SBUML: Multiple Snapshots of Linux Runtime State

Richard Potter  Kazuhiko Kato

Taking snapshots of live virtual machines is much easier than with real hardware, because a virtual machine’s devices are simulated in software where it is easier to capture 100% of the necessary state. More so than for real machines, snapshot functionality has become fundamental for virtual machines. It is a necessary feature for administrators who use virtual machines to quickly balance the loads of servers in data centers, and developers who use snapshots to restore test environments to pristine state. This paper describes SBUMIL, a system that extends the User-Mode Linux (UML) virtual machine to provide snapshot functionality. SBUMIL snapshots are more portable than those of other virtual machines, such as Xen, that take on more restrictive host requirements. SBUMIL also integrates several alternative and complementary techniques for optimizing the storage and transfer of multiple, multi-gigabyte virtual machine states. This makes SBUMIL an ideal platform for researching design trade-offs that can make snapshot techniques practical for cutting-edge applications such as sustainable systems, model checking, software demonstrations, and others.

1 Introduction

SBUMIL[1] is a system for saving OS-level Linux virtual machine and dependent state in enough detail so that execution can be continued at later time(s) or different computer environment(s). All dependencies internal to the OS are saved into a snapshot and copied to the target environment exactly (Figure 1). Some state that connects the Linux state to the host target environment is regenerated. State beyond the virtual machine’s simulated physical devices is not copied, so while network connections can stay active, they might depend on state not saved in the snapshot.

Duplicating runtime states of all sizes is common in computer system design, and the main alternative to snapshots is to regenerate the needed state. For example, rather than saving the entire virtual machine state along with RAM, it can make sense to reboot and regenerate a roughly equivalent RAM state. However, regeneration is not an option if critical inputs to rebuild the state are no longer available. Also, regeneration is often slower than snapshots.

Even when snapshots are necessary, some state is usually regenerated at the target. Most solutions that are not OS-level try to minimize the snapshot size by carefully separating out only the state that cannot be regenerated. This model would describe common examples like the saving of word processor documents. These application specific solutions require extra effort from software developers and introduce new risks for bugs. Making sure all dependencies are satisfied can be very complicated.

OS-Level snapshots eliminate this extra effort and risk, because they automatically capture the state of all programs and dependencies inside an OS. The regenerated state at the target environment is minimal, and it is easy to understand the distinction between what is copied and what is not. Therefore, OS-level snapshots can be more reliable
and easier to design into solutions. Since no explicit saving step is required by application logic, OS-level snapshots can be saved at a fine time granularity, even for legacy applications that have no intrinsic snapshot support. The detail captured is also greater than that for application-level snapshot solutions.

The key disadvantage of OS-level snapshots is the size, which might include hundreds of megabytes of memory and thousands of megabytes of disk state. In fact, OS-level snapshots have only recently become a practical option on common PC hardware, thanks to advancements in virtual machine techniques and hardware performance. Virtual machines help by simulating devices in software where it is easier to capture snapshot dependencies. It is possible to regenerate identical software-based devices, even if the actual hardware environments

Fig. 1   SBUMI copies all OS-level state exactly, regenerates minimal state for interfacing with the target host environment, and substitutes the target host’s network environment.
are significantly different. Recent virtual machine techniques have improved runtime efficiency so that software can run inside virtual machines with acceptable performance. Finally, hardware performance and storage capacities can now reasonably copy and save the gigabytes of raw state contained in some OS-level snapshots.

Other OS-level snapshot solutions exist, and people are starting to discover new applications. VMware cites examples of sales engineers who use snapshots of running products to improve sales presentations with dependable live demonstrations \[2\]. Tedious setup of the demonstration, including details such as optimal window placements, can be captured confidently in advance. Polished snapshots can be distributed to other sales staff. Customized snapshots can be tailored towards specific types of customers. For applications such as this, the reliability, level of detail, and overall end-user simplicity of OS-level snapshots more than pay for the extra computational resources required.

Application-specific optimizations are possible that enable new uses for OS-level snapshots. For example, the time required to copy hundreds of megabytes may be too long if it means that a server undergoing migration will be down for tens of seconds. Solutions have been implemented for VMware and Xen that reduce downtime to less than a second, using a trick that first copies a tentative snapshot while the server is still running. Then the server is stopped, but only long enough for the small amount of recently changed snapshot data to be copied. This optimization meets the application-specific goal, even though the overall process actually takes longer.

Counterintuitive optimizations like this show that the large sizes of OS-level snapshots do not necessarily undermine their intrinsic benefits. In the server migration example, regenerating the active server state from event logs would be very difficult, and even if possible would take too long. Therefore, optimization of snapshot based solutions can make new, cutting-edge functionality practical that is not otherwise possible.

The questions that motivate SBUML are (1) what other optimizations are possible and (2) what new applications then become appropriate for OS-level snapshots? SBUML currently focuses on optimizations for storage of snapshots and includes some techniques for optimizing migration. The techniques are integrated so that they can be applied in combination, which makes it possible to explore complex trade-offs that might benefit specific applications by experimenting on actual multi-gigabyte software distributions.

The next section presents a short history of SBUML and its relationship to other OS-level snapshot systems. Section 3 introduces the storage optimizations that are the focus of this paper. Section 4 describes some of the migration optimization features that have also been prototyped in SBUML. Section 5 explains the implementation techniques used to save and restore snapshots of UML virtual machines. Section 6 evaluates the snapshot storage optimizations and measures the effect of SBUML’s code additions on UML’s runtime performance. Section 7 briefly surveys potential applications where the optimizations increase practicality. Section 8 concludes and discusses future work.

2 History and Related Systems

SBUML is an extension of User-Mode Linux (UML) \[3\][12], which is a virtualized Linux that can run on Linux hosts. The original motivation for SBUML was to provide multiple snapshots for research of novel software development tools \[17\]. The key requirements were that snapshots had to be possible at fine time granularity and have enough detail to capture all effects of the program under development.

The alternatives to UML were other OS-level environments, such as VMware\[4\] or Bochs\[5\], or non-OS-level virtual machines, such as Java or Smalltalk. At first Java was chosen because of its popularity and availability of source code. However, its VM was tightly integrated with the host OS, which made taking snapshots more complicated than expected. Also, all the interesting example programs we were anticipating using in the VM involved networking of more than one VM, which meant that the state of the network stacks in the host OS would be an important effect of the example programs.

This made us reconsider OS-level solutions. VMware was attractive because it was fast and already had the ability to take multiple snapshots by using its suspend feature and undocumented tech-
niques. However, source code was not available which made it difficult to customize for research projects. At the other extreme, Bochs was open source, but was very slow because it emulated everything including the microprocessor.

