonalization of anonymous hardware, a user is able to effortlessly suspend work at one machine and to resume it at another. ISR can be implemented by layering virtual machine (VM) technology, such as VMware Workstation [5], on a distributed file system such as NFS, AFS or Coda. The VM performs the encapsulation of execution state and user customization; the distributed file system transports that state across space and time between suspend and resume. Use of a VM for state encapsulation eliminates the need for modifications to applications or the operating system. As a result, ISR supports unmodified Windows software.

A key obstacle to realizing ISR is *large resume latency.* The state of a typical VM today is very big — at least many GB, and possibly many tens of GB. The time it takes to move this state to the resume site may be intolerable. Since disk capacity, and hence VM state size, is growing much more rapidly than end-to-end Internet bandwidth, this delay will only worsen over time.

This paper describes how we have overcome this problem and built an ISR system with resume latency of just a few seconds — comparable to the typical delay one experiences after opening a laptop. Specifically, for a VM configured with a 4GB disk and 256MB RAM running Windows XP and an industry-standard benchmark, we demonstrate best-case resume latency of about 2.5 seconds. This assumes ample travel time betweeen suspend and resume sites, advance knowledge of resume site, and LAN connectivity. Under the less optimal conditions of minimal travel time and no prior knowledge of resume site, we demonstrate resume latency of about 30 seconds.

We begin in Section 2 with a brief review of virtual machines, and alternative approaches to user mobility. Then, in Section 3, we describe lessons from an early proof-of-concept prototype. We then motivate and describe an improved prototype in Section 4. Next, in Section 5, we describe a range of state transfer policies, their strengths and weaknesses, and the results of experiments that quantify these tradeoffs. Finally, in Sections 6 and 7, we discuss future work and summarize the main results of the paper.

2 Background and Related Work

VMware Workstation (abbreviated to just "VMware" in the rest of this paper) is a modern, commercial *virtual machine monitor (VMM)* [4] that provides a VM abstraction identical to a PC. The VMM runs within a *host* operating system and relies on it for common system services such as device management. The operating system that executes within a VM is referred to as a *guest* operating system. VMware supports many operating systems as guests including Windows 95/98, Windows 2000/XP, and Linux. A user can configure many important parameters that define the VM including the amount of memory, size and arrangement of disks, and number of network adapters.

The state of each supported VM is mapped to files in the local file system of the host. For example, if a VM is named `testvm`, its configuration is in the

file `testvm.cfg`. This file describes the types, numbers, and arrangement of the virtual hardware components. Another file, `testvm.log`, serves as a logging mechanism for the VMM. The non-volatile state of the virtual system is maintained in a handful of files: one for the standard PC non-volatile memory state (`testvm.nvram`), and one for each of the virtual disk drives (`testvm1.vmdk, testvm2.vmdk,` and so on). These non-volatile state files provide enough information to restart the VM after it has been powered-off. If `testvm` is suspended, the file `testvm.vmss` captures the volatile state of the processor, devices, and main memory at the point of suspension. Once `testvm` has been suspended, these files can be copied to a remote host with similar hardware architecture. A VMM on the remote host can then resume the VM. In other words, the VM has been *migrated.*

In contrast to the well-known difficulties of process migration [2, 13, 15], VM migration is simpler because volatile execution state is better encapsulated. It is also tolerant of greater disparity between the source and target systems across which migration occurs. For process migration to succeed, there has to be a very close match between host and target operating systems, language runtime systems, and so on. In contrast, VM migration only requires a compatible VMM and hardware architecture at the target. The price for this flexibility is a substantial increase in state transfer size.

Another well-documented approach to user mobility is to use a thin client. In this case, all execution is done remotely on a compute server and only the user interface follows a user as he moves around. Examples of this approach include Infopad [14], SLIM [10], VNC [8] and X-Move [11]. This approach is attractive in a well-connected networking environment because little state has to be transferred across sites when a user moves. However, it suffers from poor usability when network latency is high and fails completely when the client is disconnected. In contrast, ISR only requires network connectivity while state is being transferred to the resume site. After that, the network can be disconnected until the next move of the user. Interactive response is crisp even when network latency is high, because execution is local.

3 Initial Prototype

To gain hands-on experience with ISR, we implemented a simple prototype using NFS. The host OS was Linux and the guest OS was Windows XP configured for a VM with 128 MB of RAM and a 2 GB virtual disk — a small configuration by today's standards. Since the virtual disk was only half full, VMware condensed the virtual disk data file to about 1 GB. VM state resided in the local file system. On suspend, these files were copied out to NFS; on resume, they were copied in from NFS to the new site.

Figure 1 summarizes the observed suspend and resume times on this prototype. Full details of these experiments can be found elsewhere [7]. As the second column of Figure 1 shows, resume took roughly two minutes while suspend took about two and a half minutes. These times are more than an order of magnitude larger than the few seconds of delay experienced when opening or closing a laptop.

