Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems

Oren Laadan
Department of Computer Science
Columbia University

Jason Nieh
Department of Computer Science
Columbia University

Abstract

The ability to checkpoint a running application and restart it later can provide many useful benefits including fault recovery, advanced resources sharing, dynamic load balancing and improved service availability. However, applications often involve multiple processes which have dependencies through the operating system. We present a transparent mechanism for commodity operating systems that can checkpoint multiple processes in a consistent state so that they can be restarted correctly at a later time. We introduce an efficient algorithm for recording process relationships and correctly saving and restoring shared state in a manner that leverages existing operating system kernel functionality. We have implemented our system as a loadable kernel module and user-space utilities in Linux. We demonstrate its ability on real-world applications to provide transparent checkpoint-restart functionality without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Our results show checkpoint and restart times 3 to 55 times faster than OpenVZ and 5 to 1100 times faster than Xen.

1 Introduction

Application checkpoint-restart is the ability to save the state of a running application to secondary storage so that it can later resume its execution from the time at which it was checkpointed. Checkpoint-restart can provide many potential benefits, including fault recovery by rolling back an application to a previous checkpoint, better application response time by restarting applications from checkpoints instead of from scratch, and improved system utilization by stopping long running computationally intensive jobs during execution and restarting them when load decreases. An application can be migrated by checkpointing it on one machine and restarting it on another providing further benefits, including fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that applications can continue to run with minimal downtime.

Many important applications consist of multiple cooperating processes. To checkpoint-restart these applications, not only must application state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Operating system process state including shared resources and various identifiers that define process relationships such as group and session identifiers must be saved and restored correctly. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support the large existing installed base of applications running on commodity operating systems.

Zap [16] is a system that provides transparent checkpoint-restart of unmodified applications that may be composed of multiple processes on commodity operating systems. The key idea is to introduce a thin virtualization layer on top of the operating system that encapsulates a group of processes in a virtualized execution environment and decouples them from the operating system. This layer presents a host-independent virtualized view of the system so that applications can make full use of operating system services and still be checkpointed then restarted at a later time on the same machine or a different one. While previous work [16] presents key aspects of Zap's design, it did not describe a number of important engineering issues in building a robust checkpoint-restart system. In particular, a key issue that was not discussed is how to transparently checkpoint multiple processes such that they can be restarted in a globally consistent state.

Building on Zap, we address this consistency issue and discuss in detail for the first time key design issues in building a transparent checkpoint-restart mechanism for multiple processes on commodity operating systems. We combine a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for checkpoint-restart. We introduce a novel algorithm for accounting for process relationships that correctly saves and restores all process state in a globally consistent manner. This algorithm is crucial for enabling transparent checkpoint-restart of interactive graphical applications and correct job control. We introduce an efficient algorithm for identifying and accounting for shared resources and correctly saving and restoring such shared state across cooperating processes. To

reduce application downtime during checkpoints, we also provide a copy-on-write mechanism that captures a consistent checkpoint state and correctly handles shared resources across multiple processes.

We have implemented a checkpoint-restart prototype as a set of user-space utilities and a loadable kernel module in Linux that operates without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Our measurements on a wide range of real-world server and desktop Linux applications demonstrate that our prototype can provide fast checkpoint and restart times with application downtimes of less than a few tens of milliseconds. Comparing against commercial products, we show up to 12 times faster checkpoint times and 55 times faster restart times than OpenVZ [15], another operating system virtualization approach. We also show up to 550 times faster checkpoint times and 1100 faster restart times than Xen [5], a hardware virtualization approach.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides background on the Zap virtualization architecture. Section 4 provides an overview of our checkpoint-restart architecture. Section 5 discusses how processes are quiesced to provide a globally consistent checkpoint across multiple processes. Section 6 presents the algorithm for saving and restoring the set of process relationships associated with a checkpoint. Section 7 describes how shared state among processes is accounted for during checkpoint and restart. Section 8 presents experimental results on server and desktop applications. Finally, we present some concluding remarks.