UML fit nicely between these extremes. All code in UML could be executed directly on the host microprocessor, so performance could be much faster than Bochs. And unlike VMware, UML was open source. One factor that was less important was that VMware and Bochs could run Windows and other operating systems in addition to Linux. UML could only run Linux, but given the popularity of Linux, this was acceptable for the research purpose. Osamu Sato began initial development in the summer of 2002 as part of his masters research. Basic snapshot functionality was working by the Spring of 2003. The system was applied to model checking later that year.[18]

Since 2002, other OS-level systems capable of snapshots have appeared. More recent versions of VMware officially support snapshots and even support multiple snapshots. QEMU[6] is an emulation solution that uses faster techniques than Bochs and can produce snapshots portable between x86 and PowerPC architectures[15]. Xen[7][9] can create high performance OS-level virtual machines that are faster than SBUML.

Still, SBUML retains a unique niche among these solutions. Some of these are a direct result of the choice of UML. It is open source, whereas VMware still is not. UML’s direct processor access makes SBUML faster than QEMU. And SBUML snapshots are more portable than Xen snapshots, because Xen (and VMware) require ring 0 privileges on the microprocessor. Because UML only requires a host that provides standard x86 Linux system calls, SBUML snapshots can be restored inside almost any Linux instance, even if that host Linux is running inside of QEMU, Xen or VMware. SBUML snapshots created in 2003 can still be restored on PC and Macintosh x86 hardware today. The system has been used in research projects studying autonomous load balancing in cluster computing [10] and sustainable infrastructure for Web services [8][14][19].

SBUML has other unique characteristics that result from its longtime focus on multiple snapshots, such as the multiple-snapshot storage optimization techniques that are introduced in the next section.

3 Storage Optimization for Snapshots

Storage optimizations techniques work well with OS-level snapshots because they naturally contain a lot of redundant state. However, the large sizes of the snapshots leads to interesting trade-offs. SBUML helps to explore these trade-offs by supporting multiple techniques. Some techniques, like copy-on-write and copy-on-reference, attempt to reduce redundant state by preventing it. Straightforward compression and memory ballooning can be used for removing redundancy within single snapshots. Delta compression is provided to remove redundant state between multiple snapshots. The abbreviations in the following section headers (i.e. cow,cor,dl,b,gz) will be used in the tables later in the paper.

3.1 UML Copy-on-Write (cow)

Copy-on-write (COW) is a natural technique for sharing redundant state. UML’s block device driver has COW feature, which makes the block device on the guest virtual machine appear as two files on the host. One file is called backing file and is read-only. The other file is the COW file and receives all blocks written to the guest device. When the guest reads a block from the block device, it checks first to see if that particular block has been written to the COW file. This is done with a fast bitmap check. If the bit is set, it reads the block from the COW file. If not, it reads the block from the read-only backing file.

UML implements the COW feature using files that are the same length as the backing files, plus a little more for the bitmaps. However, the amount of storage consumed is much less than this length because UML uses Linux sparse files for COW. Only used blocks in the sparse file actually consume disk storage.

Two standard tar files are used to save the snapshots. One tar file stores the four virtual memory files and the status file (described in Section 5.1). The other stores the rest of the virtual machine files, including all COW device files. The two tar files are created with the -S option, which instructs the tar command to efficiently store empty blocks that are filled with zeros. Then when the another
tar command later expands the state, it has enough information to recreate the efficient Linux sparse files. This is critical, because even when a COW file stores little data, it is still the same size as the underlying virtual machine's file system. The rest is filled with zeros. If tar were to write all these zeros out, it would needlessly waste time and space writing many gigabytes of zeros to the host's file system.

One limitation of Linux sparse file support is that there is no way to test for a zeroed block without actually reading in the entire 4k block. Therefore by default, tar will actually read all the zeros in a COW file when storing it, even though they do not actually take up space on the host or the resulting tar file. For a multi-gigabyte file system, this can take several minutes, even for empty COW files. To work around this limitation in the host Linux, SBUMIL includes a version of tar customized especially for SBUMIL that includes logic to test the COW file's internal bitmap to see which blocks are actually used. This reduces the time to save nearly empty COW files to a small fraction of a second.

It is important that it be possible to match snapshots with their backing files and that these files never change. SBUMIL defines a base distribution as a set of backing files that, when paired with empty COW files, contain enough software to boot UML. Snapshots record the name of the base distribution so that the correct backing files can be found automatically when snapshots are restored. (see Section 5.5)

Without using the COW optimization technique, the information in virtual machine's block devices would have to be duplicated for every snapshot. The block devices contain the virtual machine’s file system and OS distribution, so this could amount to many gigabytes of information that would consume excessive storage and slow down the speed of saving every snapshot.

The COW technique is very effective, especially for cases of multiple snapshots where the contents of the block devices is mostly unchanged from the base distribution. When the COW files contain no changes, this optimization reduces the size of the snapshot by the size of the base distribution, and snapshots can then become as small as the virtual machine's memory. Because there is little downside to using COW, SBUMIL uses UML’s COW feature for every snapshot.

### 3.2 External Copy-on-Reference (cor)

Redundant state can also appear in the COW disks, because state that is stored by the virtual machine can appear in multiple snapshots.

To reduce this redundancy, virtual machines can be “cloned” from a snapshot in a way that uses empty COW files instead of making a copy of the COW files from the snapshot. This establishes a parent/child relationship between the virtual machine and the snapshot’s COW disks. If SBUMIL tries to read a block from the COW file and finds that it is empty, the block is copied from the parent snapshot to the child, and the read is allowed to proceed. In other words, only COW blocks that are actually used are copied to the virtual machine. Any snapshot saved from a cloned virtual machine can also produce cloned virtual machines, so a general parent/child tree of COW files is possible. A base distribution is at the root of such a tree.

Notice that the copy-on-write base distribution behavior is somewhat different from the copy-on-reference (COR) behavior of cloned virtual machines. Copy-on-write is more aggressive at reducing redundant state, because it does not copy data that is only being read. On the other hand, copy-on-reference has the advantage that frequently used data is copied down closer in the parent/child tree to the virtual machine where it can be accessed more quickly.

One downside to cloning and copy-on-reference is that all parent snapshots must be accessible in an uncompressed state so that individual device blocks can be efficiently fetched. Therefore, even though snapshots may be stored more quickly and in less space, while the virtual machine is running more space may be required.

### 3.3 Binary Delta Compression (dl)

There are limits to the effectiveness of the copy-on-write and copy-on-reference techniques. They work on file-system blocks, which only allow data to be shared at a typically 4096 byte granularity. Also, SBUMIL does not yet apply these techniques to capture redundant state in the RAM files.