Users tend to perceive resume latency more acutely than suspend latency because suspend can overlap travel — a user can depart immediately after initiating suspend unless he is paranoid about the operation failing. We used file compression to take advantage of this asymmetry in user perception. VM files were compressed at the suspend site before copyout to NFS; they were decompressed after copyin at the resume site. The third column of Figure 1 shows that resume time is reduced by nearly 40% to about 73 seconds, while suspend time is increased by 8%.

Event	No Compression	With Compression
Resume	125 (0.2)	73 (4.3)
Suspend	146 (19.6)	158 (0.9)

This table shows the average time, in seconds, of suspend and resume operations with cold NFS file caches. All experiments were repeated three times, and the observed standard deviations are shown in parentheses. All machines were 1.7 GHz Pentium 4 with 512 MB DRAM running Red Hat Linux 7.2, VMware Workstation 3.0 and NFS v3. The network was 100 Mb/s Ethernet. Compression was done with `gzip`.

Figure 1. Performance of Initial Prototype

4 Improved Prototype

4.1 Design Rationale

Our initial prototype confirmed the thesis underlying ISR: by layering a VM on top of a distributed file system, one can indeed suspend execution at one location and resume it seamlessly elsewhere. No modifications are necessary to the operating system or applications. In particular, ISR works robustly with Windows.

At the same time, our experiments clearly showed that resume times comparable to laptop suspend/resume are not attainable without a more sophisticated implementation. We have therefore built a new prototype that improves upon the original in a number of ways. In the rest of this section, we discuss the key factors that have influenced our redesign.

First, we wish to support ISR anywhere on the Internet, including locations with low-bandwidth connectivity. A typical scenario we envision is a business traveller in a hotel room with just a modem. Since NFS only works well in LAN environments, we have decided to use Coda instead. Coda's support for weakly-connected and disconnected operation [9] fits well with our goal of ubiquitous ISR.

Second, temporal locality is often present in the mobility patterns of users. For example, a common usage pattern we envision for ISR is a user working at home, suspending, travelling to his office, and resuming there; later in the day, he suspends at the office, returns home, and resumes. As another example, a worker in a corporate campus or a supervisor in a factory might visit the locations of his coworkers many times in the course of a day. We wish to exploit such mobility patterns to improve ISR performance.

Third, we envision many situations where ISR allows a user to take advantage of an unanticipated sliver of time for productive work. For example, during an unexpected delay in a doctor's office, we would like a user to be productive rather than idly leafing through a magazine. For such situations with brief usage intervals, rapid resume is essential even at the cost of slight delays later.

Fourth, it is sometimes possible to confidently predict where a user will resume work. For example, when a user leaves for the airport after suspend it is likely that he will resume either in his airline's lounge or at his preassigned aircraft seat. With the assistance of higher-level software, it may be possible to identify likely resume locations and proactively transfer state to those locations. This will lower resume latency. Since proactivity merely requires warming a file cache in our design, the consequences of acting on an erroneous prediction are mild. Useful file cache state may be flushed and network bandwidth may be wasted, but there is no loss of correctness or need for cleanup actions.

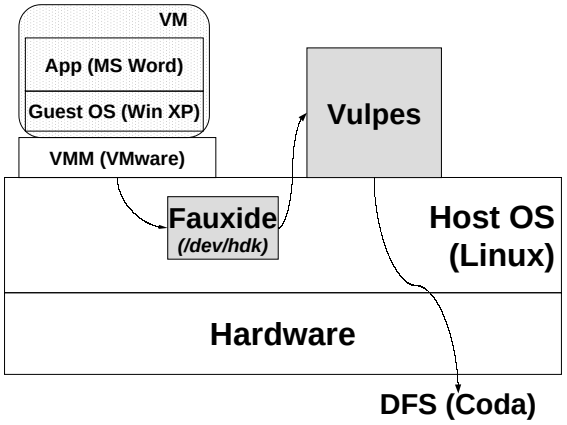


Figure 2. ISR Host Architecture

4.2 Architecture

Figure 2 shows the host architecture that we have developed in response to these design considerations. A key attribute of this architecture is that it gives us considerable flexibility in experimenting with a wide range of state transfer policies.

A loadable kernel module called *Fauxide* serves as the device driver for a pseudo-device named `/dev/hdk` in Linux. A VM is configured to use this pseudo-device as its sole virtual disk in "raw" mode. References to this pseudo-device from the VM are redirected by Fauxide to a process called *Vulpes*. It is Vulpes that implements the state transfer policy for the VM's disk by controlling the mapping of its data to files in Coda. It also controls the hoarding (i.e., Coda cache warming) of those files. Since Coda uses whole file caching, Vulpes divides the virtual disk into 256KB chunks and maps each to a separate file. These files are organized as a tree in Coda. VMware also uses Coda to store the other components of VM state mentioned in Section 2. Since these files are relatively small compared to a virtual disk, whole-file caching does not pose a problem.