2 Related Work

Many application checkpoint-restart mechanisms have been proposed [17, 22, 23]. Application-level mechanisms are directly incorporated into the applications, often with the help of languages, libraries, and preprocessors [2, 7]. These approaches are generally the most efficient, but they are non-transparent, place a major burden on the application developer, may require the use of nonstandard programming languages [8], and cannot be used for unmodified or binary-only applications.

Library checkpoint-restart mechanisms [18, 28] reduce the burden on the application developer by only requiring that applications be compiled or relinked against special libraries. However, such approaches do not capture important parts of the system state, such as interprocess communication and process dependencies through the operating system. As a result, these approaches are limited to being used with applications that severely restrict their use of operating system services.

Operating system checkpoint-restart mechanisms utilize kernel-level support to provide greater application transparency. They do not require changes to the application source code nor relinking of the application object code, but they do typically require new operating systems [9, 10] or invasive kernel modifications [15, 21, 24, 26] in commodity operating systems. None of these approaches checkpoints multiple processes consistently on unmodified commodity operating systems. Our work builds on Zap [16] to provide transparent checkpoint-restart functionality by leveraging loadable kernel module technology and operating system virtualization. The operating system virtualization approach introduced in Zap is also becoming popular for providing isolation containers for groups of processes while allowing scalable use of system resources [15, 20, 12].

Hardware virtualization approaches such as Xen [5] and VMware [29] use virtual machine monitors (VMMs) [19] that can enable an entire operating system environment, both operating system and applications, to be checkpointed and restarted. VMMs can support transparent checkpoint-restart of both Linux and Windows operating systems. However, because they operate on entire operating system instances, they cannot provide finer-granularity checkpoint-restart of individual applications decoupled from operating system instances, resulting in higher checkpoint-restart overheads and differences in how these mechanisms can be applied.

3 Virtualization Architecture

To enable checkpoint-restart, we leverage Zap's operating system virtualization. Zap introduces a thin virtualization layer between applications and the operating system to decouple applications from the underlying host. The virtualization layer provides a pod (PrOcess Domain) virtual machine abstraction which encapsulates a set of processes in a self-contained unit that can be isolated from the system, checkpointed to secondary storage, and transparently restarted later. This is made possible because each pod has its own virtual private namespace, which provides the only means for processes to access the underlying operating system. To guarantee correct operation of unmodified applications, the pod namespace provides a traditional environment with unchanged application interfaces and access to operating system services and resources.

Operating system resource identifiers, such as process IDs, must remain constant throughout the life of a process to ensure its correct operation. However, when a process is checkpointed and later restarted, possibly on a different operating system instance, there is no guarantee that the system will provide the same identifiers to the restarted process; those identifiers may in fact be in use by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names. Names within a pod are trivially assigned in a unique manner in the same way that traditional oper-

ating systems assign names, but such names are localized to the pod. These virtual private names are not changed when a pod is restarted to ensure correct operation. Instead, pod virtual resources are transparently remapped to real operating system resources.

In addition to providing a private virtual namespace for processes in a pod, our virtualization approach provides three key properties so that it can be used as a platform for checkpoint-restart. First, it provides mechanisms to synchronize the virtualization of state across multiple processes consistently with the occurrence of a checkpoint, and upon restart. Second, it allows the system to select predetermined virtual identifiers upon the allocation of resources when restarting a set of processes so that those processes can reclaim the same set of virtual resources they had used prior to the checkpoint. Third, it provides virtualization interfaces that can be used by checkpoint and restart mechanisms to translate between virtual identifiers and real operating system resource identifiers. During normal process execution, translating between virtual and real identifiers is private to the virtualization layer. However, during checkpoint-restart, the checkpoint and restart functions may also need to request such translations to match virtual and real namespaces.

4 Checkpoint-Restart Architecture

We combine pod virtualization with a mechanism for checkpointing and restarting multiple cooperating processes in a pod. For simplicity, we describe the checkpoint-restart mechanism assuming a shared storage infrastructure across participating machines. In this case, filesystem state is not generally saved and restored as part of the pod checkpoint image to reduce checkpoint image size. Available filesystem snapshot functionality [14, 6] can be used to also provide a checkpointed filesystem image. We focus only on checkpointing process state; details on how to checkpoint filesystem, network, and device state are beyond the scope of this paper.

