# OpenISR 2.0

**Da-Yoon Chung**

July 2015
CMU-CS-15-125

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Mahadev Satyanarayanan, Chair
David A. Eckhardt

*Submitted in partial fulfillment of the requirements
for the degree of Master's of Science*

© 2015 Da-Yoon Chung

**Keywords:** Virtual Machine, VM, Internet Suspend/Resume®, OpenISR®, mobile computing

## Abstract

We explore a new approach to "Web-based Transient Personal Computing" by building upon the ideas from Professor Satyanarayanan's ISR (Internet Suspend/Resume®) to create a much leaner iteration of ISR which also leverages more recent work done in virtualization. ISR decouples machine state from hardware by storing the state in a virtual machine. It increases convenience and productivity for users by allowing them to securely access personal machine state on an anonymous machine (e.g., a PC in a waiting room at the doctors office) or more portable devices, like tablets, instead of being bound to particular hardware. Although the existing ISR is functional, its codebase is complex, and some of the decisions made in its development would have been made differently today. We reconsider the original premises of ISR and build a minimal system in the spirit of the original.

Rather than working from the ISR codebase, we build upon the newer VMNetX codebase, which spun off from the ideas of ISR and is the core of the Olive Archive project. VMNetX supports a subset of ISR's functionality. The main distinction is that its VMs are read-only. Therefore, we extend this codebase to support saving user state and transferring it between the client and the server as efficiently as possible. Although we did not introduce any completely novel techniques, we instead focused on producing the most robust and user-friendly implementation of the ISR core concept thus far. The final system prioritizes usability and the user experience. New features such as a GUI and trickle back of dirty state allow users to more easily manipulate their VMs and minimize waiting for operations to complete.

# Acknowledgements

I would like to thank everyone who has supported me in any way during my undergraduate and graduate journey.

Professor Satya: thank you for your patience and guidance through my three years with the group. I feel very lucky to have stumbled upon your work as an undergrad, and it shaped my experience at CMU in a way I could not have expected.

Benjamin Gilbert: you took a personal interest in my development as a student and as a person in the field in computer science, and I know our discussions will stay with me for the rest of my career. Thank you, and I look forward to hearing about your future work.

Professor Eckhardt: thank you for agreeing to be part of my thesis committee on such short notice. The last minute guidance you provided was invaluable in producing this document. I will keep my promise to read at least one thesis or conference paper every year.

I learned a great deal from working with the rest of the members of the research group: Kiryong Ha, Yoshi Abe, Jan Harkes, Wolfgang Richter, Wenlu Hu, Zhuo Chen, and Brandon Amos. Thank you for always being open to discussing my work and often times going above the call of duty to help me.

Thank you Tracy Farbacher for your help in coordinating the final steps of my graduation.

Finally, a big thanks to my parents for both the financial and emotional support throughout my five years at Carnegie Mellon. I appreciate you giving me the opportunity to better myself at great personal sacrifice, and I hope to return the favor tenfold in the future.

Da-Yoon Chung
Pittsburgh, Pennsylvania
July 2015

# Contents

**1 Introduction** **1**
1.1 The Current State of Internet Suspend/Resume . . . . . . . . . . . . . . 1
1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 The Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3.1 Scope of Thesis . . . . . . . . . . . . . . . . . . . . . . . 3
1.3.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3.3 Validation of Thesis . . . . . . . . . . . . . . . . . . . . . 4
1.4 Document Roadmap . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Redesigning ISR** **6**
2.1 Modern . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.1.1 Django . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Usable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 User-Focused . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 Off-The-Shelf . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4.1 HTTP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4.2 JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4.3 XML . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

**3 Extending VMNetX** **13**
3.1 The Olive Archive . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1 VMNetX . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.1 VM Storage . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.2 VM Access . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2.3 User Information Management . . . . . . . . . . . . . . . . 18
3.3 Versioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.1 Implementation Details . . . . . . . . . . . . . . . . . . . . 19
3.4 Server URLs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.5 Locking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.6 Cache Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.7 Dirty State Trickle Back (Background Upload) . . . . . . . . . . . . . 22
3.7.1 Live Migration . . . . . . . . . . . . . . . . . . . . . . . . 24
3.7.2 Modifying QEMU . . . . . . . . . . . . . . . . . . . . . . 25
3.8 Bandwidth Throttling . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.8.1 Throttle Control . . . . . . . . . . . . . . . . . . . . . . . 26
3.8.2 Throttling via curl: an anecdote . . . . . . . . . . . . . . . . 27

3.8.3 Dynamic Throttle Control . . . . . . . . . . . . . . . . . . . . 28

**4 Evaluation** **29**
4.1 Feature Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2 Simplicity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3.1 Trickle Back of Dirty State . . . . . . . . . . . . . . . . . . 31
4.4 Evaluation Summary . . . . . . . . . . . . . . . . . . . . . . . 33

**5 Further Feature Discussion** **34**
5.1 Thin-Client mode . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Disconnected Operation . . . . . . . . . . . . . . . . . . . . . . 34
5.3 Micro-Checkpoints . . . . . . . . . . . . . . . . . . . . . . . . 35
5.4 Encryption . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.5 Cloud Storage . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.6 Multiple Servers for a Single ISR Deployment . . . . . . . . . . . . 39
5.7 Content Addressable Storage (CAS) on the server . . . . . . . . . . . 40
5.7.1 Interaction with background upload . . . . . . . . . . . . . . 42
5.7.2 Hash Collisions . . . . . . . . . . . . . . . . . . . . . . . . 43
5.7.3 Garbage collection . . . . . . . . . . . . . . . . . . . . . . . 43
5.8 CAS on the Client . . . . . . . . . . . . . . . . . . . . . . . . 44

**6 Related Work** **45**
6.1 OpenISR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.2 The Olive Archive . . . . . . . . . . . . . . . . . . . . . . . . 45
6.3 Alternatives to ISR . . . . . . . . . . . . . . . . . . . . . . . . 45
6.3.1 Simple document editing . . . . . . . . . . . . . . . . . . . . 46
6.3.2 Multiple OSs on a single machine . . . . . . . . . . . . . . . . 46
6.3.3 Running applications not compatible with host OS . . . . . . . 46
6.3.4 Other Emulation Platforms . . . . . . . . . . . . . . . . . . . 46

**7 Conclusion** **47**
7.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
7.3 Final Thoughts . . . . . . . . . . . . . . . . . . . . . . . . . . 48

# List of Figures


1 Original hypothetical ISR deployment . . . . . . . . . . . . . . . . . . . . 1
2 OpenISR 2.0 GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3 Window for VM Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
4 Window for VM Execution . . . . . . . . . . . . . . . . . . . . . . . . . . 9
5 The Olive Archive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6 OpenISR 2.0 System Diagram . . . . . . . . . . . . . . . . . . . . . . . . 15
7 Effect of Background Upload on Resume and Checkin . . . . . . . . . . 23
8 QEMU Live Migration Data Format . . . . . . . . . . . . . . . . . . . . . 24
9 Processes involved in background upload . . . . . . . . . . . . . . . . . 26
10 Thin Client Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

# List of Tables


1 Comparison of features in OpenISR and OpenISR 2.0 . . . . . . . . . . . 29
2 Lines of code comparison between OpenISR and OpenISR 2.0 . . . . . 30
3 Effect of background upload on checkin time . . . . . . . . . . . . . . 32
4 Bandwidth use of background upload . . . . . . . . . . . . . . . . . . . 32

# 1 Introduction

## 1.1 The Current State of Internet Suspend/Resume

In June 2002, ISR was proposed as a novel approach to mobile computing which would eliminate the requirement that the exact hardware be physically carried with someone to access it. One of the original publications described a hypothetical world where anonymous hardware was widely available [20]. One reason that this prediction was not entirely unfounded was due to increasingly lower hardware costs. It was therefore not farfetched that for example, a coffee shop would provide laptops for customers to use for a few hours at a time while they are at the store. Obviously, such a loaner machine would not have any of the user's personal data stored on it for privacy reasons. With ISR installed however, an individual could access his or her personal machine state on this device quickly and securely. ISR would handle synchronizing user state across the various machines that a user might use in their lives, from personal and work machines to these anonymous devices.

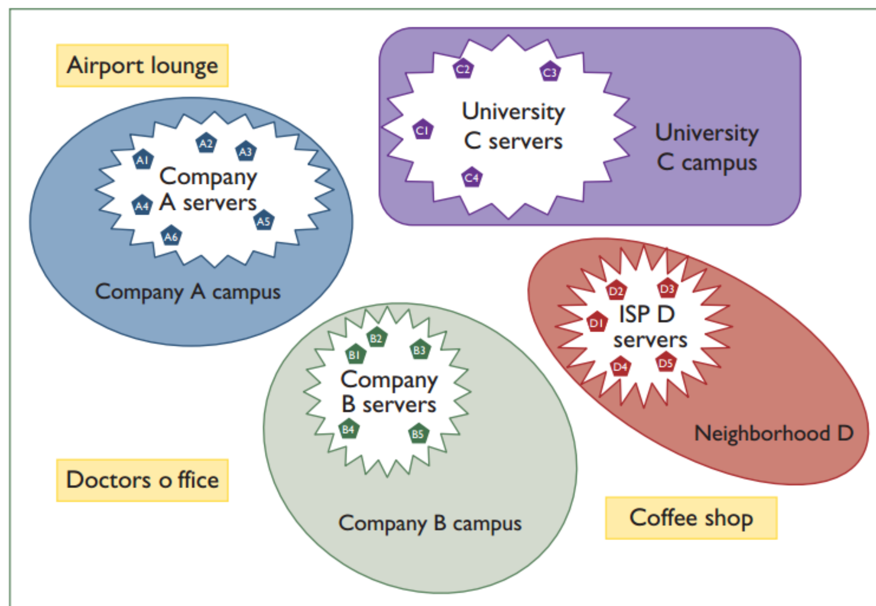



Figure 1: Original hypothetical ISR deployment

Figure 1 is a diagram from one of the first ISR publications [20] which presents a hypothetical ISR deployment. Each one of the colored regions represents a private ISR

installation that a larger entity like a company or a university would provide for its employees or students. On the fringe, there are a number of highlighted locations, such as the aforementioned coffee shop, which an ISR user from any of the larger groups may transiently visit. Users could use ISR client software installed on any of the anonymous hardware provided by these locations to access their VMs stored on their respective servers. Later on, Pocket ISR was developed so that even if these anonymous devices did not have ISR pre-installed, a portable client could be carried around in a USB device [18].

Presently, the world has not yet panned out in this manner. Instead of cheap personal computers being widespread, computing has shifted towards increasing portability. Mobile computing is clearly on the rise with the popularity of smartphones, tablets, and ultra-portable laptops, and many individuals even own multiple such devices. As a consequence of the increasing development of the Internet, many of the most commonly used applications such as word processing and email have been moved to web applications, making them available anywhere with an network connection. It is fair to concede that the average user, who only uses his or her computing devices for simple tasks like browsing the web or checking email, would probably not feel motivated to go through with the overhead of installing and using ISR.

## 1.2 Motivation

Despite these developments, a niche still exists for ISR. There are many resource-intensive applications which require the resources of a full-size machine, such as various image and video manipulation tools or 3D modeling programs. With ISR, these and many other applications can be made available in a portable environment.

Also, most, if not all, of the web services rising in popularity today such as Gmail [6] and Dropbox [5] are owned by private corporations. Because it is not practical for most individual users to set up their own email services or personal clouds, people are forced to trust large companies with their sensitive data in order to use their products. The extent to which these entities can and should be trusted is a discussion outside of the scope of this work. Regardless of the answer, an individual can use ISR to set up a personal server on a small amount of hardware, and still maintain full control of their personal data. This is an advantage which may decrease or increase in importance in the future as many truths about the actual security of our data in the hands of these big companies are revealed.

Because ISR is based on virtualization, it also provides all of the now widely appreciated benefits of using VMs. Users can run operating systems and applications that would not natively run on their host machine. Checkpoints for backing up data can be easily created via image snapshots. VMs can be shuffled between host machines in order to balance load. These and many other advantages exist.