5 VM State Transfer Policies

The copyout/copyin state transfer policy of our initial prototype (Section 3) represents the most conservative endpoint in a spectrum of possible policies. No attempt is made to propagate dirty state before suspend, and resume is blocked until the entire state has arrived. One can take three steps to shorten resume latency:

- allow resume to occur before full state has arrived. This overlaps execution at the resume site with

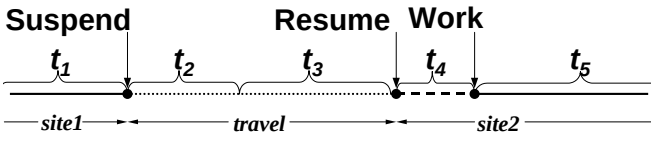


Figure 3. Conceptual ISR Timeline

state transfer.

- proactively warm the Coda file cache at the resume site. This reduces the contribution to resume latency of fetch delay from file servers.
- aggressively propagate dirty state before suspend. This reduces the contribution to resume latency of store delay to file servers.

These approaches are not mutually exclusive, and can be combined in many ways to generate a wide range of policies. In the sections that follow, we explore the relative merits of a number of these policies.

The conceptual timeline shown in Figure 3 provides a uniform framework for discussing these policies. The figure depicts a user initially working for duration *t1* at Internet location *site1*. He then suspends, and travels to Internet location *site2*. In some situations, the identity of *site2* is known (or can be guessed) *a priori*. In other situations, it becomes apparent only when the user unexpectedly shows up and initiates resume. The transfer of dirty state from *site1* to file servers continues after suspend for duration *t2*. There is then a period *t3* available for proactive file cache warming at *site2*, if known. By the end of *t3*, the user has arrived at *site2* and initiates resume. He experiences resume latency *t4* before he is able to begin work again. He continues working at *site2* for duration *t5* until he suspends again, and the above cycle repeats itself. With some state transfer policies, the user may experience slowdown during the early part of *t5* because some operations block while waiting for missing VM state to be transferred. Although the architecture shown in Figure 2 makes this deferred state transfer functionally transparent, its performance delay cannot be masked.

It is important to observe that Figure 3 is only a canonical representation of the ISR timeline. Many special or degenerate cases are possible. For example, *t2* may not end before resume if travel duration is very short. In that case, the residue of *t2* may add to *t4* in contributing to resume latency. On the other hand, a clever state transfer policy may allow this residue to overlap *t4*. In other words, propagation of dirty state from the suspend site to file servers could overlap state propagation from those servers to the resume site. Another special case is when *t5* is very brief. With such a short dwell time, full VM state may never accumulate at *site2* — only enough to allow the user a few moments of work past the suspend point at the end of *t1*. While many such special cases are conceivable, the timeline in Figure 3 is likely to cover a wide range of common real-world scenarios.

5.1 Metrics

From a user's perspective, the key performance metrics of ISR can be characterized by two questions:

- **Resume latency**: *How long after I resume at a new site can I begin useful work?* The answer to this question corresponds to the period *t4*.
- **Slowdown**: *How much is my work slowed down after I resume?* The answer corresponds to the slowdown during *t5*.

Ideally one would like zero resume latency and zero slowdown. In practice, there are tradeoffs between the two. Policies that shrink resume latency may increase average slowdown and vice versa. Our goal is to quantify these tradeoffs for workloads representative of anticipated ISR usage.

5.2 Policy Considerations

We define the following virtual machine state transfer policies.

Baseline The baseline policy most closely approximates the operation of ISR under the original prototype, but adapted to the new architecture. Refer to Figure 3. After a suspend, the entire state of the virtual machine, including both the disk and memory images, is transferred to the server during *t2*. The period *t3* is empty, and, following a resume, the entire state of the virtual machine is transferred to the resume site during *t4* and pinned to the client cache. Note here that no state transfer occurs during either execution period *t1* or *t5*, optimizing for execution speed at the cost of suspend and resume latency.

This policy is applicable when *site2* can not be predicted and when the site may become disconnected after a successful resume. For this policy,

we expect the resume latency to be the longest, as it transfers the entire state in *t4*, but expect slowdown to be the shortest, because all virtual machine state is available before resume.

Fully Proactive If we can predict *site2*, we can define a much more aggressive state transfer policy. At *site2*, this policy shifts the entire state transfer time from *t4* to earlier periods in the ISR timeline. During *t3* (or earlier, for any state already available at the servers) *site2* transfers all updated state to its local cache. Note that this includes both VM disk and memory state. At resume, all that remains is to launch the VM.

We envision this policy to be most effective when a user is working between a small set of sites, such as home and work. If two sites start in a synchronized virtual state, then the state required to be transferred during travel time is limited to the unique state modified during execution at *t1*. Like the baseline policy, after a successful resume the fully proactive policy permits operation at *site2* while disconnected.

For this policy, we expect resume latency to be shortest, because all state transfer has been moved to time *t3*. Slowdown will also be the least because all VM state is available before resume.

Pure Demand-Fetch Suppose a user arrives unexpectedly at a new site. If we wish to keep *t4* as short as possible, we can amortize the cost of retrieving the VM disk state over *t5* by using a pure demand-fetch policy. In this policy, only immediately-required VM state is retrieved during *t4*, but transfer of the disk state is deferred. As soon as the critical state has arrived, the VM may be launched. Then, during *t5*, disk accesses by the VM are demand fetched via Vulpes from the Coda server.