Checkpoint: A checkpoint is performed in the following steps:

1. *Quiesce pod*: To ensure that a globally consistent checkpoint [3] is taken of all the processes in the pod, the processes are quiesced. This forces the processes to transfer from their current state to a controlled standby state to freeze them for the duration of the checkpoint.
2. *Record pod properties*: Record pod configuration information, in particular filesystem configuration information indicating where directories private to the pod are stored on the underlying shared storage infrastructure.
3. *Dump process forest*: Record process inheritance relationships, including parent-child, sibling, and process session relationships.
4. *Record globally shared resources*: Record state associated with shared resources not tied to any specific process, such as System V IPC state, pod's network address, hostname, system time and so forth.
5. *Record process associated state*: Record state associated with individual processes and shared state attached to processes, including process run state, program name, scheduling parameters, credentials, blocking and pending signals, CPU registers, FPU state, `ptrace` state, filesystem namespace, open files, signal handling information, and virtual memory.
6. *Continue pod*: Resume the processes in the pod once the checkpoint state has been recorded to allow the processes to continue executing, or terminate the processes and the pod if the checkpoint is being used to migrate the pod to another machine. (If a filesystem snapshot is desired, it is taken prior to this step.)
7. *Commit data*: Write out buffered recorded data (if any) to storage (or to the network) and optionally force flush of the data to disk.

To reduce application downtime due to the pod being quiesced, we employ a lazy approach in which the checkpoint data is first recorded and buffered in memory. We defer writing it out to storage (or to the network) until after the pod is allowed to continue, thereby avoiding the cost of expensive I/O operations while the pod is quiesced. Since allocation of large memory chunks dynamically can become expensive too, buffers are preallocated before the pod is quiesced, based on an estimate for the required space. The data accumulated in the buffer is eventually committed in step 7 after the pod has been resumed.

We use a standard copy-on-write (COW) mechanism to keep a reference to memory pages instead of recording an explicit copy of each page. This helps to reduce memory pressure and avoids degrading cache performance. It reduces downtime further by deferring the actual memory to memory copy until when the page is either modified by the application or finally committed to storage, whichever occurs first. Using COW ensures that a valid copy of a page at time of checkpoint remains available if the application modifies the page after the pod has resumed operation but before the data has been committed. Pages that belong to shared memory regions cannot be made copy-on-write, and are handled by recording an explicit copy in the checkpoint buffer. Note that we do not use the `fork` system call for creating a COW clone [10, 18, 26] because its semantics require a process to execute the call itself. This cannot be done while the process is in a controlled standby state, making it difficult to ensure global consistency when checkpointing multiple processes.

The contents of files are not normally saved as part of the checkpoint image since they are available on the filesystem. An exception to this rule are open files that

have been unlinked. They need to be saved during checkpoint since they will no longer be accessible on the filesystem. If large unlinked files are involved, saving and restoring them as part of the checkpoint image incurs high overhead since the data needs to be both read then written during both checkpoint and restart. To avoid these data transfer costs, we instead relink the respective inode back to the filesystem. To maintain the illusion that the file is still unlinked, it is placed in a protected area that is unaccessible to processes in the pod. If relinking is not feasible, such as if a FAT filesystem implementation is used that does not support hard links, we cannot relink but instead store the unlinked file contents in a separate file in a protected area. This is still more efficient than including the data as part of the checkpoint image.

Restart: Complementary to the checkpoint, a restart is performed in the following steps:

1. *Restore pod properties*: Create a new pod, read pod properties from the checkpoint image and configure the pod, including restoring its filesystem configuration.
2. *Restore process forest*: Read process forest information, create processes at the roots of the forest, then have root processes recursively create their children.
3. *Restore globally shared resources*: Create globally shared resources, including creating the necessary virtual identifiers for those resources.
4. *Restore process associated state*: Each created process in the forest restores its own state then quiesces itself until all other processes have been restored.
5. *Continue*: Once all processes in the pod are restored, resume them so they can continue execution.