Binary delta compression mostly gets around these limitations. SBUMIL can apply the standard xdelta utility between arbitrary snapshots, so that
one of the snapshots can be deleted and be replaced with only the fine-grained differences between the two. Almost all redundant state in both COW files and RAM files can be removed. SBUML keeps track of which snapshots were involved so that de-compression can be done automatically even if the snapshots are moved (see Section 5.5). Use of delta compression on entire snapshots is still a feature unique to SBUML.

3.4 Memory Ballooning (b)
Another way to remove information redundantly stored in a snapshot is memory ballooning. Like a physical machine, the virtual machine caches duplicates of disk blocks in its memory for faster access, and both copies will be saved to a snapshot. Memory ballooning forces the guest operating system to free the duplicate copies. We currently use a simple technique of creating a large sparse file, piping the file multiple times in /dev/null, and then deleting the file. Repeating this inside the guest until free memory stops increasing has proven effective and is how the results were generated for the tables in Section 6. The process usually takes 15 to 30 seconds. Faster techniques that directly access the kernel data structures are reported on in [13].

3.5 Simple Compression (gz)
Simple compression is the easiest technique to apply. SBUML simply takes the two tar files described in Section 3.1 and compresses them with gzip.

4 Migration Optimization for Snapshots
Large snapshot sizes and limited network performance mean that optimization can make a big difference in the practicality of moving a snapshot from one machine to another for some applications. The storage optimization techniques from the previous section can help, but only if multiple snapshots are transferred. In other words, copy-on-write and copy-on-reference will not speed up migration for a single snapshot because all ancestor snapshots and the base distribution must be migrated too.

4.1 Precopy
Precopy is useful when it is acceptable for the first snapshot to be copied with no optimization. It is not necessary that the first snapshot is ever used directly. Its purpose can be to only share state with later snapshots to optimize migration. The optimized migration used by VMware and Xen for less-than-a-second server migration is an example of precopy. In other words, although these systems do not require precopy before a request is made to transfer a snapshot, the memory image of the virtual machine will be precopied before the timed portion of the migration is performed.

This technique has been used with SBUML for research in sustainable systems [8][14]. Backup servers distributed across wide area networks can stay synchronized with a master server without using too much bandwidth. Once a full copy of the virtual machine is on all backup servers, synchronization can be done by sending the binary delta compressed versions of snapshots produced by SBUML.

4.2 Demand Fetch
All of the storage and migration optimization techniques so far have taken advantage of redundant state between multiple snapshots. Another technique is to take advantage of the fact that not all virtual machine state is used. By fetching only the needed state, demand fetch techniques can optimize migration.

SBUML modifies UML’s block device (UBD) driver with an extensible mechanism for supporting demand fetching. Before UML reads or writes any block device file (either backing file or COW file), it notifies and waits for confirmation from an external process, called UBD-Helper. UBD-Helper can then use whatever techniques are necessary to copy the necessary blocks to the block device file. SBUML currently uses the curl utility program to fetch the blocks with the HTTP protocol, which has proven to be reliable. Since UBD-Helper is an external process running on the host, it is easy to substitute other methods.

SBUML supports demand fetching for all base distributions and snapshot files. To prepare for demand fetching, special copies of the base distributions or snapshots are created on the target environment. These copies differ from the originals in that empty files are substituted for each block
device file. Also, a bitmap is added for each block
device file to record which blocks have already been
downloaded.

For such a simple technique, demand fetching can
be very effective. Because demand fetching waits
until the last possible moment to decide what to
download next, it can minimize the download with
high accuracy.

4.3 Prefetch

Although effective, the downside of demand
fetching is that by waiting so late the virtual ma-
chine is almost guaranteed to stall until the needed
block of information arrives. The amount of stall
can potentially be long, because the file system and
OS often delay the next request until the previous
request has been filled. Delays such as network la-
tency and process scheduling therefore accumulate
for each of the potentially thousands of requests.
For interactive applications, the stalls are likely to
happen at frustrating times, because if the user ini-
tiates an action for the first time, the virtual ma-
chine might start downloading the information nec-
essary to satisfy the action exactly when the user
is expecting an immediate response.

Since the problem with demand fetching is down-
loading too late, the obvious solution is to download
earlier. But this means the decision to download in-
formation must be predicted before the virtual ma-
chine actually requests it.

SBUMIL has a prototype streaming server that
can performs predictions by comparing the virtual
machine’s recent disk activity with that of previ-
ous virtual machines. The previous disk accesses
are grouped into sequences that represent disk ac-
cesses that occur within a short time period. When
making a new prediction, the sequence that con-
tains the highest number of the virtual machine’s
recent disk accesses gets streamed. Since it is pos-
sible that the sequence is streamed before new re-
cent disk activity happens, the server also keeps a
transition graph of how previous virtual machines
transitioned from one sequence’s disk activity to
another. This transition graph is used to choose
follow-on sequences to stream.

5 Implementation

5.1 Virtual Machine State

Figure 1 showed the guest OS view of SBUMIL. Applica-
tions inside the guest OS see what appears
to be a normal Linux OS. For understanding the
issues regarding snapshots, the host OS view of
SBUMIL is more appropriate, which is shown in Fig-
ure 2. Both figures divide the state into copied and
regenerated state. Some aspects of the copied state
 correspond directly between guest and host views.
What looks like physical RAM on the guest appears
as a set of four files on the host. Each block device
on the guest appears as a separate copy-on-write
file on the host along with the corresponding back-
ing file in the base distribution. Almost all of the
guest virtual machine state is in these RAM files
and block device files, including guest components
such as file systems and interprocess communica-
tion objects. The internal guest components are
invisible to the host, because they are simply guest
kernel data structures stored inside the RAM files.
Because this state is in normal host files, it is easy
to copy using one of many equivalent methods (e.g.
 cp, tar, scp, etc.).

The regenerated state consists of processes and
interprocess communication objects on the host, for
which there is no standard way to copy. These ob-
jects are different from the processes and interpro-
cess communication objects inside the guest. They
are the building blocks on the host that tie to-
gether the state of the guest and connect the guest
to the cpu/memory, disk, and network. There are
some correspondences between the host and guest
objects. For example, there exists a shadow host
process for every guest process, and a TUN/TAP
device on the host corresponds to an eth0 device
on the guest. However, for the purpose of creat-
ing snapshots, they can be viewed as being entirely
different.