We expect the resume latency for this policy to be short, as only critical state is transferred during *t4*. We also expect substantial slowdown because client cache misses during *t5* introduce delay.

Working Set While the pure demand-fetch policy allows execution to commence without VM disk state present at *site2*, the downside is the performance penalty until the client cache becomes warm. This penalty can be somewhat mitigated by warming the *site2* cache. In the Working Set policy, the *site1* system estimates the current file cache working set. The system then sends a description of this set either to the server or *site2*.

If the resume site can be predicted with confidence, the working set may be prefetched at *site2* during *t3*, which we call the **Eager Working Set** policy. If execution must be resumed at an unpredicted site, the working set may be fetched concurrently with demand-fetch during *t5*. We call this the **Lazy Working Set** policy.

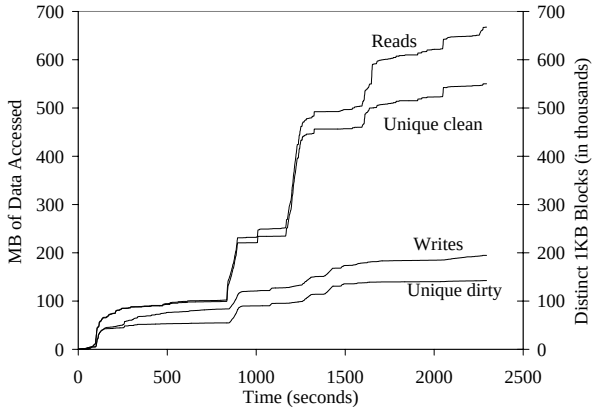
Eager Writeback Another aspect of VM state transfer is the question of when dirty state is conveyed from *site1* back to the server. We define eager writeback as a policy in which modified VM disk state is aggressively pushed to the server during *t1*. The effect of this is to reduce *t2* and thus to accomodate a shorter travel interval between suspend and resume. Of course, these eager writeback accesses may interfere with demand fetches and may degrade *t1* performance if not scheduled well.

5.3 Benchmarks

We have used three benchmarks in our evaluation of virtual machine state transfer policies. Two of these are commercial benchmarks for Windows produced by the Business Applications Performance Corporation [1]. These are named *SYSmark2002 Office Productivity (SOP)* and *SYSmark2002 Internet Content Creation (SICC)*. The third benchmark, *Roaming Software Developer (RSD)* was created by us and is representative of a software developer who uses ISR. We describe these benchmarks in the rest of this section.

SOP uses script-driven Windows applications to model a single user's office activity in an automobile company. The user creates and edits documents using Microsoft Word, Excel and Powerpoint. He accesses email using Microsoft Outlook and queries a database using Microsoft Access. He views presentations on the Web using Netscape Communicator. A part of the benchmark includes speech to text translation using Dragon NaturallySpeaking. Another part constructs a file archive using WinZip. McAfee VirusScan is run in the background during the entire benchmark.

SOP executes in a VM in about 2300 seconds of elapsed time on the hardware used in our experiments (described in Section 5.4). This includes delays modelling user think time. The benchmark concurrently



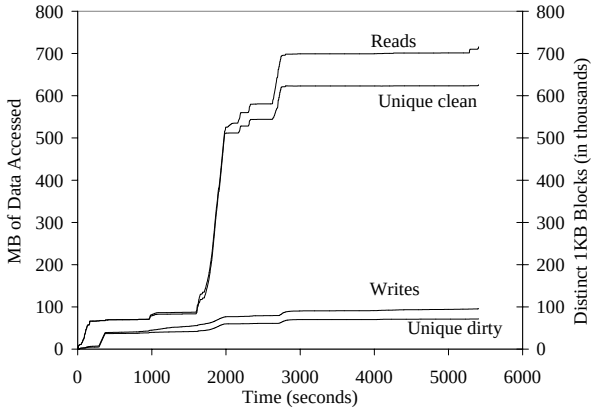
The left axis applies to the curves labelled "Reads" and "Writes"; these show the cumulative volume of read and write traffic generated over the life of the benchmark. The right axis applies to the curves labelled "Unique clean" and "Unique dirty"; these show cumulative number of distinct 1KB disk blocks read or written during the benchmark. The difference between "Read" and "Unique clean" indicates the extent of temporal read locality. Similarly, the difference between "Write" and "Unique dirty" indicates the extent of temporal write locality. Note that the right axis is in units of thousands of 1KB blocks.

Figure 4. SOP Data Access Characteristics

executes multiple applications and models the user switching between them during his work. Figure 4 shows the observed data access characteristics of SOP. We obtained this information by having Vulpes trace disk addresses while SOP is running in the VM of Figure 2.

Like SOP, SICC also consists of script-driven Windows applications. SICC models the work of a designer creating Web pages for a sports company using Macromedia Dreamweaver. He embeds images developed with Adobe Photoshop and animations created with Macromedia Flash. He then creates a promotional video using Adobe Premiere and encodes it using Windows Media Encoder. User think time, concurrency, and switching between applications are modelled as for SOP. Figure 5 shows the observed data access characteristics of this benchmark. SICC executes in about 5400 seconds on our hardware.