Before describing the checkpoint-restart steps in further detail, we first discuss three key aspects of their overall structure: first, whether the mechanism is implemented at kernel-level or user-level; second, whether it is performed within the context of each process or by an auxiliary process; and finally the ordering of operations to allow streaming of the checkpoint data.

Kernel-level vs user-level: Checkpoint-restart is performed primarily at kernel-level, not at user-level. This provides application transparency and allows applications to make use of the full range of operating system services. The kernel-level functionality is explicitly designed so that it can be implemented as a loadable module without modifying, recompiling, or relinking the operating system kernel. To simplify process creation, we leverage existing operating system services to perform the first phase of the restart algorithm at user-level. The standard process creation system call `fork` is used to reconstruct the process forest.

In context vs auxiliary: Processes are checkpointed from outside of their context and from outside of the pod using a separate auxiliary process, but processes are restarted from inside the pod within the respective context of each process. We use an auxiliary process that runs outside of the pod for two reasons. First, since all processes within the pod are checkpointed, this simplifies the implementation by avoiding the need to special case the auxiliary process from being checkpointed. Second, the auxiliary process needs to make use of unvirtualized operating system functions to perform parts of its operation. Since processes in a pod are isolated from processes outside of the pod when using the standard system call interface [16], the auxiliary process uses a special interface for accessing the processes inside of the pod to perform the checkpoint.

Furthermore, checkpoint is done not within the context of each process for four reasons. First, using an auxiliary process makes it easier to provide a globally consistent checkpoint across multiple processes by simply quiescing all processes then taking the checkpoint; there is no need to run each process to checkpoint itself and attempt to synchronize their checkpoint execution. Second, a set of processes is allowed to be checkpointed at any time and not all of the processes may be runnable. If a process cannot run, for example if it is stopped at a breakpoint as a result of being traced by another process, it cannot perform its own checkpoint. Third, to have checkpoint code run in the process context, the process must invoke this code involuntarily since we do not require process collaboration. While this can be addressed by introducing a new specific signal to the kernel [11] that is served within the kernel, it requires kernel modifications and cannot be implemented by a kernel module. Fourth, it allows for using multiple auxiliary processes concurrently (with simple synchronization) to accelerate the checkpoint operation.

Unlike checkpoint, restart is done within the context of the process that is restarted for two reasons. First, operating within process context allows us to leverage the vast majority of available kernel functionality that can only be invoked from within that context, including almost all system calls. Although checkpoint only requires saving process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly simplifies the restart mechanism. Second, because the restart mechanism creates a new set of processes that it completely controls on restart, it is simple to cause those processes to run, invoke the restart code, and synchronize their operations as necessary. As a result, the complications with running in process context during checkpoint do not arise during restart.

More specifically, restart is done by starting a supervisor process which creates and configures the pod, then injects itself into the pod. Once it is in the pod, the supervisor forks the processes that constitute the roots of the process forest. The root processes then create their chil-

dren, which recursively create their descendants. Once the process forest has been constructed, all processes switch to operating at kernel-level to complete the restart. The supervisor process first restores globally shared resources, then each process executes concurrently to restore its own process context from the checkpoint. When all processes have been restored, the restart completes and the processes are allowed to resume normal execution.

Data streaming: The steps in the checkpoint-restart algorithm are ordered and designed for streaming to support their execution using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. For example, the checkpoint can be directly streamed from one machine to another across the network and then restarted. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in extremely flexible and extensible architecture. Example filters include encryption, signature/validation, compression, and conversion between operating system versions.

5 Quiescing Processes

Quiescing a pod is the first step of the checkpoint, and is also the last step of the restart as a means to synchronize all the restarting processes and ensure they are all completely restored before they resume execution. Quiescing processes at checkpoint time prevents them from modifying system state, and thus prevents inconsistencies from occurring during the checkpoint. Quiescing also puts processes in a known state from which they can easily be restarted. Without quiescing, checkpoints would have to capture potentially arbitrary restart points deep in the kernel, wherever a process might block.