Fortunately, UML uses these host processes and
interprocess communications objects in a special-
ized way that makes it relatively simple to reliably
keep track of all the information necessary to re-
generate them. This information is stored in the
status file, making it also easy to copy.
5.2 Saving Snapshots

In essence, making a snapshot is as simple as copying the block device files, the RAM files, and the status file, which are all on the host. However, first the guest virtual machine must be frozen so that state does not partially change and become inconsistent during the copy operation. Also, care must be taken to make sure the host files are synchronized with the latest guest virtual machine state.

For the block devices files, all reading and writing is done with the host OS’s system calls, so the host OS takes care of all synchronization. The simulated RAM files are modified using host OS’s mmap, so again the host automatically handles synchronization.

For the status file, most synchronization is ongoing while UML is running. For example, UML was modified so that codes that create or destroy processes on the host update the status file. UML uses host process Local Descriptor Tables (LDT) to save the LDTs for guest processes, so the UML code responsible was modified to log the same information in the status file. Code in UML that opens or closes files, pipes, ttys, sockets, pseudo terminals, or TUN/TAP devices on the host was also modified to log the result of the actions to the status file on the host.

The rest of the status file synchronization is done when the virtual machine is being frozen. Before freezing, most host processes are in host kernel mode waiting on read operations on host pipes or sockets. SBUML first sets a global freeze flag and forces all processes out of kernel mode by putting special data on all the sockets and pipes. After the processes exit the read system call, they test the global freeze flag, which sends each process to a wait loop. Before entering the wait loop, each process writes out its current execution context (obtained with sigsetjmp system call) to the status file. Because there are a few common execution paths that all UML host processes take, only a few checks for the global freeze flag are needed to send every process to the wait loop.

Because of the special way that UML uses processes on the host, the above information is all that needs to be saved. The main use for hosts processes is to set up CPU execution environments for the guest OS. This is tricky to do using only host user-mode constructs. UML must use clever techniques to make user-mode processes inside the guest perform exactly as if running on a normal Linux OS.
Every process inside the guest has a shadow process on the host, whose only purpose is to hold memory mappings from that host process’s address space to the simulated RAM file. When control of the process is passed to guest user-mode code, these mappings behave exactly like mappings to physical RAM. One host process per guest process helps speed up UML, because it can use the host’s process switching mechanism to change guest memory maps.

Since guest user-mode code is running inside a normal process on the host, it runs with no performance penalty, at least until the code makes a system call or causes a segmentation fault. UML handles these events by using the standard Linux “ptrace” mechanism in a creative way on the host. One special host process, called the “tracing thread”, monitors all the host shadow processes. When these events occur, the host Linux OS automatically passes control to the tracing thread, which passes the event to the guest Linux OS kernel, which then processes the event appropriately by using its own kernel data structures and ordinary host system calls.

Therefore, although SBUML has to basically migrate these host processes to the target environment, it is much simpler than general process migration. With general process migration, it would be important to save the contents of open files. Since UML only opens the RAM file and block devices files that are already saved in the snapshot, this is already taken care of. The address space mappings inside the process would normally have to be saved. However, these mappings reflect the guest kernel mappings, which are already saved in the page tables contained in the RAM files. For general process migration, data could possibly be queued in the interprocess communication objects. For communication that connects outside the guest OS, any possible queued data is ignored by design. For other communication between UML host processes, the purpose is to pass control, not data, between host process. Therefore, data is never queued and SBUML does not have to take any special action.

5.3 Restoring Snapshots

When SBUML restores a virtual machine, it first makes a fresh copy of the files that contain the exactly duplicated state. For the regenerated state, it first reopen all the state files, which regenerates the open file descriptors. Then it regenerates all host interprocess communication objects. Most of these do not correspond to anything inside the guest and are only used for passing control between host processes. Exceptions are pseudo terminals and TUN/TAP connections, which connect directly to the console and network devices in the guest. But these are still better thought of as independent regenerated state. A good physical machine analogy would be the plugging in of a stateless dumb terminal and a stateless network cable.

After all files and interprocess communication objects are reopened on the host, host processes are created with a minimal memory mapping that contains the kernel executable and its global variables. Some host processes must share memory mappings exactly like the processes they shadow in the guest, so care is taken to be sure that such processes are created properly using the standard clone system call on the host. The LDT is restored to host processes as necessary. Ptrace relationships are restored between the host shadow processes and the host tracing thread process. Finally, each host process is returned to the wait loop by restoring the saved context with a siglongjmp system call.

When the global freeze flag is unset, all host processes return to their previous state. When the first host segfault for each processes occurs, the guest UML kernel automatically restores the correct memory mappings as needed based on its internal page tables.

5.4 Networking Support

Since all guest networking configuration is saved in its RAM files, things such as IP address, network routing tables, and even open socket connections remain unchanged. For cases where the guest has been migrated to another networking environment, SBUML provides user-level scripts for the guest and host that automatically coordinate to update the guest IP address, routing tables on the host, routing tables in the guest, and guest DNS information to match the new environment and not conflict with other simultaneously running virtual machines.

For virtual machines that connect to the network as clients, these scripts are enough to allow them
to functional normally, much like when a notebook
computer connects to a new network with DHCP.
For virtual machines that are hosting services, we
do have experience with services like sshd and httpd
and know that they keep working when the IP ad-
dress is changed, although existing connections are
cut. The functionality provided by SBUML’s net-
working scripts is limited to changing the guest IP
address and making sure that packets that arrive at
the host directed to that IP address are delivered
to the virtual machine. For some services, more
specialized configuration might be necessary that
is not provided by the scripts. For example, the
scripts do not deal with how other clients are able
to locate the virtual machine at its new IP address.

5.5 Locating Snapshots
When a snapshot is restored, it must have ac-
cess to all its ancestor snapshots and its base dis-
tribution. All base distributions and snapshots are
given unique identifiers, which makes it possible
for them to change locations and still be reliably
found when needed. Like base distributions, snap-
shots are read-only objects, so finding one with the
matching identifier implies that all the state inside
is unchanged and matches as well.

Each identifier is a random string of hex digits.
By tradition, base distribution hex strings are the
md5sum of all of its component files concatenated
together. By tradition, the snapshot hex strings
are the first 16 characters of md5sum of the save
command’s process id and the output of the date
command. But there is really no meaning in the
digits other than that they should be unique for a
particular snapshot or base distribution.

Given a list of search locations, SBUML will au-
tomatically find a snapshot or base matching a
given set of hex digits. Search locations can be ab-
solute or relative paths on the local file system, or
paths on HTTP servers. If a file named “more-
download-paths.txt” exists at any of those loca-
tions, all paths listed inside the file are placed at
the beginning of the list of search locations. In this
way, SBUML will search depth first through the
list. However, searching of remote paths are always
postponed until all local paths have been searched.