The sole performance metric reported by SOP and SICC is *average response time.* This is the time, averaged across all benchmark operations, that it takes the system to complete an operation initiated by the benchmark script. In Microsoft Word, for example, the response time for a "Replace All" operation is the delay between clicking in the "Edit/Replace" dialog box and the appearance of the completion dialog box. Response time is thus a direct measure of system sluggishness perceived by the user. In the context of ISR, this metric includes any delays induced by state transfer policy — for example, the delay caused by a file cache miss in Coda. The slowdown metric for an ISR policy is given by the difference between the average response time when a benchmark runs with that policy and the corresponding time for a reference configuration.



The interpretation of these curves is the same as for Figure 4. Note that the right axis is in units of thousands of 1KB blocks.

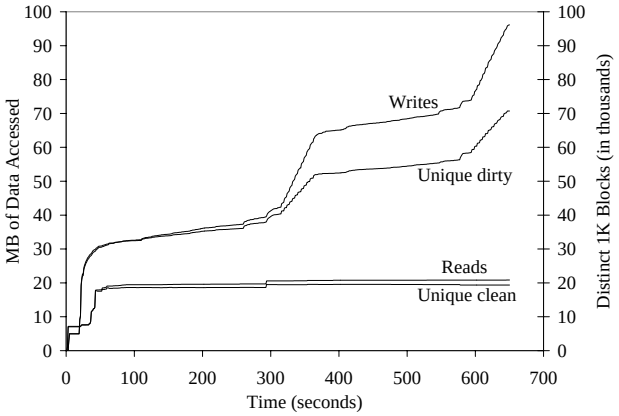
Figure 5. SICC Data Access Characteristics

Focusing solely on average values can sometimes be misleading. Looking at distributions or grouping by application may offer useful insights. Although such detailed data can be extracted from SOP and SICC, we are unable to present them here. This is because the SYSmark licensing terms prohibit reporting of any numbers other than average response time.

The RSD benchmark models a software developer working in a CygWin environment on Windows. The benchmark consists of a sequence of tasks with a fixed 10-second delay between them to model think time. The developer first untars and unzips the source archive of the Sphinx speech recognizer. He then configures the source tree and then builds Sphinx. Next, he `grep`'s the source tree for a string, touches all header files in the tree, and then rebuilds Sphinx. Figure 6 shows the data access characteristics of RSD. The total running time of RSD is 640 seconds on our hardware. The difference in time between a configuration that uses a particular ISR policy and a reference configuration gives the slowdown caused by that policy. The slowdown of individual tasks in the benchmark offers more detail on the effects of the policy.

5.4 Experimental Setup and Methodology

Our experimental infrastructure consisted of single-processor client computers connected to single-processor Coda servers through 100 MB/s Ethernet.



The interpretation of these curves is the same as for Figure 4. Note that the right axis is in units of thousands of 1KB blocks.

Figure 6. RSD Data Access Characteristics

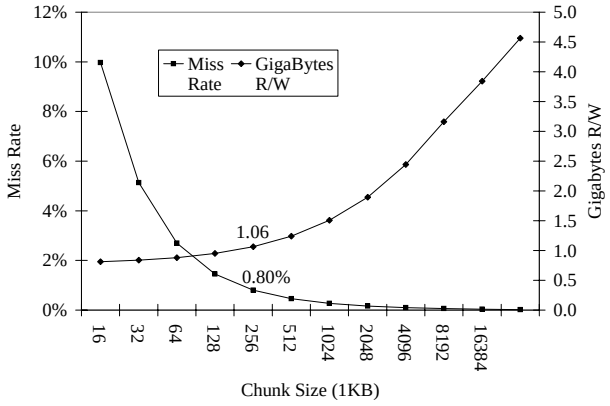


Figure 7. Disk State Chunk Caching

The clients were 2.0 GHz Pentium 4 processor-based computers with 1 GB of SDRAM, and the servers were 1.2 GHz Pentium III Xeon processor-based machines with 1 GB of SDRAM.

All computers were running RedHat 7.2 with the Linux 2.4.18 kernel installed. The clients were running VMware Workstation 3.1 and version 5.3.19 of Coda with an 8 GB file cache. The non-replicated servers were also running version 5.3.19 of Coda.

5.4.1 Virtual Disk Representation

As mention in Section 4.2, Vulpes accomodates Coda's whole-file caching by dividing VM disk state into 256 KB chunks. We use a two-level directory structure to organize the virtual disk file chunks.

Chunk size may have a serious impact on the user experience. With larger chunk size, Coda's whole-file caching will waste bandwidth when partially written files are transferred to servers. Further, if the ISR policy is demand-driven, each demand-miss on the client will result in a chunk fetch. Again, if the chunks are too large, bandwidth will be wasted as the system fetches data that it never uses. On the other hand, if the chunk size is too small, we miss an opportunity to exploit spatial locality – resulting in a greater file cache miss ratio.