Processes are quiesced by sending them a `SIGSTOP` signal to force them into the stopped state. A process is normally either running in user-space or executing a system call in the kernel, in which case it may be blocked. Unless we allow intrusive changes to the kernel code, signaling a process is the only method to force a process from user-space into the kernel or to interrupt a blocking system call. The `SIGSTOP` signal is guaranteed to be delivered and not ignored or blocked by the process. Using signals simplifies quiescing as signal delivery already handles potential race conditions, particularly in the presence of threads.

Using `SIGSTOP` to force processes into the stopped state has additional benefits for processes that are running or blocked in the kernel, which will handle the `SIGSTOP` immediately prior to returning to user mode. If a process is in a non-interruptible system call or handling an interrupt or trap, it will be quiesced after the kernel processing of the respective event. The processing time for these events is generally small. If a process is in an interruptible system call, it will immediately return and handle the signal. The effect of the signal is transparent as the system call will in most cases be automatically restarted, or in some cases return a suitable error code that the caller should be prepared to handle. The scheme is elegant in that it builds nicely on the existing semantics of Unix/Linux, and ideal in that it forces processes to a state with only a trivial kernel stack to save and restore on checkpoint-restart.

In quiescing a pod, we must be careful to also handle potential side effects [27] that can occur when a signal is sent to a process. For example, the parent of a process is always notified by a signal when either `SIGSTOP` or `SIGCONT` signals are handled by the process, and a process that is traced always notifies the tracer process about every signal received. While these signals can normally occur on a Unix system, they may have undesirable side effects in the context of checkpoint-restart. We address this issue by ensuring that the virtualization layer masks out signals that are generated as a side effect of the quiesce and restore operations.

The use of `SIGSTOP` to quiesce processes is sufficiently generic to handle every execution scenario with the exception of three cases in which a process may already be in a state similar to the stopped state. First, a process that is already stopped does not need to be quiesced, but instead needs to be marked so that the restart correctly leaves it in the stopped state instead of sending it a `SIGCONT` to resume execution.

Second, a process executing the `sigsuspend` system call is put in a deliberate suspended state until it receives a signal from a given set of signals. If a process is blocked in that system call and then checkpointed, it must be accounted for on restart by having the restarting process call `sigsuspend` as the last step of the restart, instead of stopping itself. Otherwise, it will resume to user mode without really having received a valid signal.

Third, a process that is traced via the `ptrace` mechanism [13] will be stopped for tracing at any location where a trace event may be generated, such as entry and exit of system calls, receipt of signals, events like fork, vfork, exec, and so forth. Each such trace point generates a notification to the controlling process. The ptrace mechanism raises two issues. First, a `SIGSTOP` that is sent to quiesce a pod will itself produce a trace event for traced processes, which—while possible in Unix—is undesirable from a look-and-feel point of view (imagine your debugger reporting spurious signals received by the program). This is solved by making traced process exempt from quiesce (as they already are stopped) and from continue (as they should remain stopped). Second, like `sigsuspend`, the system must record at which point the process was traced, and use this data upon restart. The action to be taken at restart varies with the specific trace event. For instance, for system call entry, the restart code will not stop the pro-

cess but instead cause it to enter a ptrace-like state in which it will block until told to continue. Only then will it invoke the system call directly, thus avoiding an improper trigger of the system call entry event.

6 Process Forest

To checkpoint multiple cooperating processes, it is crucial to capture a globally consistent state across all processes, and preserve process dependencies. Process dependencies include *process hierarchy* such as parent-child relationships, identifiers that define *process relationships* such as group and session identifiers (PGIDs and SIDs respectively), and *shared resources* such as common file descriptors. The first two are particularly important for interactive applications and other activities that involve job control. All of these dependencies must be checkpointed and restarted correctly. The term *process forest* encapsulates these three components: hierarchy, relationships and resources sharing. On restart, the restored process forest must satisfy all of the constraints imposed by process dependencies. Otherwise, applications may not work correctly. For instance, incorrect settings of SIDs will cause incorrect handling of signals related to terminals (including `xterm`), as well as confuse job control since PGIDs will not be restored correctly either.