This makes distribution of SBUML bases and
snapshots very simple. They can be placed in
any local file path or HTTP server. The “more-
download-paths.txt” files can link dependent snap-
shots and bases together, even if they are hosted
on different servers. Relative paths can appear
in “more-download-paths.txt”, so directory hierar-
chies of snapshots can be moved around or mirrored
with minimal effort.

For delta compression, identification of the tar
file used for the parent object must be very spe-
cific. For this reason, both the snapshot hex string
and the md5sum of the parent tar file are used in
the search. Tar files that make up a snapshot have
the md5sum included as part of the file name to
help confirm the match.

5.6 Evolution of the Prototype
The implementation of SBUML has been a pro-
cess of taking a working system, which in the be-
inning was UML itself, adding to it incrementally,
and making sure it still worked. Even the
snapshot functionality was implemented in stages.
First, code was added to remove one host process
and regenerate it. Then code was added to remove
all processes and regenerated them. After all pro-
cesses could be removed and regenerated, code was
written and tested for each type of file descriptor,
on by one.

Once save and restore were working, scripts were
written to hide the rough interface used to commu-
nicate with the SBUML kernel additions. Develop-
ment of the scripts and development of the kernel
code took on different character because the ker-
nel code, and thus all its bugs, gets saved into a
snapshot. Therefore, changes to the guest kernel
became more incremental with special care not to
break anything. In contrast, the scripts could be
changed more quickly and with less permanent ef-
effects, so when possible, functionality was put in the
high-level scripts. Reliability of the SBUML kernel
has been the highest priority.

An example of this is the demand fetch frame-
work. The code inside the kernel is very simple. It
merely passes control to an external process (writ-
ten in C) that actually performs the initial demand
fetch logic, such as checking the bitmap. This pro-
cess then passes control to BASH scripts that han-
dle all the logic necessary to find the parent snap-
shots and fetch the necessary block. So even though
the logic in the scripts has changed many times as
bugs are fixed and features added, all snapshots
since 2004 can make use of the later functionality. The disadvantage is that performance would be better if some of the demand fetch logic were moved into the guest kernel, which would remove much of the demand fetch overhead described in Section 6.6. Also SBUML's kernel is quite far behind the UML kernel. SBUML is still a 2.4 kernel, while development for UML is in the 2.6 series. But these changes will not be done until it becomes a high enough priority that enough effort can be allocated to ensure the reliability of the updated kernel.

SBUML adds about 7000 source lines of C code to UML. The higher-level interfaces for the user and for debugging are written mostly in BASH, and consist of about 8000 source lines of shell script.

6 Performance Evaluation

To give a sampling of how well the storage optimization techniques perform, this section presents the results of applying the techniques to multiple snapshots from virtual machines used for two different tasks. The effectiveness of storage optimization depends on how and to what extent the tasks generate redundant data. Therefore, a short introduction to the two tasks is necessary to put the results in context. The first task involves little data and is well suited to OS-level snapshots. The second uses and generates data, which makes practical use of OS-level snapshots more challenging. Contrasting between these two tasks will help show how the various compression techniques work.

6.1 Tasks

6.1.1 Software Demonstration Task

The first task is to archive a proof-of-concept research prototype in such a way that it is easy to set up for demonstrations in the future. The specific example benchmarked here is a software prototype reported on in a workshop paper[17]. For this, snapshots at the OS-level are advantageous because they can capture fine details, including GUI details that are tedious to set up yet make it easier to grasp the research issues the demonstration is meant to communicate[16]. The portability of OS-level snapshots is also a benefit because the demonstrations might take place far in the future on computers with different software installations. For example, the software was written in the Lisp extension language of version 20.7.1 of the Emacs text editor. The research project ended before the code was updated to run with more recent versions.

The size of OS-level snapshots could be a practical issue because they are much larger than the obvious alternative of distributing the source code along with instructions about how to regenerate the demonstration state. The code and data for the prototype is about 120KB compressed and instructions for how to set it up would have added little more.

There are ten snapshots, eight for each of the figures in the workshop paper that showed a screenshot of the prototype, plus two extra snapshots that record steps to set up the demonstrations. When the eight demonstration snapshots are hyperlinked to figures in an HTML copy of the workshop paper, it is possible to click on the figures and restore live virtual machines showing the prototype in same states as in the workshop paper figures.

The virtual machine for the demonstrations was booted with 64M from a file system installed with 645M of software from RedHat 7.2. Installing the prototype code and running the demos added little to the virtual machine size, and thus the raw size of each snapshot is about 716M. It would be possible to reduce the size by booting with only the files necessary to run Emacs and X Windows. Still the raw snapshot size would probably be greater than 200M.

Even unoptimized at over 7GB, storing the whole collection of raw snapshots could be considered practical, because it could be done on a small fraction of a 100GB plus USB external hard drive. Setting up the demonstration from snapshots would save the researcher of the need to further polish the prototype or create installation instructions. For the user, it relieves the need to gather supporting software and follow off-the-demo-topic installation instructions. In this way, software demonstrations and OS-level snapshots are a good match.

So for this task, the goal is not to make it practical but to make it practical in more situations. If smaller, the demonstrations could be kept on active disks or on-line in a form that is practical to download. Optimization of time benchmarks would also widen the practicality. For example, the faster it is possible to set up the demonstrations, the more user attention that can be used to understand the
demonstration itself.

6.1.2 Kernel Compile Task

The second task is less naturally suited to OS-level snapshots, but is interesting for comparison purposes. The task is to archive a software build environment for the Linux kernel. In general, software build environments have less to gain from OS-level snapshots, because they generate a lot of temporary state (e.g. *.o files) that is easy (e.g. via make files) to regenerate.

However, there are still potential benefits to using snapshots. Some software projects are sensitive to certain versions of compilers, linkers, and libraries. OS-level snapshots capture this detail in a portable way that makes it possible to use the tools even on systems that have incompatible versions of the build tools. For compilations that involve simple modifications, expanding the temporary state from a snapshot can be faster than regenerating it will build tools. Also for projects, such as SBUML, where a version of the binary may be run long into the future without restarting, being able to recreate the exact same build environment for that binary can help in debugging. For example, paths to source files contained in the binary will be correct when a (remote) debugger is run inside the virtual machine.

The snapshots were built by first booting the same RedHat 7.2 file system as above with 256MB of memory. Snapshots were saved after installing the compiler, expanding the source code for Linux kernel 2.4.24, doing a default configuration, and doing a full compile. As an example of how the snapshot could be later used, one file was edited, a snapshot saved, and then an incremental recompile was done, followed by another snapshot.