The primary use of ISR in the current computing ecosystem is authoring. A user who needs to access a specific version of an application on a specific operating system can do so without possessing the extra hardware or time to make a physical installation. For example, a user with only an Ubuntu machine can potentially use ISR to run Photoshop on a Windows 7 VM after installing only the ISR client. A user who owns a tablet with limited processing power can access heavy applications for any OS using a thin-client.

By focusing on a minimal feature set and optimizations which reduce the amount of time that the user spends waiting for ISR to complete operations, ISR provides these benefits to users while being as close to the experience of using a physical machine as possible.

## 1.3 The Thesis

The goal of this research is to redesign and implement a new iteration of the original ISR concept. Instead of working from the previous code and working around prior decisions and various technical debt, the thesis builds upon a more recent code base, allowing it to prioritize features that were previously unimplemented but later recognized as essential. The thesis statement is thus:

> *ISR can be modernized and made more usable by redesigning and rebuilding it. By learning from the lessons of past iterations of the software, the architecture and codebase can be built with user-focused priorities in mind while taking advantage of off-the-shelf technologies, instead of modifying existing code to accommodate new features.*

### 1.3.1 Scope of Thesis

This thesis describes a complete overhaul of the existing ISR project. It is not intended to be a one-to-one port of the existing implementation, but a complete redesign of the original idea to best fit the current computing status quo. The thesis makes the following assumptions about ISR and future deployments.

- The servers running ISR are trusted, in terms of hardware and administrator.
- Minimizing storage cost on both the clients and the servers is not a priority, as storage is assumed to be relatively abundant and cheap to increase.
- Network bandwidth and speeds are a priority, in that it is important to reduce download and upload sizes. Upload bandwidth is assumed to be a more scarce resource than download bandwidth, as the two values are usually not symmetric in commercial environments.

### 1.3.2 Approach

The biggest priority when considering the new ISR system was the quality of the user experience. Due to the large amount of experience that people involved with the original project have accumulated over the years, there was plentiful feedback that was considered in the process. The guiding principles of the thesis can be enumerated as follows.

- Minimize the amount of time that the user spends waiting for anything, from the boot time of a VM to the final checkin of their VM back to the server.
- Minimize the amount of time the user spends looking for anything, whether it is information about their VMs or the operations available to them.
- Do fewer things better. Support a minimal set of features, but make sure they are correctly implemented and more importantly are easily extensible.

### 1.3.3 Validation of Thesis

The thesis is based upon a well-established body of code whose objectives are well-defined. The new system designed in this thesis can therefore be compared to the previous iteration in terms of simplicity and feature set. Empirical measurements are used to quantify the effect of optimizations.

## 1.4 Document Roadmap

The rest of this document consists of six different chapters. Chapter 2 describes the process taken in considering which aspects of ISR are the most important to carry forward and which flaws and mistakes should be avoided or rectified in a new implementation.

Chapter 3 details the more recent codebase that OpenISR 2.0 is built from and the technical additions that were made to support the features from chapter 2. Chapter 4 evaluates the new features that were outlined in chapter 3. Chapter 5 is a discussion of features that were seriously considered and reasoned about in the process of designing OpenISR 2.0, but were never implemented. Chapter 6 goes over related work and some of the more recent developments in VMs, cloud, and web applications which maybe serve a similar purpose to some aspect of ISR. Finally, the concluding chapter describes contributions and future work of this thesis.

# 2 Redesigning ISR

Very early on in the design process, it was decided that OpenISR 2.0 should be built by extending a more recent codebase rather than the previous implementation of ISR, OpenISR [10]. Therefore, instead of cleaning up or optimizing existing, completely functional features, the process began by choosing which features to reimplement or implement completely differently. Again, the principles that were emphasized were making OpenISR 2.0 more modern, usable, user-focused, and understandable in terms of the protocols and file formats used.

## 2.1 Modern

OpenISR 2.0 is made more modern by taking advantage of recently popularized programming languages and frameworks to replace some of the older proprietary code in the original implementation. One of the major benefits of using existing tools instead of building custom ones is that there is an abundance of resources already available online. This makes the initial implementation of OpenISR 2.0 much faster to develop, and hopefully also lowers the barrier of entry for future collaborators.

### 2.1.1 Django

Django [4] is an open source web application platform which is used very widely on both small and large scale web applications. A simple web search can provide a list of many very popular companies which use it to build their sites. However, one of the reasons for its popularity among individual developers is that it is relatively simple to bootstrap and provides lots of tools for testing, most notably a lightweight development server. This allows the developer to test code without having a dedicated web server. Django web applications are written entirely in Python, and it conveniently integrates commonly used functionality such as security, databases models, and authentication into modules.

The Olive Archive [9], to be revisited later, uses Django as its server for both managing user information and serving VMs, so it was decided that Django would be sufficient for building a server for OpenISR 2.0 as well. It is expected, although not proven, that the Django-based server would not have issues with scalability for a reasonably-sized user base.

## 2.2 Usable

Usability of a system can be measured in different ways. For ISR, the goal was to simplify the user experience in terms of issuing commands and gathering information VMs.

All prior iterations of ISR have a command-line interface. The UI proved to be lacking in clearly conveying all of the commands and options actually available to users. For OpenISR 2.0, we designed a client GUI (see Figure 2). We conveniently place all the most commonly used actions on a toolbar and relegate less frequently-used actions to the menu bar. Users should be able to glean all essential information about the state of a VM, such as its current version number or whether it has local changes, without navigating to find it.

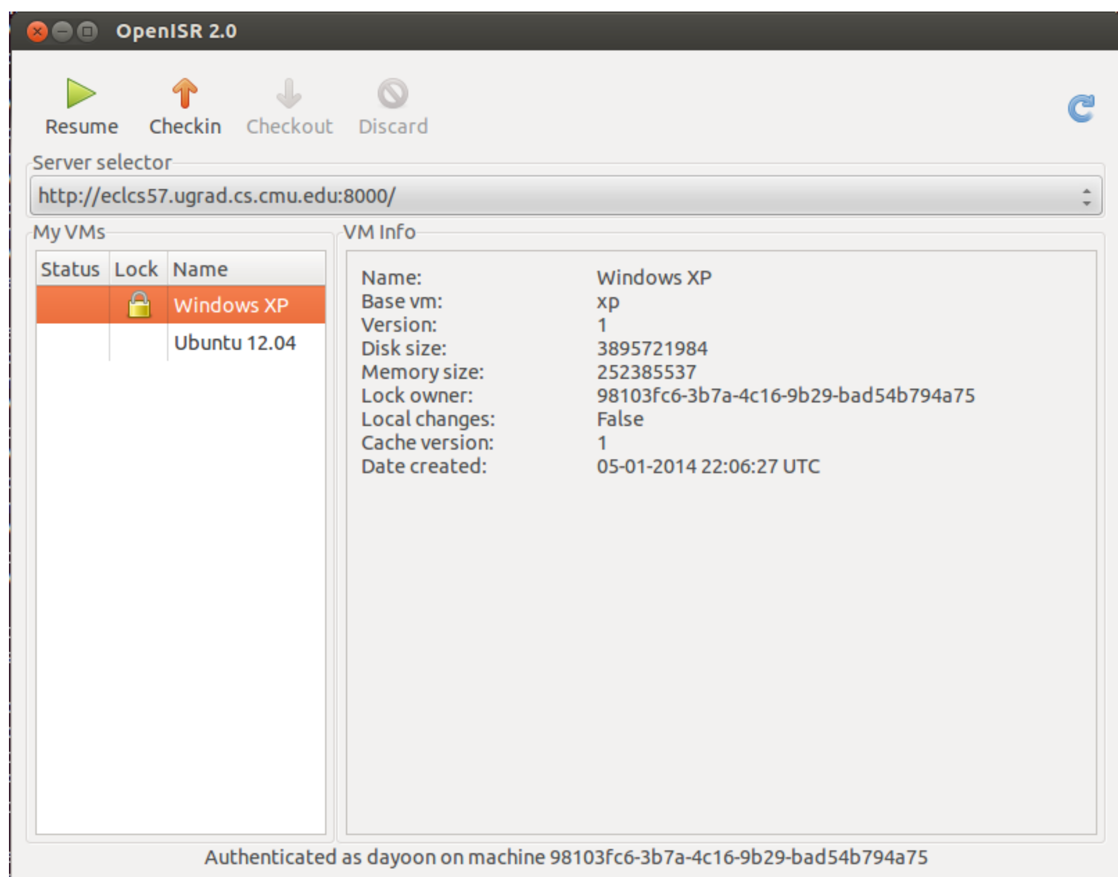



Figure 2: OpenISR 2.0 GUI

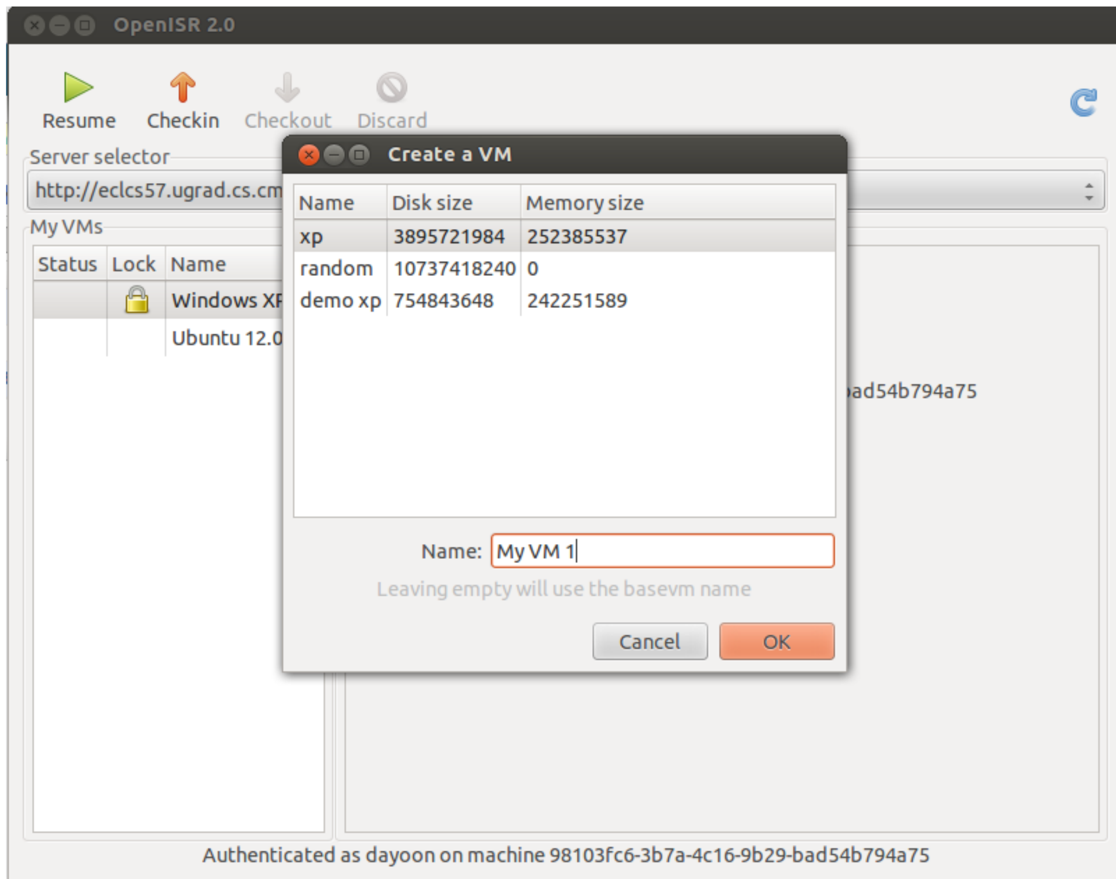



Figure 3: Window for VM Creation

## 2.3 User-Focused

One difference in how Django is utilized by the two projects is that Olive actually has a user-facing web UI at www.olivearchive.org. This website provides information about the project as well as displays information about the available VMs. Users also have the ability to launch VMs directly from a its webpage and then run the client to view and interact with the VM locally. The one major caveat about this design decision is that the user must be able to access to the Olive Archive website to be able to run any VMs. This is an acceptable expectation for Olive because all of the VM images are fetched on-demand from the server in order to execute them anyway, so a user who could not access the website would not have been able to access any VMs regardless.