To determine appropriate values for the chunk size, we captured a trace of the disk blocks fetched during the execution of the two Sysmark benchmarks. We adapted a cache simulation package, Dinero IV [3], to calculate the file chunk miss rate and bandwidth consumed during workload execution for various chunk sizes. In these experiments, we assumed that the client was operating in a demand-fetch mode and that client chunk cache was large enough to contain the entire set of file chunks. The results of these simulations are shown in Figure 7. In our experiments we have chosen to employ a chunk size of 256 KB to strike a balance between the miss rate and wasted bandwidth.

5.4.2 Snapshots

For experimentation with the various benchmarks at different points in their respective execution, we maintain VM state snapshots of each benchmark. For each benchmark, the first snapshot, denoted *s0* is obtained by suspending the VM at the start of the workload. For SOP and SICC, this point is after SysMark has finished its setup and the launch of the windows applications is about to commence. For RSD, this point is immediately preceeding the execution of any of the component steps of the benchmark.

For each benchmark, a mid-execution snapshot, denoted *s1*, was taken by suspending the VM at a point approximately halfway through the workload.

Finally, a terminal snapshot, denoted *s2*, was taken for each workload by suspending the VM after completion of the workload. For SOP and SIPP, this occured after the conclusion of the windows applications, but before the cleanup/summary actions of the Sysmark driver script.

Policy	SOP	SICC	RSD
Baseline	2060 (29)	2020 (82)	2034 (23)
Proactive	2.5 (0.15)	2.5 (0.14)	3 (0.02)
Demand	27 (1.9)	33 (5.1)	18 (1.0)

Average resume time in seconds and standard deviation for three iterations of the SOP, SICC, and RSD benchmarks.

Figure 8. Resume latency for s1 snapshot

5.5 Evaluation: Resume Time

A summary of the resume times for three ISR policies and our three benchmarks is provided in Figure 8. For each benchmark, we resumed a VM from snapshot s1. We define the end of the resume operation to be the first instant that the user is able to interact with the VM.

In the baseline policy, resume involves three serial phases. First, the disk data is transferred from the server to the client. Second, the memory image is transferred from the server to the client. Finally, VMware is launched.

The pure demand-fetch policy launches the VM before the entire disk image is cached locally. Hence, this policy avoids the latency of the first phase. The resume time for demand-fetch is the sum of the memory image transfer time and the VMM launch time.

The fully proactive policy further reduces resume time by eliminating the memory image transfer phase. Under this policy, we have assumed that the client machine is constantly synchronizing cached copies of the disk and memory data with the server. Consequently, with a warm client file cache, the resume time is equal to the time required to launch the VMM.

Figure 8 shows that by employing either the fully proactive or pure demand-fetch policies, we can reduce the user-perceived resume latency substantially. The proactive policy improves the resume time from approximately 2000 seconds to 2.5 seconds. The demand policy also shows a marked improvement – reducing the resume latency from 2000 seconds to approximately 30 seconds.

The resume latency is directly related to the quantity of data that must be transferred during *t4*. The largest portions of VM state are the disk data (divided into chunks as mentioned previously) and the memory image. Identical volumes of virtual disk state is employed in all three images, namely 4 GB. However, because the time required to transfer the memory image is a significant component of the resume time when the user begins with a cold client file cache, we compress

	SOP	SICC	RSD
vmss size	262	262	262
vmss.gz size	96	142	62
compression	63.4%	45.8%	76.3%

Figure 9. Memory state file sizes (MB)

the memory image on the server in our implementation. The size of the resulting files is given in Figure 9.

The difference between the resume times of the benchmarks for pure demand-fetch in Figure 8 can be traced directly to the difference between the sizes of the compressed memory images reported in Figure 9. Naturally, the difference in compressed file sizes is directly related to the entropy of the data in the memory image because the size of the memory allocated to the virtual machine is the same in all cases (256 MB).

Comparing the baseline resume time of Figure 8 to the resume times of our initial prototype (Figure 1), we see that our implementation, based on Coda, is slower than our initial prototype, based on NFS. This difference is due to two factors. First, the initial VM was much smaller – comprising a 2 GB disk which was only half full and 128 MB of main memory. The VMs in the current experiments have disks and memory both scaled by factor of two. Additionally, our current VM state, including the Sysmark installation, is only half as compressible; hence, the volume of data used to create Figure 8 was about four times greater than the data used to create Figure 1. By substituting an rsync operation over NFS for the hoard walk on Coda, we observed data transfer rates that are consistent with the original experiments – approximately 60 MB/s.

The second factor is that the Coda hoard walk operation is much more expensive than an rsync over NFS. The transfer rate during a hoard walk is approximately 18 Mb/s. This factor of three reduction in transfer rate is due partly to the additional bookkeeping involved in the hoard operation, partly to the tighter consistency guarantees of Coda, and partly to the experimental nature of the Coda implementation. However, we felt that the benefits of the hoard walk functionality, which is particularly useful under the fully proactive policy, outweighed the drawback of the reduced transfer rate.