6.2 Individual Techniques

Table 1 and Table 2 show the total effectiveness of the five storage optimization techniques applied individually. Because SBUML always uses copy-on-write, measures for the the other four are taken in combination with it. The size of the base distribution is included in the total. The size for raw copy is estimated and included in the tables for baseline comparison.

For both tasks, binary delta compression gives the best results. For the software demonstration task, this is not surprising because all the snapshots are similar. Table 3 shows values for each snapshot individually (not including the base distribution), and it can be seen that delta compression sometimes can reduce a virtual machine with 64M of RAM to less than 200K. For the kernel compile task, more data is generated, which is reflected in the delta compression numbers in Table 4. The snapshot saved after compiling shows a large increase in the delta compressed size. The snapshot size after the simple code modification step is reduced to 2.7MB, which is just 1% of the virtual machine RAM size.

The other optimizations perform similarly between the two tasks. Simple compression performs second best, followed by copy-on-reference, copy-on-write, and then ballooning. The ballooning result is poor because filling the memory with zeros does not save space unless additional compression is applied to the result.

The most interesting contrast between the two tasks is the copy-on-reference result. For the software demonstration task, copy-on-reference performs similarly to copy-on-write, because little data is generated. For the kernel compile task, copy-on-
Table 3  Software Demonstration Task: Individual Storage Techniques (Snapshot sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Boot 64M</th>
<th>Start 1</th>
<th>Demo 2</th>
<th>Demo 3</th>
<th>Demo 4</th>
<th>Demo 5</th>
<th>Demo 6</th>
<th>Demo 7</th>
<th>Demo 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw copy</td>
<td>715</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
</tr>
<tr>
<td>cow</td>
<td>70</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>cow,gz</td>
<td>12</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>0.632</td>
<td>0.304</td>
<td>0.196</td>
<td>0.256</td>
<td>0.220</td>
</tr>
<tr>
<td>cow,dl</td>
<td>12</td>
<td>11</td>
<td>1.8</td>
<td>0.560</td>
<td>0.304</td>
<td>0.196</td>
<td>0.256</td>
<td>0.220</td>
<td>0.356</td>
</tr>
<tr>
<td>cow,b</td>
<td>70</td>
<td>79</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>cor</td>
<td>70</td>
<td>69</td>
<td>70</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 4  Kernel Compile Task: Individual Storage Techniques (Snapshot sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Boot 256M</th>
<th>Import code and Patches</th>
<th>Install GCC</th>
<th>Untar and Patch</th>
<th>Configure and Compile</th>
<th>Simple Code Modification</th>
<th>Compile Again</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw copy</td>
<td>907</td>
<td>907</td>
<td>967</td>
<td>1163</td>
<td>1466</td>
<td>1466</td>
<td>1541</td>
</tr>
<tr>
<td>cow</td>
<td>262</td>
<td>262</td>
<td>322</td>
<td>518</td>
<td>821</td>
<td>821</td>
<td>896</td>
</tr>
<tr>
<td>cow,gz</td>
<td>12</td>
<td>128</td>
<td>172</td>
<td>183</td>
<td>256</td>
<td>257</td>
<td>276</td>
</tr>
<tr>
<td>cow,dl</td>
<td>12</td>
<td>117</td>
<td>69</td>
<td>86</td>
<td>152</td>
<td>2.7</td>
<td>6.5</td>
</tr>
<tr>
<td>cow,b</td>
<td>262</td>
<td>322</td>
<td>338</td>
<td>524</td>
<td>821</td>
<td>822</td>
<td>896</td>
</tr>
<tr>
<td>cor</td>
<td>262</td>
<td>321</td>
<td>278</td>
<td>457</td>
<td>582</td>
<td>261</td>
<td>431</td>
</tr>
</tbody>
</table>

Table 5  Software Demonstration Task: Storage Technique Combinations (Total sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Total w/ 645MB Base</th>
<th>Percent of Raw Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw copy</td>
<td>7158</td>
<td>100%</td>
</tr>
<tr>
<td>cow,dl</td>
<td>672</td>
<td>9.4%</td>
</tr>
<tr>
<td>cor,dl</td>
<td>672</td>
<td>9.4%</td>
</tr>
<tr>
<td>cor,b</td>
<td>1389</td>
<td>19%</td>
</tr>
<tr>
<td>cor,b,dl</td>
<td>678</td>
<td>9.5%</td>
</tr>
</tbody>
</table>

Table 6  Kernel Compile Task: Storage Technique Combinations (Total sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Total w/ 645MB Base</th>
<th>Percent of Raw Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw copy</td>
<td>8417</td>
<td>100%</td>
</tr>
<tr>
<td>cow,dl</td>
<td>1149</td>
<td>14%</td>
</tr>
<tr>
<td>cor,dl</td>
<td>3376</td>
<td>40%</td>
</tr>
<tr>
<td>cor,b,dl</td>
<td>932</td>
<td>11%</td>
</tr>
</tbody>
</table>

reference is able to save space by not copying the compilation data from snapshot to snapshot. In Table 4, this can be seen by noting how the size of the copy-on-write snapshot always increases in size, but the copy-on-reference varies according to the amount of data generated in each step. The lowest value of 261MB is for the snapshot after the “simple code modification” step, which caused little data from the parent snapshot to be fetched. The highest value of 582MB occurred after compiling the kernel, which referenced and copied much of the kernel source from the parent and generated compiler output.