However, a user of ISR can be reasonably be expected to want to use their VMs

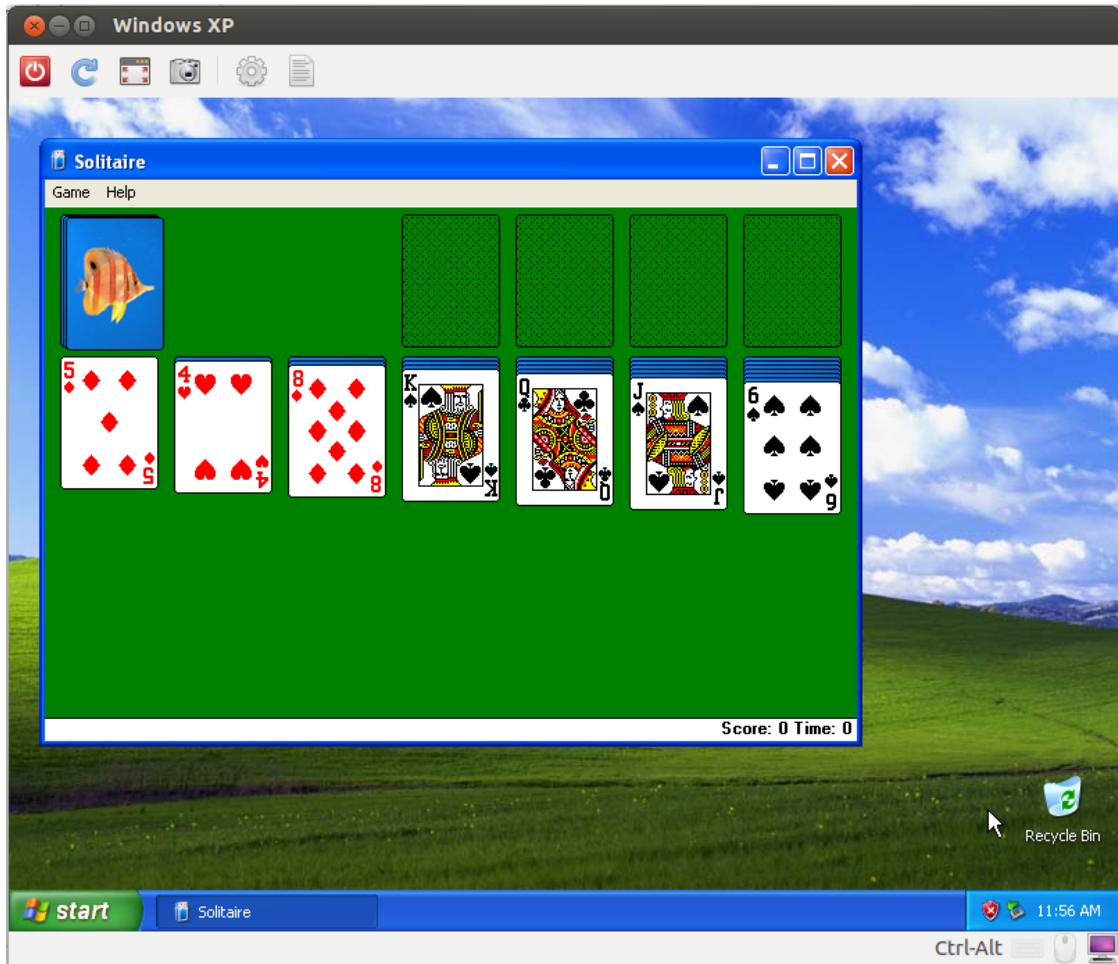



Figure 4: Window for VM Execution

without an Internet connection. For example, he or she might want to work a VM during a long flight, and then have the ability to save modifications to the server once landing. In order for users to be able to continue using their VMs in a disconnected environment, it is important that a web based-UI is not the sole method of accessing VMs, although one could most certainly be added later through Django.

## 2.4 Off-The-Shelf

When designing OpenISR 2.0, it was important to make as many of the components as easily debuggable and understandable as possible. One way to accomplish this goal is to take advantage of well-known protocols and file formats instead of creating custom ones.

### 2.4.1 HTTP

In the original ISR, the server is written in Perl. It requires specific configuration and installation and uses a custom protocol which allows it to communicate only with the ISR client. The Olive Archive server with which VMNetX communicates is a generic HTTP server built using the Django framework for Python. VMs are fetched on demand via regular HTTP requests which contain range headers that specify offsets in the memory and disk images that the client requires. Debugging the interactions between the client and the server is also more straightforward because of this design decision, as tools like WireShark [16] were able to be used to monitor traffic and identify anomalies.

### 2.4.2 JSON

JSON or Javascript Object Notation is a human-readable data format commonly used in web applications to exchange data. Its appearance is very similar to nested dicts in Python and essentially consists of a series of nested parentheses.

OpenISR 2.0 uses JSON for all non-chunk transfers of information between the client and the server. For example, information about a particular user's VM might be transferred in the following JSON object.

```
{
  "32b832b3-3a43-4fb4-ae4e-d8a2e6721885": {
    "Base vm": "ubuntu server",
    "Cache version": 1,
    "Date created": "07-20-2015 14:02:58 UTC",
    "Disk size": 1566900224,
    "Key": "",
    "Local changes": false,
    "Lock owner": "98103fc6-3b7a-4c16-9b29-bad54b794a75",
    "Memory size": 1090961408,
    "Name": "1",
    "Status": 4,
    "Version": 1,
    "uuid": "32b832b3-3a43-4fb4-ae4e-d8a2e6721885"
  }
}
```

### 2.4.3 XML

XML or Extensible Markup Language is another commonly used language for web-related applications. OpenISR 2.0 uses XML for all configuration files, namely the libvirt configuration files and the VMNetFS configuration files. VMNetFS is a component of VMNetX, a core part of the implementation, which interacts with the VM's file system. The following is an example of a VMNetFS configuration file.

```
<?xml version=’1.0’ encoding=’UTF-8’?>
<config xmlns="http://olivearchive.org/xmlns/vmnetx/vmnetfs">
  <image>
    <name>disk</name>
    <size>1566900224</size>
    <origin>
      <url>http://localhost:8000/vm/<uuid>/1/disk/chunk</url>
      <offset>0</offset>
      <validators>
        <last-modified>1437395934</last-modified>
        <etag>etag</etag>
      </validators>
    </origin>
    <cache>
      <pristine>
        <path>/home/dayoon/.cache/vmnetx/chunks/<uuid>/disk/131072</path>
      </pristine>
      <modified>
        <path>/home/dayoon/.local/share/vmnetx/chunks/<uuid>/disk/131072</path>
      </modified>
      <chunk-size>131072</chunk-size>
    </cache>
```

```
    <fetch>
      <mode>demand</mode>
    </fetch>
    <upload>
      <checkin>0</checkin>
      <rate>0.01</rate>
    </upload>
  </image>
  <image>
    <name>memory</name>
    <size>1090961408</size>
    <origin>
      <url>http://localhost:8000/vm/<uuid>/1/memory/chunk</url>
      <offset>0</offset>
      <validators>
        <last-modified>1437395934</last-modified>
        <etag>etag</etag>
      </validators>
    </origin>
    <cache>
      <pristine>
        <path>/home/dayoon/.cache/vmnetx/chunks/<uuid>/memory/131072</path>
      </pristine>
      <modified>
        <path>/home/dayoon/.local/share/vmnetx/chunks/<uuid>/memory/131072</path>
      </modified>
      <chunk-size>131072</chunk-size>
    </cache>
    <fetch>
      <mode>stream</mode>
    </fetch>
    <upload>
      <checkin>0</checkin>
      <rate>0.01</rate>
    </upload>
  </image>
</config>
```

In addition to being human readable, XML allows the code to define the specifications of these files to check for correctness. The high-level component of VMNetX, which is written in Python, generates a XML configuration file for VMNetFS, which is written in C. VMNetFS uses a C XML parser library to read and interpret the configuration settings during initialization.

# 3 Extending VMNetX

## 3.1 The Olive Archive

The Olive Archive [9] is a more recent project from Professor Satyanarayanan's group which focuses on archiving executable content in VMs. The driving idea is that old hardware can not be preserved indefinitely due to physical degradation caused by time. Old software which can run only on specific hardware which has since been discontinued is therefore at risk of disappearing along with the hardware. The Olive Archive is an effort to make sure this is not the case, by preserving software and operating systems inside VMs. As an extra step, these VMs are made accessible and easily executable to users through the Olive website, using software called VMNetX.

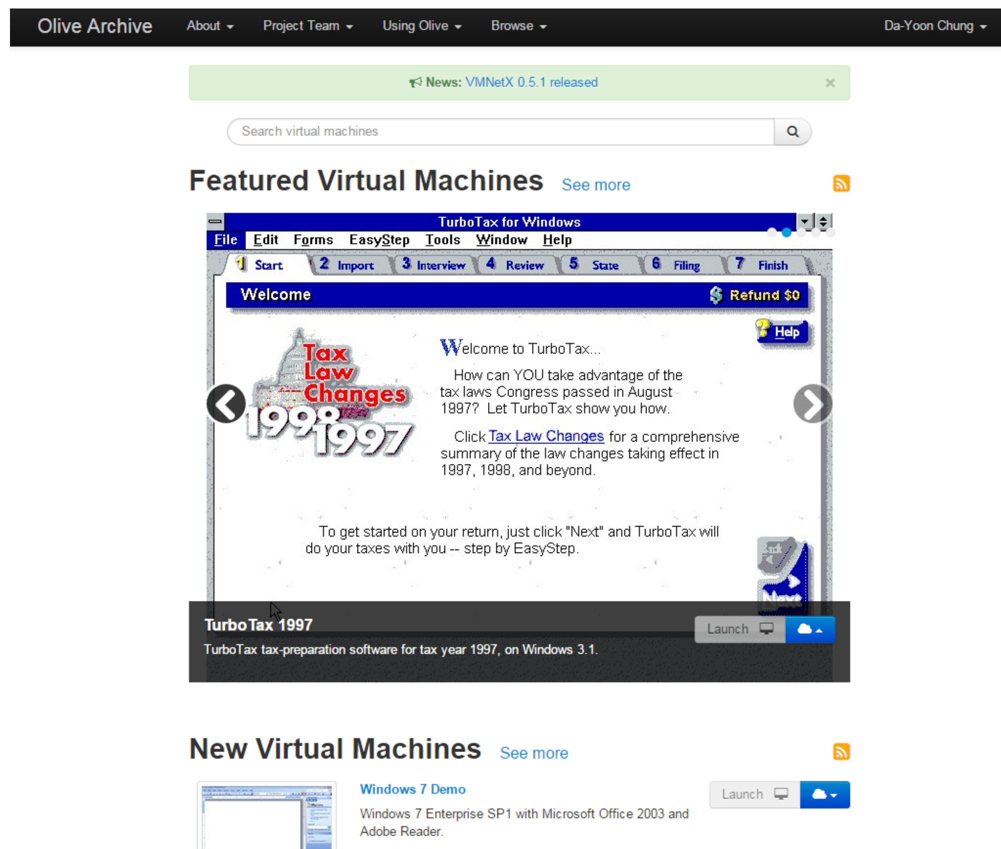



Figure 5: The Olive Archive

### 3.1.1 VMNetX

VMNetX [14] is a codebase partially derived from OpenISR's *parcelkeeper* module. It supports the execution of VMs being fetched on-demand like ISR, but with a number of significant differences. First, all of the VM metadata and chunks are fetched from an ordinary HTTP server written using the Django instead of a proprietary server with a custom protocol. It also differs in that the VMs that the users interact with are read-only, so any and all changes that someone makes to a VM are discarded at the end of execution. This decision makes sense in the context of the use case of Olive; users of a very old VM, such as the TurboTax 1997 VM displayed in Figure 2, are not likely to want to save their progress with the VM because the program does not have many practical uses in 2015. However, if we want to adapt VMNetX to be a part of ISR, the lack of the ability to save user changes is a huge shortfall because saving user state is a key feature.

As previously mentioned, VMNetX has a simpler design because its VMs are read-only. For each VM, the server stores a whole disk and memory image and delivers the arbitrary byte ranges that clients requests by reading from the images at the correct offsets. The client creates sparse disk and memory images and populates these by making server requests as different byte ranges are accessed during execution.