5.6 Evaluation: Response Time

Figure 8 the resume time may be reduced to approximately 30 seconds by employing the pure demand-fetch policy. However, the demand policy incurs a performance penalty. Figure 10 reports our mea-

Policy	SOP	SICC	RSD
Baseline	2.6 (0.32)	3.7 (0.44)	640 (56)
Proactive	2.6 (0.32)	3.7 (0.44)	640 (56)
Demand	5.7 (†)	5.9 (0.44)	660 (18)

Average response time in seconds and standard deviation for three iterations of the SOP, SICC, and RSD benchmarks. †The SOP response time for the demand policy is from a single iteration. We expect to supply the additional two iterations for the final version of the paper.

Figure 10. Benchmark response times

	Baseline/Proactive	Demand
untar	65 (38)	28 (2.4)
configure	62 (2.4)	96 (4.5)
build1	260 (14)	280 (11)
build2	250 (14)	260 (13)
Total	640 (56)	660 (18)

Average execution time in seconds and standard deviation for three iterations of the RSD benchmark.

Figure 11. Component execution times for RSD

surement of this penalty. We gathered the response times for SOP and SICC reported by Sysmark and measured the run time of our RSD benchmark. For the SOP benchmark, the demand policy introduced an approximately 120% increase in response time relative the baseline policy. For the SICC policy, the increase in response time is 59%. The RSD policy saw the smallest demand penalty, 3%, perhaps due to its much higher ratio of writes to reads. The average data request rate seen by Vulpes is comparable in all three benchmarks: 385 requests/s for SOP, 154 requests/s for SICC, and 187 requests/s for RSD.

Figure 11 reports the execution times of the individual components of the RSD benchmark. From the figure, we see that the `build1` and `build2` suffer very little under the demand policy despite generating many disk writes. The `configure` component does suffer a 50% performance degradation under the demand policy. Inexplicably, the `untar` component experienced an average speedup under the demand policy. While we were not able to fully explain this behavior, we believe that it may coincide with background Coda bookkeeping that arises because the Baseline experiments were run with Coda in the disconnected mode and the Demand experiments were executed with Coda in the connected mode.

To further contrast the performance differences between fully proactive and on-demand caching policies, we created a disk-intensive microbenchmark, ENC. This microbenchmark uses the Lame MP3 encoder to

	Baseline/Proactive	Demand
Total	680 (70)	2500 (156)

Average execution time in seconds and standard deviation for three iterations of the ENC microbenchmark.

Figure 12. Execution time for ENC

	SOP	SICC	RSD
Interval s0-s1	888	859	669
Interval s1-s2	413	474	223
Interval s0-s2	1001	1081	767

Number of chunks modified during each workload interval.

Figure 13. Modified chunks per workload interval

compress 990 MB of WAV music files to 90 MB of MP3 files. We measured the total encoding time for both the fully proactive and on-demand case. Figure 12 shows a factor of 3.6 increase in encoding time between the proactive and on-demand policies for our disk-intensive workload.

5.7 Evaluation: Incremental Copyout/Copyin

Another important aspect of ISR is the volume of data and time required to copy modified state at *site1* to the DFS (interval *t2*). For the fully proactive policy, this volume of data must also be transmitted to the hoarding sites during *t3*.

Given our VM snapshots, the volume of modified data corresponds to the set of chunks that were modified during the time intervals of the workloads, *s0-s1* for the first half of the workload, *s1-s2* for the second half of the workload, and *s0-s2* for the entire workload interval. The counts of modified chunks are presented in Figure 13.

If we wish to translate these counts to required time, we can use the measured transfer time in and out of Coda during a hoard walk and apply that transfer rate to the chunk counts. When we do this, we get the copyout/copyin transfer times presented in Figure 14.

One can interpret these results relative to a usage scenario. Suppose we take the case of migrating VM

	SOP	SICC	RSD
Interval s0-s1	98.6	95.4	74.3
Interval s1-s2	45.8	52.6	24.7
Interval s0-s2	111.2	120.1	85.2

Estimated time in seconds required to copy all modified file chunks to the server.

Figure 14. Estimated transfer time for modified chunks

state from work to home. Assume that both sites start in the same synchronized state. At work, the user is performing work similar to the SOP benchmark and suspends at time *s1*. The time required to copyout VM disk state should take about 99 seconds, and the time to compress and copyout VM memory state should take about 30 seconds[1] This copyout occurs after the suspend point during interval *t2*. If the user effectively employs the fully proactive policy, the system will transfer the same data to the set of synchronizing clients during interval *t3*.

6 Future Work

This work can be extended by investigating a broader range of state transfer policies. We plan to begin with the Eager Working Set, Lazy Working Set, and Eager Writeback policies described in Section 5. Quantifying the tradeoffs of these policies for the SOP, SICC and RSD benchmarks is an obvious first step. Broadening the range of benchmarks is another way to extend this work. The ultimate validation is, of course, to deploy ISR to a user community and to obtain empirical feedback from the deployment. Before such deployment is possible, some important limitations of the current prototype will have to be addressed. We discuss these in the rest of this section.

Security is not addressed in our current implementation. There are two dimensions to this problem. First, the distributed file system used for ISR may not encrypt network transmission. Transmitting VM state in the clear is particularly risky because volatile state may contain highly sensitive information such as passwords in the clear. We plan to address this by encrypting data at the suspend site, and decrypting it at the resume site. Hence, suspended VM state will never be transmitted or stored in the clear. Second, there is a need for mutual authentication of user and host at resume. Our approach here will be to leverage the work of others such as the Trusted Computing Platform Alliance [12].