6.3 Techniques in Combination

SBUMIL allows the five storage optimization techniques to be combined, which makes it possible to explore other ways to reduce storage consumption. Table 5 and Table 6 list seven possible combinations. One easy to predict result is that memory ballooning, which did poorly on its own, does much better when simple compression is also ap-
Table 7  Software Demonstration Task: Storage Technique Combinations (Snapshot sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Boot</th>
<th>Start</th>
<th>Demo</th>
<th>Demo</th>
<th>Demo</th>
<th>Demo</th>
<th>Demo</th>
<th>Demo</th>
<th>Demo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64M</td>
<td>Emacs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>(raw copy)</td>
<td>715</td>
<td>715</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
<td>716</td>
</tr>
<tr>
<td>cow,b,gz</td>
<td>4.0</td>
<td>6.9</td>
<td>7.8</td>
<td>8.6</td>
<td>8.4</td>
<td>9.4</td>
<td>8.5</td>
<td>9.0</td>
<td>8.0</td>
</tr>
<tr>
<td>cow,b,dl</td>
<td>4.2</td>
<td>4.6</td>
<td>3.1</td>
<td>3.4</td>
<td>2.9</td>
<td>2.5</td>
<td>1.3</td>
<td>2.1</td>
<td>1.5</td>
</tr>
<tr>
<td>cor,gz</td>
<td>12</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>cor,dl</td>
<td>12</td>
<td>11</td>
<td>1.9</td>
<td>0.644</td>
<td>0.520</td>
<td>0.240</td>
<td>0.256</td>
<td>0.256</td>
<td>0.252</td>
</tr>
<tr>
<td>cor,b</td>
<td>70</td>
<td>77</td>
<td>79</td>
<td>76</td>
<td>73</td>
<td>75</td>
<td>74</td>
<td>73</td>
<td>74</td>
</tr>
<tr>
<td>cor,b,gz</td>
<td>4</td>
<td>8.6</td>
<td>7.0</td>
<td>8.5</td>
<td>6.4</td>
<td>6.2</td>
<td>6.5</td>
<td>7.1</td>
<td>5.8</td>
</tr>
<tr>
<td>cor,b,dl</td>
<td>4.0</td>
<td>6.4</td>
<td>3.3</td>
<td>4.2</td>
<td>2.3</td>
<td>2.8</td>
<td>2.8</td>
<td>2.7</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table 8  Kernel Compile Task: Storage Technique Combinations (Snapshot sizes in MB)

<table>
<thead>
<tr>
<th></th>
<th>Boot</th>
<th>Import code and Patches</th>
<th>Install GCC</th>
<th>Untar and Patch</th>
<th>Configure and Compile</th>
<th>Simple Code Modification</th>
<th>Compile Again</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>256M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(raw copy)</td>
<td>907</td>
<td>907</td>
<td>967</td>
<td>1163</td>
<td>1466</td>
<td>1466</td>
<td>1541</td>
</tr>
<tr>
<td>cow,b,gz</td>
<td>4.4</td>
<td>63</td>
<td>69</td>
<td>108</td>
<td>189</td>
<td>189</td>
<td>209</td>
</tr>
<tr>
<td>cow,b,dl</td>
<td>4.4</td>
<td>60</td>
<td>6.6</td>
<td>42</td>
<td>87</td>
<td>0.94</td>
<td>1.9</td>
</tr>
<tr>
<td>cor,gz</td>
<td>12</td>
<td>187</td>
<td>119</td>
<td>131</td>
<td>159</td>
<td>75</td>
<td>118</td>
</tr>
<tr>
<td>cor,dl</td>
<td>12</td>
<td>176</td>
<td>16</td>
<td>93</td>
<td>152</td>
<td>2.7</td>
<td>52</td>
</tr>
<tr>
<td>cor,b</td>
<td>262</td>
<td>321</td>
<td>278</td>
<td>476</td>
<td>672</td>
<td>261</td>
<td>461</td>
</tr>
<tr>
<td>cor,b,gz</td>
<td>4.4</td>
<td>63</td>
<td>10</td>
<td>72</td>
<td>111</td>
<td>6.8</td>
<td>61</td>
</tr>
<tr>
<td>cor,b,dl</td>
<td>4.4</td>
<td>60</td>
<td>6.6</td>
<td>70</td>
<td>89</td>
<td>0.99</td>
<td>56</td>
</tr>
</tbody>
</table>

plied to strip out the zeros. Less obvious is how well delta compression can perform in combination with ballooning, because the ballooning process might modify the virtual machine images in ways that could confuse the delta compression algorithm. In fact, applying delta compression on top of a set of ballooned virtual machine snapshots produced the best storage optimization for both tasks.

The kernel compile task results suggest that if delta compression is used, it is best to apply it to copy-on-write snapshots. For example, the cow,b,dl snapshot set requires 848MB total vs 932MB total for the cor,b,dl snapshot set. However, when simple compression is used, the copy-on-reference snapshots compress better. For example, the cor,b,gz set requires 973MB, which is better than the 1476MB required by the cow,b,gz set of seven snapshots.

Table 7 and Table 8 show the storage sizes of the individual snapshots using the seven combinations. It can be seen that the addition of ballooning to simple or delta compression often reduces the size of individual snapshots by a factor of two or more. Using ballooning and delta compression, the snapshot size after the simple code modification step is reduced to less than a megabyte.

6.4 Other Possibilities

In the previous tables, an optimization or combination of optimizations was applied consistently to every snapshot in the set. SBUML can apply optimizations in ways that are more flexible than these examples illustrate. For example, it could make sense only use copy-on-reference for only the “simple code modification” step, because that is where it produces the largest benefit. Then copy-on-write could be chosen for the other steps, because it is simpler and introduces fewer dependencies between snapshots.

On one hand, introducing dependencies between snapshots makes good storage optimizations possible. On the other hand, dependencies make deleting snapshots more complicated. For example, if delta compression is applied and the parent snapshot is deleted, the child snapshot becomes impos-
Table 9 Restore Time and Post-restore Run Time

<table>
<thead>
<tr>
<th></th>
<th>Time to Restore Snapshot</th>
<th>Time to Re-compile Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>cow</td>
<td>50.0s</td>
<td>304s</td>
</tr>
<tr>
<td>cor</td>
<td>13.8s</td>
<td>290s</td>
</tr>
<tr>
<td>cow,b</td>
<td>38.3s</td>
<td>100s</td>
</tr>
<tr>
<td>cor,b</td>
<td>2.1s</td>
<td>280s</td>
</tr>
</tbody>
</table>

The downside is that if any of the snapshots are deleted, all later snapshots become impossible to decompress. For cases where this is not the desired trade-off, SBUML allows delta compression to be applied more freely. The depended upon parent snapshot can be chosen to be any other snapshot, even snapshots with no shared history. This flexibility to mix and match techniques opens up many possibilities for satisfying the requirements of a particular task.

6.5 Performance Trade-offs

The paper so far has focused on storage optimization. However, there are many other performance measures for virtual machines. Like real machines, execution and I/O speed are important. With the flexible ways that virtual machines can be saved, restored, and migrated, other performance measures become important. Although not the focus of this paper, these performance measures deserve some discussion, especially in terms of how they are affected by the storage optimization techniques.

In theory, ballooning and copy-on-reference should make restoring faster, because less data has to be copied. However, they should reduce runtime performance because guest memory caches have to be refilled and data must be demand fetched from parent snapshots. Also in theory, simple compression and delta compression should make restoring take longer, but not affect runtime, because their processing finishes before the machine starts running. In practice, the effects on performance are hard to predict, because of interactions with the host’s disk caches and the fact that ballooning clears out kernel structures in the guest.