In order to efficiently create and save small incremental changes to the disk and memory images so that ISR can support features like versioning, ISR must split up individual image files into smaller ordered chunks. VMNetX was therefore modified to request chunks from the server by calculating the corresponding chunk numbers of the desired byte range. It was also changed to write out its modifications to the images downloaded from the server into individual chunks which are saved on disk, so that they persist across VM runs, whereas the original VMNetX would discard these changes.

OpenISR 2.0 stores its disk and memory images in chunks of 128 KB, a size which performed well for the previous iterations of ISR. If the chunk size is too large, bandwidth will be wasted since whole chunks are requested for even small ranges of bytes requested. On the other hand, if the chunk size is too small, the amount of IO between the client and the server from making more individual chunk requests can slow the system down. Determining the optimal chunk size is outside the scope of this project, but can be evaluated in the future, as the code is written to be agnostic to the chunk size, which is simply set in a configuration file.

Obviously, since ISR needs to store user modifications on the server, VMNetX had to be given the ability to upload the modified chunks to the server. Because of the decision to use a HTTP server, OpenISR 2.0 is able to use the same approach that VMNetX

uses to fetch chunks from the server, namely using curl [3] to make HTTP GET requests. Additional routines which iterate through the modified cache and similarly make HTTP PUT requests to the server containing individual chunks were added.

## 3.2 Server

As previously mentioned, one of the goals of OpenISR 2.0 is to take advantage of as many generic protocols as possible so that it is more easily understood and debugged. Using Django to implement the server is in accordance with this objective.

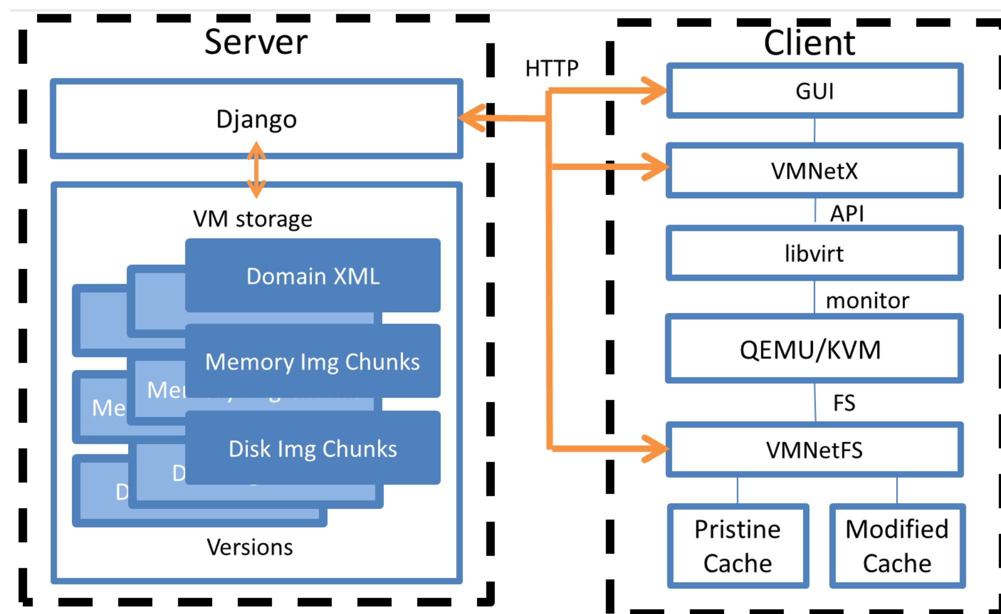



Figure 6: OpenISR 2.0 System Diagram

Figure 6 is a diagram of the client and server breakdown of the system. The major client components are all adopted from the original VMNetX code, with the exception of the GUI. VMNetFS is modified in order to support uploading chunks to the server.

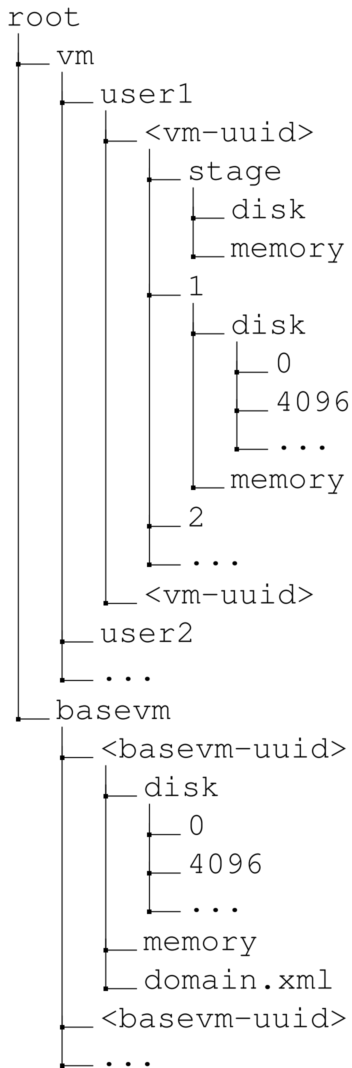
### 3.2.1 VM Storage

VMs are stored locally on the server, for ease of access and simplicity. Like the Olive Archive server, a VM consists of a disk image, a memory image, and a domain XML file,

which is required by QEMU to resume the VM. As previously mentioned, a change in OpenISR 2.0 from Olive is that the image files are stored as numbered chunks rather than single complete files. Each directory of chunks is divided into subdirectories of 4096 chunks to prevent individual directories from holding potentially millions of chunk files and becoming difficult to navigate.

ISR also makes a distinction between a “base VM” and a regular VM that a user owns. These base VMs represent an unmodified standard from which users can make their personal changes. Many users can have personalized VMs all stemming from a single base VM. As an example, a base VM might consist of a vanilla Windows [8] or Ubuntu [12] installation with no other modifications or a suite of software. Different users would make unique modifications to these base VMs. The server can also achieve significant storage savings with this approach, since the first version of every VM can point to the corresponding base VM.

The base directory of VMs and base VMs on the server is structured as follows:




The `stage` directory contains chunks that have been staged on the server but not yet committed into a new version. These are chunks that have been uploaded during background upload or during checkin.

Disk and memory chunks are split into directories of 4096 chunks simply for organizational purposes.

### 3.2.2 VM Access

VMNetXs primary mode of operation on the client side involves requesting individual chunks using HTTP GET requests. Each chunk has a unique URL defined using Django's URL dispatcher. Domain XML files also have unique URLs. One advantage of using a web server is that these objects can be fetched using a browser or any HTTP client and not just the ISR client for debugging or testing purposes.

VMs are not only fetched but also uploaded to the server on an individual chunk basis. As an unfortunate side effect, even if the client knows it want to download or upload a byte range spanning ten chunks in advance, it must request each of these ten chunks individually. Future work in OpenISR 2.0 would take advantage of commonly used techniques such as batching or streaming to avoid unnecessary I/O.

### 3.2.3 User Information Management

Django provides database integration and user models to allow the server to easily track user accounts and permissions. Individual users, VMs, and VM version objects are all stored in their respective models which allow the server to easily manage and retrieve this information without having to directly query the database, which ISR previously had to do.

## 3.3 Versioning

Versioning refers to the ability to save checkpoints or "versions" of the VM that the user can resume from at any time. A checkin operation to the server creates a new version. Naively copying the disk image and saving the entire memory image required to make a checkpoint is not difficult using existing functionality in QEMU, which can write out the a memory snapshot to a file once a VM is suspended. However, the challenge comes from creating these checkpoints quickly and efficiently with regards to storage. A modern VM can easily have a disk image tens of gigabytes in size and a memory image also within that range. Storing whole disk and memory images for each version on the server would scale very poorly, and uploading this quantity of data from the client to the server would take a huge amount of time and bandwidth.

Versioning has been identified as a core feature of ISR which gives it a distinct advantage over physical machines. In any machine running a modern operating system, only a single logical version of the machine can be running at one time. Consider a sce-

nario where a user has put many hours of work into a project only to discover that one of the files was corrupted a few days ago. The user has luckily had the foresight create a backup of the system a week ago. At this point, the user can restore the system from the backup, but doing so will erase any work done since that point. With ISR, the user can easily launch two VMs, one executing at the current state and another at the point of restoration, and easily make visual comparisons of the changes made between the two points in time and preserve whatever modifications are most important. It should be noted that versions within ISR are currently only created when the user manually issues a checkin command. Therefore, any modifications that a user makes since their last checkin is not backed up on the server.

### 3.3.1 Implementation Details

Each of a user's VMs is derived from a base VM. A user requests a VM to be created from a base VM through the UI as shown in Figure 4 above. When this request reaches the server, the server sets up the VM directory for the user. The first version of the VM is version 1, which is identical to the chunks and domain XML file from the base VM. Any future checkins create higher version numbers.

An important point to note is how the first version is created. Since the first version is again identical to the base VM, it would be a waste of storage to copy over each of the individual chunk files every time a user wanted a new VM. Therefore, the server links the base VM files to the version 1 files. This is done on the server by the following code snippet:

```
version_dir = os.path.join(vm_dir, str(vm.current_version)
src = os.path.join(settings.STORAGE_DIR, 'basevm/%s' \
    % basevm.uuid)
os.symlink(src, version_dir)
```

The last line creates a symlink from the base VM directory to the directory of the first version of the VM. Because symbolic links point to file names, rather than the file data itself, they are not reference counted. Therefore, even in the unlikely case where a base VM is pointed to by an extremely large number of VMs, there will not be any issues with reaching the maximum number of hard links possible.

Versions after the first consist of directories containing only the chunks that have been modified since the last version. Through this design, the chunks follow a "copy on write" behavior, in that any chunk is only written again when it is modified.

Consider an example where the client requests some chunk *n*. Chunk *n* is not guaranteed to be present in every version, only those in which it was modified. The chunk is retrieved by the server by iterating through the version directories in decreasing order and returning the first chunk file found.This is guaranteed to be the latest modified copy of the chunk, because the search starts from the latest version and works backwards, and chunk *n* is guaranteed to exist for at least one version because the first version is again identical to the base VM, which contains all of the chunks. In other words, if a chunk is never modified, it will be retrieved from the first version (base VM).

Currently, ISR does not have the ability to delete an intermediate version. This would require lots of book-keeping to determine which chunks from which versions must be kept such that the future versions are not corrupted. This is a limitation that could be addressed in the future.

## 3.4 Server URLs

The following is a breakdown of all of the information available to the ISR client (or any HTTP client) through basic HTTP requests. In order to make sure private information of VMs is not mishandled, each request must contain the secret key of the user whose information is being request. If the secret key is missing, the request is rejected immediately by returning a 404 response code.

<server-url>/vm/info
: Returns a JSON object containing a list of information about all of the user's VMs.

<server-url>/vm/base_info
: Returns information about all of the base VMs available on the server. This information is required by the client when the user wants to create a new VM.

<server-url>/vm/checkout/<uuid>
: Check out the VM corresponding to the passed UUID.

<server-url>/vm/create
: Create a new VM for the user from an argument base VM uuid passed in the body of the request.

<server-url>/vm/<uuid>/update
: Update some metadata about the given VM.

`<server-url>/vm/<uuid>/<version-num1>/<version-num2>`
: Return a JSON object containing a list of the chunks modified between version 1 and 2 for rolling back.

`<server-url>/vm/<uuid>/<version>`
: Checkout the VM and acquire the lock, or checkin the VM and set up the directories for the next version.

`<server-url>/vm/<uuid>/<version>/<image>/size`
: Return or update the size of the disk or memory image.

`<server-url>/vm/<uuid>/<version>/<image>/chunk/<chunk-num>`
: Return or upload a chunk for the disk or memory image.

`<server-url>/vm/version/<uuid>`
: Return information about the version history for a given VM.

`<server-url>/vm/commit/<uuid>`
: Commit the changes on the server to the VM (from the staging area).

`<server-url>/vm/comment/<uuid>`
: Modify the comment for the next version to be committed.

`<server-url>/vm/discard/<uuid>`
: Remove the chunks in the staging area.

All manipulations to the VM on the server can be done through a simple HTTP request, as shown above.

## 3.5 Locking

Locking is a mechanism which prevents race conditions when ISR is used on multiple machines by a single user. In the current implementation, only a single machine can own the lock to a VM at one time. The lock itself is implemented at a randomly generated UUID that the ISR client generates the first time it is executed on a machine.