A useful property of our checkpoint-restart algorithm is that the restart phase can recreate the process forest using standard system calls, simplifying the restart process. However, system calls do not allow process relationships and identifiers to be changed arbitrarily after a process has already been created. A key observation in devising suitable algorithms for saving and restoring the process forest is determining what subset of dependencies require a priori resolution, then leaving others to be setup retroactively.

There are two primary process relationships that must be established as part of process creation to correctly construct a process forest. The key challenge is preserving session relationships. Sessions must be inherited by correctly ordering process creation because the operating system interface only allows a process to change its own session, to change it just once, and to change it to a new session and become the leader. The second issue is preserving thread group relationships, which arises in Linux because of its threading model which treats threads as special processes; this issue does not arise in operating system implementations which do not treat threads as processes. Hereinafter we assume the threading model of Linux 2.6 in which threads are grouped into thread groups with a single thread group leader, which is always the first thread in the group. A thread must be created by its thread group leader because the operating system provides no other way to set the thread group. Given the correct handling of session identifiers and thread groups, other relationships and shared resources can be manipulated after process creation using the operating system interface, and are hence simply assigned once all processes have been created.

Since these two process relationships must be established at process creation time to correctly construct a process forest, the order in which processes are created is crucial. Simply reusing the parent-child relationships maintained by the kernel to create a matching process forest is not sufficient since the forest depends on more than the process hierarchy at the time of checkpoint. For example, it is important to know the original parent of a process to ensure that it inherits its correct SID, however since orphaned children are promptly re-parented to `init`, the information about their original parent is lost. While one could log all process creations and deletions to later determine the original parent, this adds unnecessary runtime overhead and complexity.

We introduce two algorithms—`DumpForest` and `MakeForest`—that use existing operating system interfaces to efficiently save and restore the process forest, respectively. The algorithms correctly restore a process forest at restart that is the same as the original process forest at checkpoint. However, they do not require any state other than what is available at checkpoint time because they do not necessarily recreate the matching process forest in the same way as the original forest was created.

6.1 DumpForest Algorithm

The `DumpForest` algorithm captures the state of the process forest in two passes. It runs in linear time with the number of process identifiers in use in a pod. A process identifier is in use even if a process has terminated as long as the identifier is still being used, for example as an SID for some session group. The first pass scans the list of processes identifiers within the pod and fills in a table of entries; the table is not sorted. Each entry in the table represents a PID in the forest. The second pass records the process relationships by filling in information in each table entry. A primary goal of this pass is to determine the creating parent (*creator*) of each process, including which processes have `init` as their parent. At restart, those processes will be created first to serve as the roots of the forest, and will recursively create the remaining processes as instructed by the table.

Each entry in the table consists of the following set of fields: status, PID, SID, thread group identifier, and three pointers to the table locations of the entry's creator, next sibling, and first child processes to be used by `MakeForest`. Note that these processes may not necessarily correspond to the parent, next sibling, and first child processes of a process at the time of checkpoint. Table 1 lists the possible flags for the status field. In particular, Linux allows a process to be created by its sibling, thereby

inheriting the same parent, which differs from traditional parent-child only fork creation semantics; a `Sibling` flag is necessary to note this case.

Flag	Property of Table Entry
Dead	Corresponds to a terminated process
Session	Inherits ancestor SID before parent changes its own
Thread	A thread but not a thread group leader
Sibling	Created by sibling via parent inheritance

Table 1: Possible flags in the status field

6.1.1 Basic Algorithm

For simplicity, we first assume no parent inheritance in describing the `DumpForest` algorithm. The first pass of the algorithm initializes the PID and SID fields of each entry according to the process it represents, and all remaining fields to be empty. As shown in Algorithm 1, the second pass calls `FindCreator` on each table entry to populate the empty fields and alter the status field as necessary. The algorithm can be thought of as determining under what circumstances the current parent of a process at time of checkpoint cannot be used to create the process at restart.