As an example, Table 9 shows the time necessary to restore the sixth snapshot from the kernel compile task and the time required to repeat the “compile again” step. These are shown for all the storage combinations that do not use compression. (The copy-on-reference cases assume that the parent snapshots are already in expanded form so that they are ready to be sources of demand fetched data. Otherwise, the expanded forms of the five parent snapshots would add about 48 seconds to produce the 724MB of expanded state. All values are averages of three measurements taken on a 1.7GHz Pentium M, 1GB, Thinkpad X31.)

As expected, copy-on-reference and ballooning reduced the time to restore. However, copy-on-reference did not slow down execution, even though about 16400 demand fetches were performed that were not performed for copy-on-write. Also surprising was the positive affect ballooning seemed to have on the execution speed. It reduced the time to compile for the copy-on-write case by a factor of 3.

For applications where these performance trade-offs are critical, the flexibility of SBUML’s storage
options make it possible to test various possibilities and choose the one that works best in practice.

6.6 Runtime Overhead

SBUML runtime performance is almost exactly the same as UML itself. Table 10 shows how execution speed compares to native speed on both a single processor notebook computer (1.7GHz Pentium M, 1GB, Thinkpad X31) and a dual processor server (2.8GHz Pentium Xeon, 2GB, Dell Precision 650). Both computers used Fedora Core 1 as the host OS. The overhead of UML’s virtualization technique clearly depends on the task. The ForIARow (from freebench.org) benchmark represents a processor bound task, for which execution speed can approach 99% of native host speed. The second benchmark is a compile of Linux kernel 2.4.24, a task with considerable disk I/O and process creation, which introduces considerable overhead and reduces performance to 20% to 30% of native host speed. These results are similar to performance measures of UML reported in other papers[9][11].

For these tests, SBUML and UML were booted with 256M of memory, run in TT mode, and configured to use the hosts’ tmpfs file systems for the guest virtual memory files. UML has an alternative to TT mode called SKAS mode that offers modest performance gains, however SBUML at this time only supports TT mode.

The SBUML additions have little effect on the speed of UML, because most of SBUML processing takes place during snapshot save and restore. Only two parts of SBUML code are executed during runtime. One part does updates to the status file when processes are forked or killed. The other intercepts reads and writes to the block devices and coordinates with an external process to implement demand fetching (see Section 4.2). Table 11 shows that updates to the process status file cause negligible overhead.

The table also shows that UML’s process creation comes with extremely high overhead. Process creation is especially time consuming for UML because it must create new processes on the host and attach them to the tracing thread on the host. This is a significant source of overhead, but fortunately is not crippling, because most programs do not create an excessive number of processes. For example, the kernel compilation created about 3300 processes, so for the dual CPU case, the fork overhead amounts to about 208 seconds, or about 30% of SBUML’s runtime overhead.

The demand fetch intercept (Table 12) causes significant overhead. However, it is still a minor factor even for kernel compilation, a task that has many file operations. Note that the overhead in the table are for the case when no demand fetch is necessary. When the demand fetch intercept actually performs a demand fetch, performance is naturally much slower due to network access and file block copying.

The runtime space requirements for SBUML are almost the same as for UML when using the COW feature. Enough disk space must exist on the host to hold the virtual machine’s memory, backing disks, and COW disks. SBUML adds the status file, which is 1M in size, and various miscellaneous files that typically take less than 100KB. UML virtual machines can store all machine state in sparse files, so machine size is based on blocks actually used. For example, if machine has a 2GB block device and only 100MB of it is filled, only about 100MB will be consumed on the host.

7 Other Applications

7.1 Build Environment

The idea of using SBUML to host a build environment was already used as one of the example tasks
earlier in the paper. Since we actually use it for that purpose, this task has given us some qualitative evidence for the usefulness of OS-level snapshots. Compiling inside SBUML is slower, and when doing intense debugging with many re-compiles, we tend to copy the debug environment out to the host environment. However, we end up doing most development and all official builds inside SBUML because it is more convenient. It is easier to checkpoint progress and do so with high confidence. The environment can easily be moved from one machine to another. Backups are easy to send to a central HTTP server. When coming back to kernel issues after a month or two break, the snapshot provide incidental but helpful context. For example, the text editor running inside already has the most recently edited files open. Scroll and command line history in the windows helps give context.

More dramatic benefits have been seen when collaborating with others, since even as a full “cow.gz” snapshot, the entire build environment can be copied across the Internet and installed in a few minutes.

Recently, we have moving more functionality inside the official build snapshot to see if it is practical to make it a central repository of everything. For example, an official code management system installed is inside. The HTTP server inside the snapshot is configured to publish various customized builds of SBUML. The “cow.gz” version is now over 1.5GB in size, so the copy-on-reference functionality has become a necessity.

7.2 Downloadable File Server

SBUML runs Linux, and Linux can host a Samba Window-compatible file server. An SBUML snapshot hosting a Samba server can run with only 32MB of memory and can be compressed to about 7.5MB in “cow.gz” form. Once restored, it can be mounted by another computer and any type of information copied in. Then, when saving a snapshot, any combination of SBUML storage optimizations can be used.

One application could be as a way to publish a large data repository of which only a small part is likely to be popular. If the popular part is demand fetched and then saved into a smaller copy-on-reference snapshot, users could download just the smaller copy-on-reference snapshot and have fast local access to a file server that seems to be hosting the entire large data repository.

Another application would be incremental backups. Standard file tools could synchronize local directory contents to the file server. Delta compression and copy-on-reference could be used as appropriate to archive backed up information, but make it so that enough of the file directory information stays in the local snapshot to continue to do incremental backup efficiently.

8 Conclusions and Future Work

SBUML provides an open source infrastructure for exploring how optimization techniques can enable new applications for virtual machine snapshots. Because the snapshot mechanism is an extension to User-Mode Linux, the portability of snapshots is not restricted by the need for hardware virtualization support or special extensions to the host kernel. SBUML currently provides storage optimization techniques that can be combined in flexible ways so that researchers can verify performance trade-offs on actual multi-gigabyte virtual machine images. Prototype demand fetching and streaming features have been integrated, which optimize migration of virtual machine snapshots. Search features allow interdependent snapshots to be distributed freely among HTTP servers so that they can be downloaded and restored automatically.

Additional optimizations and other functionality are planned. For applications that require fast snapshots, copy-on-write snapshots of the RAM file would be useful. The dependencies between snapshots can become complex, therefore automated tools that manage the snapshots need to be carefully designed. A final goal is to create high-level programming interfaces that can hide the optimization techniques while exposing the powerful aspects of OS-Level snapshots for integration into practical solutions.

References