Only the owner of the lock can resume the VM and subsequently checkin any modifications to the server. This prevents situations where, for example, two client machines have different modifications since the most recent version, and both machines try to checkin at the same time. The user also has the option to forcibly release the lock owned by a machine if the machine cannot be accessed to release the lock on it, for example, if it is stolen or destroyed in flood. At this point, any local changes on the previous owner

machine must be discarded, but this ensures the user is aware that his or her actions have this consequence.

The lock is acquired when a VM is checked out given that it's available, and released when the VM is checked in. The state of the lock is stored as a field in the Django model for a VM and therefore persists across server crashes, as long as the database is intact.

## 3.6 Cache Validation

Cache validation is another feature which prevents race conditions and data corruption when going between different machines and/or versions of a VM. For example, the user may have a version of a certain VM on one machine which is older than the most recent version of the same VM on a different machine. When this happens, the system must ensure the state of the pristine cache on both machines is identical in order to prevent invalid disk and memory chunks from being read.

To address this issue, the client device must validate its disk and memory image cache before executing a VM. The client machine tracks which version $n$ of the VM its caches correspond to at all times. When a user requests that a VM be resumed to version $m$, the client requests that the server send it a list of all the chunks that were modified between version $n$ and $m$. The client can then discard of these chunks from its pristine caches such that they are re-downloaded from the server to make sure they are up to date. This approach ensures only the chunks that could have potentially changed between the versions are re-checked and the rest are preserved, minimizing startup time and communication with the server.

## 3.7 Dirty State Trickle Back (Background Upload)

As previously mentioned, the focus when considering which features and optimizations to implement is always improving the user experience. At the highest level, this involves reducing the amount of time that the user is waiting for anything, unable to do any actual work until ISR completes an operation. In the previous iteration of ISR, the step which took the longest amount of time by far was checkin. When VM images can be on the order of gigabytes, uploading these to the server can potentially take hours, during which the user is unable to interrupt ISR or the Internet connection.

In order to remedy this flaw, OpenISR 2.0 implements the background uploading of the modified state of the VM while it is resumed by the user. Without this feature in

place, all of the traffic between the initial resume and checkin operation was downstream from the server, in the form of chunks being fetched on-demand during execution. By beginning the upload of modified chunks as soon as the VM is resumed, ISR frontloads the heavy checkin operation. In the optimal case, ISR can potentially upload all of the dirty state to the server by the time the user suspends the VM, and the checkin operation can be finished almost immediately.

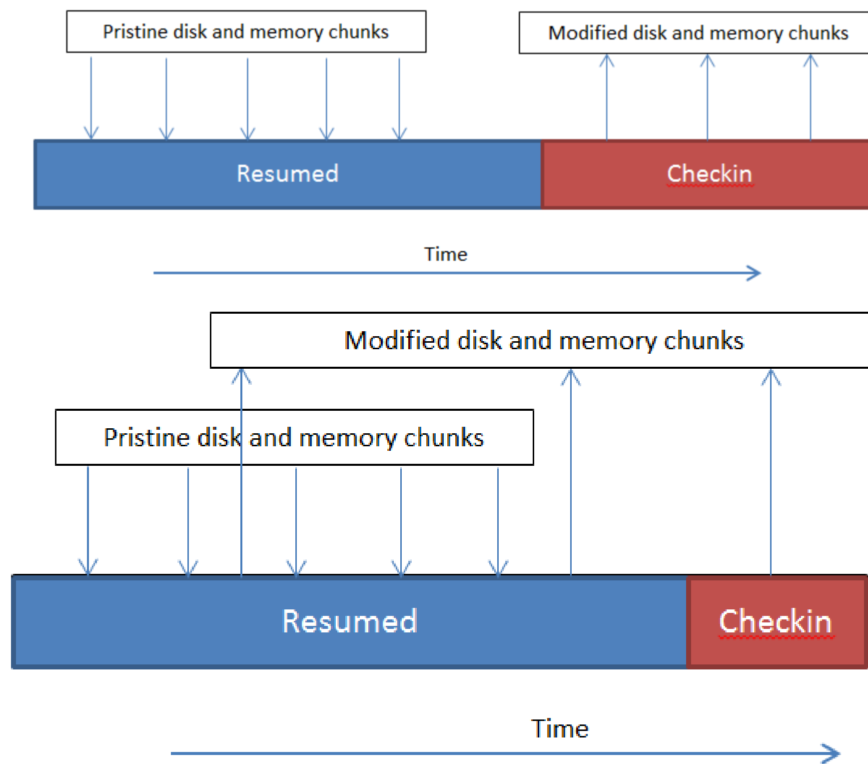



Figure 7: Effect of Background Upload on Resume and Checkin

It was known that this feature would require the heaviest modifications to ISR in order to support, because unmodified QEMU does not expose the memory image, which ISR must read in order to determine which parts of it were dirtied. Therefore, without making some modifications to QEMU itself, it would be impossible to determine which modified bytes to upload in advance to the server.

### 3.7.1 Live Migration

QEMU supports an operation called "live migration", which was leveraged in order to expose intermittent memory snapshots to ISR. Live migration is used in order to migrate a VM between two different host machines without having to suspend the VM on the source machine before initializing the transfer. The user therefore experiences very brief or no interruptions to his or her running VM. The steps that live migration takes when migrating a VM running on machine A to machine B can be outlined as follows:

1. Initiate live migration on machine A
2. Take snapshot of current memory image on A and transfer the snapshot to B while the VM is still running on A by using copy on-write
3. Determine which chunks which have been modified on A since the last snapshot and the time that the transfer finished (called an "iteration")
4. Repeat step 3 until the amount of modifications is small
5. Suspend execution on A
6. Transfer final memory modifications and transfer control to B

With access to the raw migration data being sent from machine A to machine B, the client could reconstruct a memory snapshot from the data, all without interrupting the user's VM. There would obviously be a performance penalty, but it was decided that the benefits of reducing checkin time outweighed any such tradeoff.

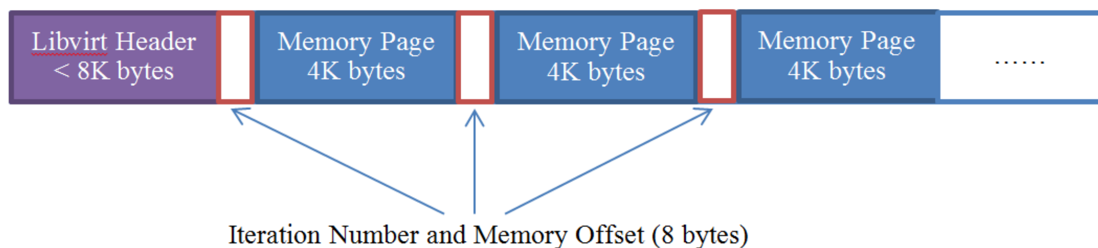



Figure 8: QEMU Live Migration Data Format

It should be noted that the memory snapshot produced by live migration in this manner is always the full size of the memory image as defined in the libvirt XML, because it is the raw memory image. There is no compression or deduplication done.

### 3.7.2 Modifying QEMU

Elijah-QEMU is a custom QEMU implementation created by another member of Professor Satya's group, Yoshihisa Abe, for a different project. It allows live migration to be manually iterated (step 4 above), and can write the live migration data to a named pipe, instead of transferring to another host machine. When the *save* operation for QEMU is initiated, instead of suspending the VM and writing its memory snapshot to a given file path, *save* now initiates a memory dump of the live migration format to a given file path, which it treats as a named pipe. ISR can read the raw migration format from the pipe and assemble a memory-snapshot from which QEMU can resume.

Elijah-QEMU makes a modification to the execution path of a restore operation, which requires a memory snapshot and resumes the VM to the state at which it was suspended. In the original QEMU, when resuming, the entire memory snapshot is copied into a newly malloc'ed region of memory before the VM is resumed. All future reads and writes into memory are done into this memory.

The original use of Elijah-QEMU required being able to capture incremental memory reads from the original memory snapshot, which is not possible if the entire memory image is initially read in. Therefore, Elijah-QEMU replaces the malloc and copy of the image with a mmap.

When ISR was first executed with this modified QEMU, this subtlety was not known. It was noticed though that as an unexpected side effect, ISR VMs would begin executing almost immediately from a cold resume, because the memory image was now being demand-fetched in addition to the disk image. Without the mmap change, the entire memory image is fetched from the server at resume if not present.

Unfortunately, due to an issue with version 3.20 of the Linux kernel running on the machine that was used to develop OpenISR 2.0, this mmap-malloc change had to be reverted. Since ISR is directly manipulating the memory image, when VMNetX attempted to open the memory image file to read and write pristine and modified chunks to it, a kernel panic occured because mmap was also holding an open file pointer to the memory image. This issue disappeared when the memory image was read to a malloc'ed region instead of the mmap.

The modified QEMU receives commands to begin, iterate, and terminate live migration via QMP, the QEMU message passing protocol. QMP is enabled in the configuration file passed to libvirt, after which JSON messages can be passed to the running QEMU process via QMP. This is used to allow ISR to control the frequency of iterations

of live migration.

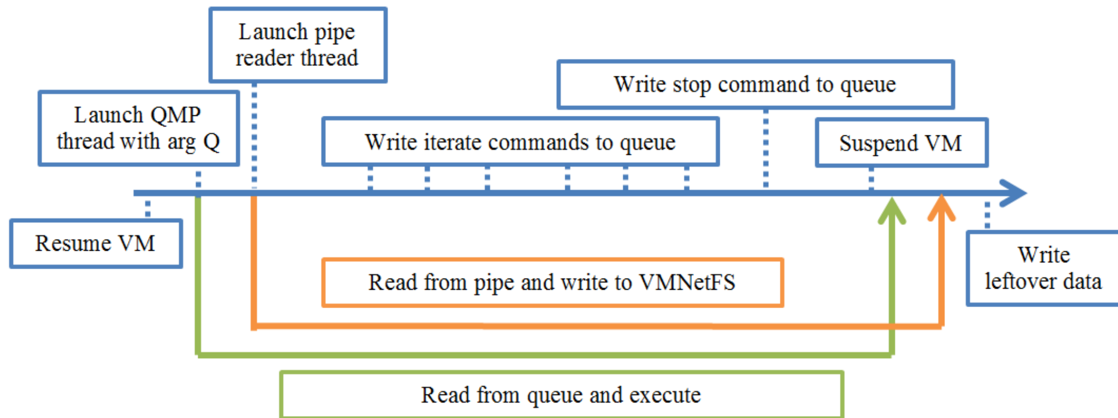



Figure 9: Processes involved in background upload

## 3.8 Bandwidth Throttling

With the addition of the background uploading feature, it was important to give the user some control over how much bandwidth ISR is allowed to consume for trickling back state to the server while the VM is resumed. A similar option is not provided for download bandwidth because the rate at which chunks are downloaded is directly related to the user experience; because chunks are fetched on demand, delays in downloading are manifested as stuttering or freezes. On the other hand, the benefits of uploading dirty state in the background are not tangible to the user until checking in, at which time the amount of state that must be uploaded to the server is hopefully reduced.

### 3.8.1 Throttle Control

Two different ways of presenting the option to throttle uploading to users were considered. The first approach is to allow users to define an absolute amount of bandwidth e.g. 500 kbps that ISR is allocated for background upload. This approach has the benefits of being very easy to implement. Chunks are uploaded using the curl library, which conveniently has a bandwidth limiting option for individual requests. This option accepts an absolute value. Therefore, ISR would simply have to pass this value in the configuration XML for VMNetFS, and set it as the max bandwidth for individual curl requests.

The second approach is allowing the user to define a relative amount of bandwidth to use. The user would choose a value between 0.0 and 1.0, and ISR would use that fraction of the available bandwidth at any time for background uploading. This is trickier to implement, but can ensure that the VM always has some available bandwidth and is insensitive to changes in available bandwidth while the VM is running. This option is preferable to the former because of its robustness to varying network conditions, and was the one implemented.