As ISR becomes popular, we expect that users will demand some conveniences to augment the basic implementation described here. One such feature is the ability to ensure that at most one copy of a VM is executing at a time. This guards against a user absent-mindedly resuming a second time using stale execution state, after an initial resume elsewhere. The mutual exclusion mechanism we envision is a simple VM lock server on the Internet that is accessed through a Web browser. If a resume request fails, the lock server could return information identifying the location at which the VM is already executing, when it was resumed there, and so on.

A related, but distinct, feature is the ability to remotely suspend a VM. This would be useful in a variety of situations where a user departs from a location without suspending his VM. For example, he may forget to suspend before leaving for home; or, he may visit a coworker's location and discover that he needs to resume there. Of course, there are important security issues that would have to be addressed in implementing such a mechanism.

Another important area of future effort is to explore techniques for synthesizing much of the transferred state at the resume site. This can reduce the volume of data that has to be transferred between suspend and resume. It can also can be effective in the absence of locality — for example, when a user resumes at an unexpected site. This approach has promise because a large fraction of disk content on a personal system consists of standard operating system and application software (e.g., Windows and the Microsoft Office suite). If disk images of the standard software are available at the resume site, suspended state can be reconstructed by first applying those disk images and then applying overlays transmitted from the suspend site. This technique can be extended to situations where network bandwidth is highly uneven across sites. The large standard disk images can be widely distributed using peer-to-peer techniques such as Gnutella [6] and accessed at high bandwidth from a nearby site. Only the much smaller user-specific state (user directory contents plus customized system files) needs to be transmitted at low bandwidth from the suspend site.

7 Conclusion

Seamless computation in spite of user movement is the holy grail of mobile computing. This paper shows that ISR can be a valuable mechanism to help achieve this goal. To the best of our knowledge, we are the first researchers to identify this unique capability and to show how it can be effectively implemented. The use of VM technology for state encapsulation, and the use of a distributed file system for state transport are

[1] This assumes that the compress and copyout of the memory state is comparable to the copyout and uncompress of the memory state as in the demand policy resume time presented in Figure 8.

the salient aspects of our design. We have shown in this paper that large VM state size, the key obstacle to ISR, can be overcome by exploiting proactivity and by using incremental state transfer to the resume site.

The experiments reported here confirm that resume latency can be reduced to just a few seconds through the use of these techniques. This is comparable to the delay one experiences after opening a laptop. The paper describes a number of state transfer policies and quantifies their relative merits using an industry-standard benchmark. These results give us confidence that ISR has an important role to play in the future of mobile computing.

8 Acknowledgments

AFS is the trademark of IBM Corporation. Linux is the trademark of Linus Torvalds. Microsoft Office and Windows are trademarks of Microsoft Corporation. Pentium is the trademark of Intel Corporation. SYSmark is the trademark of Business Applications Performance Corporation. Any other unidentified trademarks used in this paper are properties of their respective owners.

References

[1] BUSINESS APPLICATIONS PERFORMANCE CORPORATION. *SYSmark 2002*, March 2002. http://www.bapco.com.

[2] DOUGLIS, F., AND OUSTERHOUT, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience 21*, 8 (1991).

[3] EDLER, J., AND HILL, M. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://www.cs.wisc.edu/ markhill/DineroIV/.

[4] GOLDBERG, R.P. Survey of Virtual Machine Research. *IEEE Computer 7*, 6 (June 1974).

[5] GRINZO, L. Getting Virtual with VMware 2.0. *Linux Magazine* (June 2000).

[6] HARRIS, R. Gnutella: a Net autopirate. *Seattle Post-Intelligencer* (April 13, 2000).

[7] KOZUCH, M., AND SATYANARAYANAN, M. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications* (Callicoon, NY, 2002).

[8] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual Network Computing. *IEEE Internet Computing 2*, 1 (Jan/Feb 1998).

[9] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems 20*, 2 (May 2002).

[10] SCHMIDT, B.K., LAM, M.S., NORTHCUTT, J.D. The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles* (Kiawah Island, SC, December 1999).

[11] SOLOMITA, E., KEMPF, J., DUCHAMP, D. XMOVE: A pseudoserver for X window movement. *The X Resource 11*, 1 (1994).

[12] TCPA. *Trusted Computing Platform Alliance: Main Specification Version 1.1a*, November 2001. http://www.trustedpc.org.

[13] THEIMER, M., LANTZ, K., CHERITON, D. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the 10th Symposium on Operating System Principles* (Orcas Island, WA, December 1985).

[14] TRUMAN, T.E., PERING, T., DOERING, R., BRODERSEN, R.W. The InfoPad Multimedia Terminal: A Portable Device for Wireless Information Access. *IEEE Transactions on Computers 47*, 10 (October 1998).

[15] ZAYAS, E. Attacking the Process Migration Bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles* (Austin, TX, November 1987).