### 3.8.2 Throttling via curl: an anecdote

Through the course of determining the best way of implementing throttling, a bug in the curl codebase was discovered. The bug was stumbled upon while trying to determine how curl handled the rate limiting option. Because the absolute bandwidth rate was passed a value in bytes per second, it was unclear whether this was an average value over a second or a larger interval or if curl was actually throttling individual packets. The distinction is illustrated below:

In the example on the left, three packets are sent at the start of every second, resulting in bursty traffic, but still averaging out to 3 packets per second. In the example on the right, one packet is sent every third of a second, and the average is still 3 packets per second. Ideally, curl would demonstrate the behavior on the right, because the smoother use of bandwidth would be less likely to interrupt any guest use of the bandwidth.

To figure this out, WireShark was used to intercept packets being sent via curl to see the timing between the packets. When this was done, it was clear that the packets were actually following the behavior on the left, being sent in quick succession at the top of every second.

Upon close examination of the source code, it was determined that this was actually a bug and not the intended behavior. In the code which calculates how long to sleep before the next packet is sent actually had an additional factor of 8 multiplied into it, apparently a bit to byte conversion which should not be there since all of the values were in bytes. Upon its removal, the packets were sent in an evenly distributed fashion.

Discovering and fixing this bug was an unexpected hurdle, but a valuable lesson. It shows that no codebases are exempt from bugs, even those that are well-established and widely-used. A patch was submitted and accepted very promptly [3].

### 3.8.3 Dynamic Throttle Control

Users are given an additional level of control over throttling while the VM is resumed. The idea is that users should be able to take advantage of their knowledge of how much bandwidth available at any time. With control over the throttle rate not just at resume time, they can temporarily disable trickle back if they know their VM needs more bandwidth, or allow it to use all of the available bandwidth if the opposite is true. This control is manifested as a simple slider between 0.0 and 1.0 on the VM GUI window.

# 4 Evaluation

In the evaluation section, the various methods of validation mentioned in the introduction are quantified. When building a new system as opposed to adding a feature or an optimization, it is more difficult to accurately compare a single aspect of the new system versus the old because there are many components which may or may not have an analogous part.

## 4.1 Feature Set

As mentioned in the design section, because this iteration of ISR was to be built from VMNetX, which had almost none of the core components of ISR other than the ability to fetch and execute VMs on demand from a server, the first step was to decide which of the features of ISR would be prioritized. This was particularly important given the limited time that was available to essentially rebuild a system that was developed during a period of over a decade. The following table attempts to make a side by side comparison of some of the features in OpenISR and OpenISR 2.0. The table does not claim to be complete, but does visualize some of the design decisions made this time around. The features are ordered in decreasing priority for OpenISR 2.0.

| Feature | OpenISR | OpenISR 2.0 |
|---|---|---|
| Dirty state trickle back | no | yes |
| GUI | no | yes |
| Content Addressable Storage | no | no |
| Versioning | yes | yes |
| Multiple VMM support | yes | no |
| Disconnected operation | yes | no |
| Thin client | no | no |

Table 1: Comparison of features in OpenISR and OpenISR 2.0

It is worth noting that content addressable storage [19] and a thin-client mode [21] were both studied extensively with respect to performance in the original ISR. The former work concluded that CAS could provide storage savings on the server upwards of 60 percent if the chunk size were reduced to 4KB. However, CAS was never actually implemented in ISR. A thin-client mode was actually introduced in VMNetX for the Olive Archive, but never for ISR. The work to quantify the user experience for thin clients concluded that “stateless thick clients”, like VMNetX, are better for preserving responsiveness. However, thin clients also open up the possibility of using ISR with handheld devices, which is an advantage that was no previously considered given their relative

lack of popularity at the time.

Because all iterations of ISR support the basic resume/suspend/checkin commands, OpenISR 2.0 supports the minimal operations that make ISR what it is, as well as a few new features. There is unfortunately a long list of new features that were not able to be implemented, and they are discussed in a future chapter.

## 4.2 Simplicity

One straightforward way to quantify how much simpler OpenISR 2.0 is than its predecessor is to make a direct comparison between the number of lines of code it takes to implement the common core features.

| Component | Name | OpenISR | Name | OpenISR 2.0 |
|---|---|---|---|---|
| Client | parcelkeeper | 5605 (C) | VMNetX | 4241 (C) |
| UI | commandline client | 4206 (Perl) | PyGTK based GUI | 1872 (Python) |
| Server | lockserv | 1351 (Perl) | Django server | 863 (Python) |

Table 2: Lines of code comparison between OpenISR and OpenISR 2.0

The measurements in Table 2 were made using the tool CLOC (count lines of code) [17]. Comments were excluded from the measurements. Some of the helper scripts were not included in the counts for OpenISR components.

For the three core components of ISR, there is a 24 percent decrease in code for the client, 55 percent decrease in code for the UI, and a 36 percent decrease in code for the server. Of course, these numbers are not a precise measurement of the quality of the code, but there has a been a noticeable decrease in the amount of code written to implement the same core features.

## 4.3 Performance

The performance of ISR can be measured in two ways: the user-visible performance and the effectiveness of its features for the system. The former refers to how the system appears to perform to the user. The latter is about how the optimizations that were implemented in this version of ISR improve aspects of the system such as bandwidth usage or time of operations.

### 4.3.1 Trickle Back of Dirty State

The main objective of implementing background upload was to reduce the amount of time that the user must spend waiting for checkin operation to complete. The amount of dirty state created during VM execution can vary drastically depending on the workload. In the experiments below, the goal was to see if the time savings from background upload could be noticed for a couple simple trials.

For each of these trials, a vanilla Ubuntu 12.04 Server VM was used. The VM has a 1494 MB disk image, and a 1040 MB memory image. Both the ISR client and the server were executed on the same machine, i.e., the server was located at localhost:8000. The machine running the experiments had 30 Mbps download bandwidth, which was used by the guest VM in trial 2 to download the 200 MB file. A 16 Mbps upload bandwidth between the client and the server was simulated by configuring the curl command which uploads chunks to the server to limit the bandwidth used to 16 Mbps. The user-configurable throttle rate was set to 1.0 in the GUI, meaning all 16 Mbps of the upload bandwidth was utilized at all times for both background uploading and checkin.

Although obviously running the client and server on the same machine is not an accurate representation of a real-life ISR deployment, the goal of this experiment was to determine the amount of state that is dirtied and uploaded for a particular VM, not the performance of the VM itself. This setup also gives more control over the simulated network conditions because factors like other machines on the network or fluctuations in signal strength are not a factor.

In the first trial, the VM is resumed but no other interactions with the VM are made. It is suspended after 400 seconds, an amount of time which gave the client enough time to upload a non-trivial portion of the dirty state. As previously mentioned, the very first iteration of the live migration code in in the modified QEMU writes a complete memory snapshot to the pipe, resulting in nearly 1 GB of modified data to be chunked and uploaded. As a consequence, even making no changes to the VM creates a large amount of dirty state. Again, uploading these changes is not a waste of bandwidth, because many of these chunks are identical to the final versions of the chunks that are uploaded at checkin. Instead, they should be considered a frontloading of the uploading.

In the second trial, the VM is booted, and within seconds, a download of a 200 MB file is started to the current directory. This process is initiated in the following manner:

```
wget http://ipv4.download.thinkbroadband.com/200MB.zip
```

The www.thinkbroadband.com server hosts fixed-sized files for users to test their

download bandwidth. These files are convenient for experiments such as this. The download is allowed to finish, and then the VM is suspended and checked in. The expectation with this trial was that some dirty disk state should appear due to the download, and hopefully get trickled back to the server as it was being downloaded.

Each row in Tables 3 and 4 corresponds to a single run of the trial.

| Trial | Time Resumed (s) | Checkin w/ BU (s) | Checkin w/o BU (s) |
|---|---|---|---|
| Idle VM | 400 | 380 | 521.25 |
| Download 200 MB file | 500 | 390 | 626.5 |

Table 3: Effect of background upload on checkin time

The fourth column labeled “Checkin w/o BU (s)” is an approximation of the checkin time without background upload enabled. This value was calculated simply by dividing the total amount of data that was uploaded during the trial across both background upload and final checkin by the upload bandwidth. This provides a lower bound on the time that the final checkin would have taken, without taking into account any overhead of the checkin process.

| Trial | Uploaded (MB) disk/memory | Checked In (MB) disk/memory | Reuploaded (chunks) | Wasted BW (MB) |
|---|---|---|---|---|
| Idle VM | .625/391.25<br>(5/3125) chunks | 0/653.625<br>(0/5229) chunks | 0/35 | 4.375 |
| Download 200 MB file | 213/389<br>(1703/3113) chunks | 0/655<br>(0/5241) chunks | 38/17 | 6.875 |

Table 4: Bandwidth use of background upload

Table 4 breaks down the bandwidth usage during the experiments. As expected, in the first trial, very few disk chunks are modified. The small amount of modifications can be attributed to some of the background processes of the operating system that run when the VM is started up. ISR itself does not provide any additional insight into modified chunks, as they are treated as raw pieces of data. In the second trial, about 200 MB of disk chunks are modified and written to the server as expected. Also, note that all of the disk modifications are uploaded in the background by the time checkin is called. This is because the disk and memory chunks are uploaded in parallel, so they should share an equal amount of bandwidth when dirty chunks are available for both. More memory chunks were uploaded in the background than disk chunks because dirty memory state

was still being uploaded before and after the file download.

In terms of wasted bandwidth for these trials, the amount is less than one percent of the total data that was uploaded. There are two important notes about this value. First, with background upload disabled, this value is always zero, since only the final version of every chunk is uploaded at checkin time. Second, this value will vary drastically depending on the workload. As an example of a worst-case workload that would maximize the amount of redundant data uploaded, consider one where a large file on disk is repeatedly rewritten with new data. If the background upload code is not aware of the chunks it is uploading, it could potentially upload every one of the rewrites to the server, instead of the final version of the file. Making the background upload code aware of which chunks are commonly rewritten is a possible future optimization.

## 4.4 Evaluation Summary

This chapter set out to determine how well OpenISR 2.0 expanded upon the core functionality of ISR while decreasing the complexity of the code. The addition of two completely new features is a clear sign of progression from the previous iteration, and the implementation of the same core features in fewer lines of code shows that there were quantifiable simplifications to the structure of the system.

In the experiments to measure the savings provided by background upload, it was made clear that trickling dirty state back to the server while resumed can provide tangible reductions to checkin time at minor wasted bandwidth overhead. More complex workloads would be helpful in determining what savings can be attained in practice.

# 5 Further Feature Discussion

There are a number of features that were forgone due to time constraints. However, a lot of time was spent reasoning about these features with respect to both the users perspective and the server administrators perspective. The following sections provide a discussion, rather than necessarily a decision, with regards to a number of improvements to OpenISR 2.0 that were at one point seriously considered and could be revisited in the future of the project. Possible approaches to implementing the features are also considered.

## 5.1 Thin-Client mode

In the original ISR and in OpenISR 2.0, although the VMs are stored on a server, they must be executed on the client machine. This adds limitations to the system because it requires that the client machine be able to run QEMU and VNC or SPICE, as well as have sufficient processing power to execute a VM. This more or less limits the pool of client devices to laptops and PCs.

A solution to being bound to particular hardware is to offload both the storage and execution of VMs and allow the client to optionally connect to the VM remotely using a VNC or SPICE viewer. This means that users would only need a device with the appropriate viewer software installed to fully interact with their VMs. Not only does this reduce most of the burden of execution on laptops and PCs, but it opens up the possibility of using ISR with mobile devices like tablet PCs.

Tablets are growing increasingly popular, but today, they can't yet replicate the full application suite of a full-size PC. With thin-client support built into ISR, the system will allow users to combine the convenient form factor of their mobile devices with the functionality and utility provided by VMs in ISR. The Olive Archive recently implemented this feature for its VMs, and it may be possible to integrate their client with a custom ISR GUI for thin clients.

## 5.2 Disconnected Operation

Hoarding is a term used in the original ISR to refer to downloading the entire memory and disk image to the client to allow the VM to be run without being connected to the server. This functionality is useful when the user expects to be without a fast Internet connection for a while, for example during a flight.

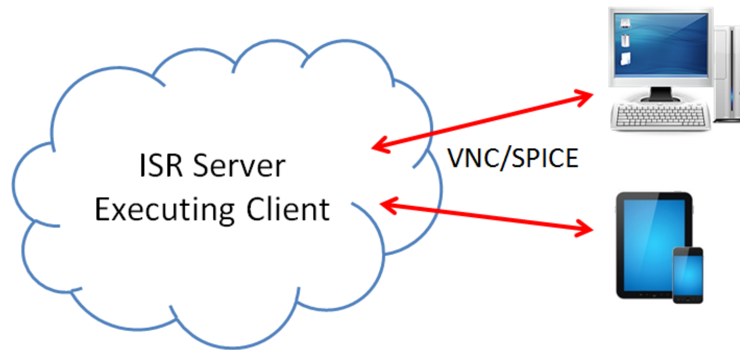



Figure 10: Thin Client Mode

OpenISR 2.0 must handle all of the problems that its predecessors had to handle in order to implement disconnected operation. Currently, all of the information about a user's VMs is being fetched on demand from the server. The metadata for all of the VMs must be cached on the client in order to be made available even without a connection, which is not hard to accomplish by simply writing it to a file in JSON format. Assuming a VM is fully hoarded or pre-fetched onto the client, the user can resume and modify the VM. Obviously, background uploading will be disabled for this duration. Finally, once the user becomes connected once again, the system must be able to synchronize the modified state with the server and other machines that were accessing the VM. For example, ISR must handle the case where two machines make modifications to the same VM for some user. Only one of the modifications can become the new version at checkin, but perhaps the user wants to keep their changes on the other machine and check them in as an entirely new VM. The extent to which these scenarios should be accommodated is not yet clear.

## 5.3 Micro-Checkpoints

One of the primary advantages of OpenISR over a traditional machine is the ability to resume a VM from any checkpoint created without losing the current state. The process of creating a system restore point on a physical device in and of itself is time and resource consuming, and can easily require hundreds of gigabytes of storage space to accommodate a single restore point. From personal experience, this results in the average users relying on one or two backups of their machines at arbitrary, unreliable points in time.

With the current implementation of OpenISR 2.0, users have fully functional checkpoints with each checkin they perform. However, the period of time between two sequential checkins can often times be of too coarse a granularity to undo the mistakes that a user might make while resumed. For example, consider the scenario where a user has been working on a document for 3 hours, but has not checked in their VM for a day. If the user accidentally corrupts or deletes their file, they won't be able to recover it because it was never checked in to the server. To mitigate the damage done in such a scenario, micro-checkpoints are introduced.

Micro-checkpoints are full checkin points created intermittently in the background while the user is using a resumed VM, similar to auto-save features in many applications. This is nontrivial because a check-in requires a synchronized snapshot of both the memory and disk images, now with the added requirement that creating it does not interrupt the user's experience.

From the code which parses the output of the live migration in the background, it should be possible to create consistent snapshots of the memory image each time the live migration is iterated. In order to have a snapshot of the disk at those same points in time, ISR requires a copy of each modified disk chunk at those points. One naive solution would be copy the entire modified disk directory to another location every time live migration is iterated. Although this would increasingly take up large amounts of disk space on the client, each copy of the modified disk chunks should allow a consistent disk image to be created by combining the pristine cache and the modified chunks.

If micro-checkpoints can be implemented, they would reduce the user's dependence on his or her own diligent checking in, and improve the robustness of ISR.

## 5.4 Encryption

Encryption, more specifically encryption of the image chunks, must eventually be added to the implementation of OpenISR 2.0, especially if it is to become available as a service to users who do not also operate their own servers.

**How it affects server administrators:** Server administrators would be trusted with less of their user's data. It should not really affect how they use the system however. Server administrators will store the key that the user uses to decrypt its data from storage, encrypted with a key that only the user knows.

**How it affects users:** The system should have security in place to be taken seriously

by users. More users would be likely to use the system, especially if there are trusted server providers available.

However, now users will have to keep track of 3 pieces of information:

1. Username for the ISR server
2. Password for the ISR server
3. Data password (used to decrypt chunks from the server)

The first two can be treated by the ISR server as they would be by any other web service. For example, a user who forgets his or her password can ask the server to reset it by sending a link to their personal email. However, the data password is the responsibility of solely the user, because allowing the server administrator to read it would defeat its entire purpose. If it is lost, any of the chunks encrypted using this password will be inaccessible. This is a very big burden on the user. There are of course strategies that can users recover their data password such as user-specific hints on the server, or reconstructing the password from pieces, but these do not guarantee it will be recoverable.

**How it affects the implementation:** The biggest modification to the existing system will be the actual encryption and decryption of chunks by the client before reading them into disk and memory images, since user account management is already being done using Django's user model. Again, the encrypted data password for each user must be stored on the server, but this is not difficult to do by adding an extra field to the user model. There will be a performance overhead from the encryption and decryption operations, but it is difficult to quantify it at this point in time.

## 5.5 Cloud Storage

The current ISR system relies on local storage on the server machine. Django writes and reads chunks from an administrator-specified directory. This approach is easy to implement and easy to test just by looking at the local directories and making sure files exist. However, this design does not scale and does not support a service that serious server administrators would desire in a system which can require a lot of storage as it grows. Also, the storage and the server are both on the same machine, creating a single point of compromise. For these reasons, OpenISR 2.0 should support cloud storage.

The system should still support local storage however. This will add to the implementation complexity of the system but implementing complex features like CAS first and testing them locally is a logical first step to developing the system, and this step can just be preserved as an option instead of being discarded once cloud storage works. Also, this reduces the barrier of entry to the system by allowing users to try out the entire system on a single machine before committing to a full deployment, especially if they do not have a cloud storage account.

**How it affects server administrators.** When a server administrator installs ISR, he or she is given the option of storing the VMs locally on the server machine or providing credentials for a cloud storage service. The system should then store the VMs at the set location without any more configuration. When the server is launched, it will parse the settings file and determine where to look for VMs and store them.

**How it affects users.** Unless the user is personally opposed to having their data hosted on a cloud, it should make no difference to the user. If implemented correctly, there should not be a significant difference in performance other than a slight increase in latency as chunks make a few extra jumps between the server and the cloud before reaching the client. If the user trusts the server administrator fully, he or she should have no problem with local storage. If the user does not trust the server administrator with his or her data, cloud storage may be a compromise: the server would handle user information and encrypted user data keys, while encrypted data would be stored on the cloud.

**How it affects the implementation.** Each cloud storage provider may have slightly varying APIs. Depending on the complexity of the calls that have to be made to support cloud storage, supporting multiple different cloud providers may become a "simple matter of programming" rather than a worthwhile use of time.

The following operations will have to be reimplemented in order support cloud storage.

**Uploading a chunk**

Currently, uploading a chunk for during checkin or background upload consists of a HTTP PUT request from VMNetFS to a chunk-specific server URL. With cloud storage, chunks must still go through Django. Django can allocate a directory as a staging area for the chunks before a commit is made to push the chunks to storage, reducing the number of queries made to the cloud storage service. Also, metadata can be tracked without having to intermittently scan the cloud storage for chunks that were uploaded directly through the cloud.

**Downloading a chunk**

Downloading a chunk involves HTTP GET request to a chunk-specific URL on the server. The server can still handle chunk requests as opposed to clients directly communicating with the cloud. The alternative is to generate a temporary url to chunks directly in cloud storage which the client can use while resumed. There are additional complications in this approach, such as handling temporary access expiration.

**Locating a chunk**

Extra metadata must be kept on the server to track which files are on the cloud versus local storage. For each version of each VM, an array of tags must be stored to track each chunk number corresponding to an actual chunk file in storage somewhere. These can be stored in a plaintext file on on the server. In order to efficiently look up these relationships, this file can be read into memory instead of being constantly read and rewritten.

Cloud storage should be supported by ISR to some extent in the future. Supporting only a single cloud provider's API, namely Amazon S3 [1], is reasonable. That would cover one of the more popular commercial options, as well as be compatible with Ceph's object store [2], a popular open source alternative. Implementation should begin with setting up cloud storage on a local Ceph instance using the S3 API if possible, with the expectation that the same commands should work with Amazon directly. This setup is likely to be easier to debug and will not cost any money to run and test if minimal required hardware is available.

## 5.6 Multiple Servers for a Single ISR Deployment

This feature refers to the ability for a server administrator to store VMs across physically distinct servers, whereas the current implementation only allows the machine which is running the Django server code to act as the storage server as well as the HTTP server.

**How it affects server administrators:** Server administrators now have extra freedom to put different VMs on different servers. Why would they want to ever do this? It may not be practical for a real deployment of ISR, but a function to support migration might be useful if an administrator wants to move all of their chunks from one storage server to another, as opposed to having some of their chunks in one place and some in another.

The server can be definitely be used with a distributed file system to avoid these migrations for the sake of increasing scalability. However, the migration feature could still prove useful, because it would allow server administrators to offer users the option to integrate their personal storage with the ISR server.

**How it affects users:** If information about where their VMs are being stored is never exposed to users, it should not matter in most cases. Consider a scenario where a user owns his or her own Ceph storage server, but does not want to have a personally maintained installation of ISR. Should the user be able to tell a server admin to store his or her VMs only on his or her personal cloud store, instead of the server admin's server? The only reason a user might do this is because he or she do not trust the administrator, since using personal storage costs extra money for the user. The fact that the server administrator can not decrypt the data in storage without the user's data key should be enough to satisfy security concerns. If users are really serious about data, they would probably just create their own ISR server.

**How it affects the implementation:** This can be implemented minimally by adding a field to the VM model in Django to track which cloud storage instance the VM is stored. Given an abstracted set of functions to talk to any of the supported cloud storage services, the server should be able to talk to any storage server using the same commands. The implementation is complicated by functionality like migrating VMs between two different clouds. If CAS is implemented on the server, the reference counts must be updated in the source and destination clouds to make sure that dead chunks are removed correctly. If an intra-parcel policy is implemented, migration is simple: move the entire pool of chunks from one cloud to another with the same reference counts and metadata. With the ALL policy, where all chunks for all VMs is in a single pool [19], this is much trickier because chunks that are still valid for other parcels on the cloud must be preserved, and redundant chunks should not be copied over.

## 5.7 Content Addressable Storage (CAS) on the server

Given the decision to encrypt all of the data stored in the cloud, adding CAS to the server does not require any CAS-specific restructuring of the system.

The big design decision that must be made is determining which storage policy to use. In an intra-VM policy, "each parcel is represented by a separate pool of unique chunks shared by all versions, $v_1, ..., v_n$, of that parcel". In the ALL policy, "all parcels for all users are represented by a single pool of unique chunks" [19]. An inter-parcel

policy, one that puts all of a single user's VMs into a single pool, is not considered separately because it is not expected to yield noteworthy savings over the intra-VM policy, since a user's VMs are likely to be different in content, e.g., a windows VM and an ubuntu VM will not share any system files, packages, system updates, etc.

The ALL policy yields greater savings in storage on the cloud because each chunk needs only be stored once across all users and all VMs. Note that this results in network savings for users as well, without having CAS on the client. For example, if an update for Windows is released, it only has to be uploaded by one user instead of all users who have a VM based on the Windows base VM.

These benefits from the ALL policy come at the sacrifice of privacy. Because all of the chunks require a consistent encryption scheme, someone could perform dictionary-based traffic analysis of requests [19]. This could be used to discover copyrighted material in a user's VM by someone who has access to the chunk pool. However, this potential threat is not part of our threat model.

Instead of encrypting each chunk with the user's data key and a chunk index, ISR can use convergent encryption on each of the chunks of a user's VMs. This means a keyring will have to be maintained for each VM. Instead of encrypting each of a user's chunks with a key that only the user has, encrypt the chunks using the hash of the content of the chunk and share the chunks in a pool of some policy (intra-parcel, inter-parcel, ALL). The keyring for a VM is encrypted with the user's key, so a certain level of privacy about which chunks comprise the VM is maintained.

**How it affects server administrators:** Some storage savings on the server. Reduced requests to the cloud storage service. Note that currently with Amazon S3, the pricing for a PUT request (upload chunk) is \$0.005/1000 requests and a GET request (download chunk) is \$0.004/1000 requests, 12.5 times cheaper. Therefore, reducing the number of PUT requests by using the ALL policy is likely to add up to savings over time.

**How it affects users:** Upload bandwidth is saved: at check in time, the client can hash each of the chunks in its modified cache and send the list to the server. The server can then check which ones do not exist in storage, and return a list of only new chunks to the client. The client can then only upload only the missing chunks, and also the updated keyring encrypted with the data key. This should reduce the cost of checkin operations in terms of time and bandwidth.

**How it affects the implementation:** First, a keyring must be created and maintained for each VM in storage. Each VM has an array of (tag, key) pairs where key is

the encryption key for the chunk (SHA-1 hash of plaintext) and tag is the SHA-1 hash of the encrypted chunk. Each of these keyrings is encrypted with the user's data key and stored on the server, rather than the cloud. This allows the cloud to store purely chunks, and if the cloud stored the keyrings, they would have to be downloaded through a server request anyway. Keeping the encrypted keyrings on the server is not a security threat because the key is only accessible to the user.

Fetching a chunk would be as follows:

1. VMNetX has a pristine cache miss
2. Client checks the decrypted keyring and finds the hash of the encrypted chunk
3. Client requests server for the chunk corresponding to the hash
4. Server gets the encrypted chunk from storage
5. Server returns the encrypted chunk to the client
6. Client decrypts chunk using the key from the keyring and puts it in the pristine cache

Checkin would be as follows:

1. Client takes hashes of the encrypted modified chunks
2. Client sends array of hashes to server
3. Server returns list of modified chunks not in the cloud storage
4. Client uploads the hash and the encrypted chunk pairs to the server
5. Client uploads new keyring for updated version
6. Commit: Server uploads chunks to cloud and stores keyring with new version

### 5.7.1 Interaction with background upload

Background upload requires that chunks be put into a staging area before being committed into a version. Because the process can result in chunks being overwritten multiple times, they are staged before an explicit checkin is done. The staging area should be on the server rather than cloud so that every chunk in the cloud belongs to some version of a VM and also to reduce the number of jumps that each upload causes.

When the user does a checkin with CAS enabled, the client sends an array of modified chunks and their hashes. The server compares this list to the (chunk number, hash) pairs in the staging area and returns the list of chunks that are not in the staging area. The client then uploads all of these missing chunks to the staging area. Once the server confirms all the new chunks comprising the new version are present, the client is notified, and the chunks are uploaded to the cloud.

### 5.7.2 Hash Collisions

If by some chance two distinct chunks have the same hash, and as a result a chunk in the CAS pool is incorrectly overwritten, any of the versions of the user's VMs which contain the collided chunks become corrupted. There is a very small probability of such an event occurring, but it is non-zero and should be noted.

### 5.7.3 Garbage collection

Garbage collection is important for remove chunks which are not referenced by any versions of any VM in cloud storage. Each chunk stored on the server must be associated with a reference count which tracks the number of VMs (or VM versions if deletion of versions is allowed) that still depend on that chunk. Once this count reaches zero, it is eligible for garbage collection.

Reference counts are stored in table (hash(chunk), count) on the server where the count is the number of unique VMs that own that chunk. This reference count array must be updated every time a new VM version is created (checkin and VM creation time) This array is stored on the server similar to the keyrings, in a Django model which rows consisting of (hash, vm_id). To get the actual reference count of a chunk with hash, all the rows with the corresponding hash must be selected. All modifications to this table must be atomic, and a checkin of a VM must own the lock to this table so that chunks are not garbage collected before the system confirms that they are present and therefore do not need to be uploaded.

When a chunk is staged, its reference count is not modified. It is possible that a chunk which already exists on the cloud is staged at the point of checkin. If this happens, the chunk is simply not uploaded to cloud storage. Otherwise, if the chunk does not exist at the point of checkin, it is uploaded and the count incremented by adding another row to the reference count table.

This routine could be scheduled to run at a administrator-specific time to minimize the number of active users while it runs, or manually every once in a while.

## 5.8 CAS on the Client

This section also assumes encryption of chunks is being done. Without CAS on the client, the keyring downloaded from the server is used to identify and download each of the required chunks, which are then decrypted and cached on the client. Multiple copies of the same chunk can be downloaded if it appears in multiple indices of the keyring, because the client does not check for duplicates.

Implementing CAS on the client means that each user has a local pool of content addressable chunks to prevent downloading the same chunk multiple times. This saves download bandwidth at the expense of implementation complexity and extra metadata on the client.

CAS on the client was not a priority because the savings in download bandwidth do not justify the additional complexity. This feature would reduce resume times and the number of cache misses leading to chunk requests from the server, but resume times are only prohibitive with a cold cache resume, which does not occur as often as a warm cache resume (only at first checkout or after a clean).

CAS on the client can be added to the system at a later point without affecting the rest of the system, only client side storage. Users would not have to make serious modifications to their system, only clear their current caches and repopulate them.

It may be worthwhile to create a special case for the zero chunk however, due to its frequency and ease of identification. This could be done as follows: on the client, receive and store the hash (tag) of the zero chunk from the server at resume time. This can be fetched in advance, and cached, just one time. A zero chunk is constructed and cached on the client at resume time. This is only 128 KB and is safe to delete for obvious reasons. When a cache miss occurs, the client goes to the keyring to find the tag of the chunk. If the tag is the same as that of the zero chunk, it caches the zero chunk by copying the file (not linking it, due to maximum reference count limits). Otherwise, it requests the chunk from the server as with any other chunk. This operation costs one hash comparison and potentially one file copy every time a miss occurs, but should not create prohibitive overhead.

# 6 Related Work

There have not been any efforts to directly rebuild ISR as this work attempts. However, there are many options for mobile computing which may be considered in place of OpenISR 2.0.

## 6.1 OpenISR

OpenISR, while not currently being developed, remains a completely usable system. The code is available as open source, and there are detailed instructions on setting up both the client and the server. Some details about how OpenISR and OpenISR 2.0 are discussed in other parts of this document. Despite its slightly different feature set, OpenISR at its core still allows users to make their VMs portable, and is a more established, tested alternative to OpenISR 2.0.

## 6.2 The Olive Archive

The Olive Archive, or more specifically VMNetX, is in many ways the predecessor to OpenISR 2.0. Both pieces of software provide on-demand execution of VMs from a HTTP server. Both also support only QEMU/KVM as the hypervisor.

The Olive Archive already has a functional thin-client mode, where the VM is executed on the server or a user-designated cloud, and can be accessed remotely through a VNC or a SPICE client. In addition to providing the benefits of offloading computation, this allows Microsoft Windows clients to use VMs by installing a custom VNC client instead of installing VMNetX, which currently only supports Linux hosts. The VMs can also be accessed through an Android client, which is essentially a VNC or SPICE viewer with a custom control interface so that the keyboard is easily accessible.

The Olive Archive also has only a web UI for seeing the available VMs, as opposed to a local one like OpenISR 2.0. This distinction is reasonable as Olive does not support disconnected use of its VMs.

## 6.3 Alternatives to ISR

Since the inception of ISR, there have been many developments in software and web applications which cover a subset of the possible uses of ISR.

### 6.3.1 Simple document editing

Apps such as Google Docs [7] are commonly used for the purposes of editing and collaborating on simple word documents or spreadsheets. While the main draw of these apps is the ability to collaborate, if the only objective were to portably edit a text document, one of these alternatives is probably a better option than ISR because they are more lightweight and equally accessible across multiple platforms.

### 6.3.2 Multiple OSs on a single machine

Applications such as VirtualBox [13] or QEMU can be used to locally launch virtual machines of any OS on supported host machines. If a user works strictly on a single machine and is diligent about creating backups, then he or she might find that either of these alternatives is sufficient for their purposes. ISR itself builds upon QEMU to make VMs conveniently accessible on multiple machines in addition to tracking changes made to the VM over time.

### 6.3.3 Running applications not compatible with host OS

If the user's objective is more specifically to simply run an unsupported application on their machine, even more alternatives exist. For example, if a user on Ubuntu wanted to run a Windows program, one potential solution would be to try WINE [15], which exists for just that purpose. However, WINE is difficult to configure and often times, applications must be individually configured to get around various issues like mouse capture or graphics incompatibility. VMs are more reliable because they emulate the entire environment, so if an application works for a specific OS, it would always work for a VM of that OS running in ISR.

### 6.3.4 Other Emulation Platforms

Hardware emulation is not an idea unique to ISR or VMs. For example, many emulators exist for various video game consoles, which allow users to play games intended for proprietary hardware on their personal machines.

Simics [11] is an example of a full-system emulator which can be used to simulate various systems such as MIPS and ARM in order to develop software for these embedded systems.

# 7 Conclusion

OpenISR 2.0 is an attempt to reenvision the original ISR concept in a different computing landscape than was originally predicted. Personal devices have gotten smaller while anonymous hardware is not as widespread as anticipated, and the role of portable personalized machine state has been divvied up into various web services like cloud storage and web-based document editors. Despite these developments, ISR remains a viable option for making more specialized applications running on specific operating systems available to users in a portable environment. This document explores many ways that ISR leverages and can continue to leverage modern web technologies, instead of having its uses subsumed by them.

## 7.1 Contributions

The main contribution of this work is the redesign and reimplementation of ISR. Although the code is not as robust or complete as the previous implementation, it shows that the ideas presented in the redesign are more than feasible. Two important features, the GUI and background upload, demonstrate that the user experience can be improved noticeably without negatively impacting the complexity of the code. The new HTTP-based server leverages a modern web application frameworks while making the client-server interactions more generic. These features were prioritized over the ones described in Chapter 5, because of their immediate and obvious effect for users rather than server administrators.

The document also provides a detailed analysis of where to take the system in the future with respect to new features that incorporate recent services like cloud storage. Because many of these planned features drastically affect many different parts of the code, it is important to know in advance which files require modifications, and to write current code in a modular manner with possible improvements in mind.

## 7.2 Future Work

As Chapter 5 and 6 suggest, there are numerous opportunities for improving OpenISR 2.0. Any future work on ISR would most likely determine which of those features are still the most relevant, and then add it to the OpenISR 2.0 implementation.

There are also methods of improving already implemented features, such as reducing the amount of redundant data uploaded to the server during VM execution. This could

be done in a naive manner by tracking the order in which VM chunks are modified, and then uploading them in reverse order i.e. least recently modified chunks first. The assumption behind this heuristic is that the longer a chunk is unmodified, the less likely it is to be modified in the future. Data should be collected to verify this claim, but it is intuitively a reasonable place to start.

## 7.3 Final Thoughts

Internet Suspend/Resume was originally introduced as an approach to improve mobile computing by decoupling machine state from physical hardware. Many of its original use cases have been usurped by a more recent developments in web applications and mobile devices. This work was an attempt to show that ISR can be brought up to speed with current computing trends, and be made even more usable than its previous implementations. Further advances in networking like increased bandwidth will only positively affect ISR by reducing resume and checkin times further. ISR remains a viable option for users who require an entire virtualized computing environment and want to be able to quickly access it from different physical machines.

# References

[1] Amazon S3. http://aws.amazon.com/s3/.

[2] Ceph. http://ceph.com/.

[3] curl. http://curl.haxx.se/.

[4] Django. https://www.djangoproject.com/.

[5] Dropbox. http://dropbox.com.

[6] Gmail. http://gmail.com.

[7] Google docs. https://www.google.com/docs/about/.

[8] Microsoft Windows. https://www.microsoft.com/en-us/windows.

[9] The Olive Archive. https://olivearchive.org/.

[10] OpenISR. https://github.com/cmusatyalab/openisr.

[11] Simics. www.windriver.com/products/simics/.

[12] Ubuntu. www.ubuntu.com/.

[13] Virtualbox. https://www.virtualbox.org/wiki/Downloads.

[14] VMNetX. https://github.com/cmusatyalab/vmnetx/.

[15] Wine. https://www.winehq.org/.

[16] Wireshark. https://www.wireshark.org/.

[17] Al Danial. Count lines of code. https://cloc.sourceforge.net/.

[18] Benjamin Gilbert, Adam Goode, and Mahadev Satyanarayanan. Pocket ISR: Virtual machines anywhere. Technical report, 2010.

[19] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[20] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andre Lagar-Cavilla. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing*, 11(2):16–25, 2007.

[21] Niraj Tolia, David G. Andersen, and M. Satyanarayanan. Quantifying Interactive User Experience on Thin Clients. *Computer*, 39(3):46–52, March 2006.