The algorithm looks at each table entry and determines what to do based on the properties of the entry. If the entry is a thread and not the thread group leader, we mark its creator as the thread group leader and add `Thread` to its status field so that it must be created as a thread on restart. The thread group leader can be handled as a regular process, and hence is treated as part of the other cases.

Otherwise, if the entry is a session leader, this is an entry that at some point called `setsid`. It does not need to inherit its session from anyone, so its creator can just be set to its parent. If a pod had only one session group, the session leader would be at the root of the forest and its parent would be `init`.

Otherwise, if the entry corresponds to a dead process (no current process exists with the given PID), the only constraint that must be satisfied is that it inherit the correct session group from its parent. Its creator is just set to be its session leader. The correct session group must be maintained for a process that has already terminated because it may be necessary to have the process create other processes before terminating itself, to ensure that those other processes have their session groups set correctly.

Otherwise, if the entry corresponds to an orphan process, it cannot inherit the correct session group from `init`. Therefore, we add a placeholder process in the table whose function on restart is to inherit the session group from the entry's session leader, create the process, then terminate so that the process will be orphaned. The placeholder is assigned an arbitrary PID that is not already in the table, and the SID identifying the session. To remember to terminate the placeholder process, the placeholder entry's status field is marked `Dead`.

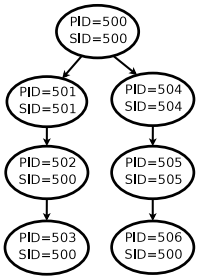
Algorithm 1 DumpForest (second pass)

```
 1: Procedure DumpForest
 2: for all entries ent in the table do
 3:   if (ent.creator == NIL) then
 4:     call FindCreator(ent)
 5:   end if
 6: end for
 7: End
 8:
 9: Procedure FindCreator(ent)
10: leader ← session leader entry
11: if ent is a dead process then
12:   parent ← init
13:   ent.status |= Dead
14: else
15:   parent ← parent process entry
16: end if
17: if ent is thread (but not thread group leader) then
18:   ent.creator ← thread group leader
19:   ent.status |= Thread
20: else if (ent == leader) then
21:   ent.creator ← parent
22: else if (ent.status & Dead) then
23:   ent.creator ← leader
24: else if (parent == init) then
25:   call AddPlaceHolder(ent, leader)
26: else if (ent.sid == parent.sid) then
27:   ent.creator ← parent
28: else
29:   ent.creator ← parent
30:   sid ← ent.sid
31:   repeat
32:     ent.status |= Session
33:     if (ent.creator == init) then
34:       call AddPlaceHolder(ent, leader)
35:     end if
36:     ent ← ent.creator
37:     if (ent.creator == NIL) then
38:       call FindCreator(ent)
39:     end if
40:   until (ent.sid == sid) or (ent.status & Session)
41: end if
42: End
43:
44: Procedure AddPlaceHolder(ent, leader)
45: add new entry new to table
46: new.creator ← leader
47: new.status |= Dead
48: ent.creator ← new
49: End
```

(Note: when the creator field is set, the matching child and sibling fields are adjusted accordingly; details are omitted for brevity).

Otherwise, if the entry's SID is equal to its parent's, the only constraint that must be satisfied is that it inherit the correct session group from its parent. This is simply done by setting its creator to be its parent.

If none of the previous cases apply, then the entry corresponds to a process which is not a session leader, does not have a session group the same as its parent, and therefore whose session group must be inherited from an ancestor further up the process forest. This case arises because the process was forked by its parent before the parent changed

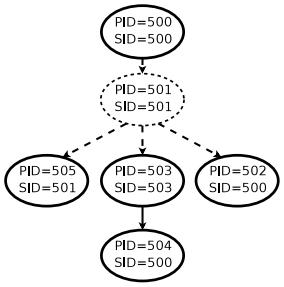


(a) Scenario

PID	Creator	Status
500	init	
501	500	
502	501	*Session*
503	502	
504	500	
505	504	*Session*
506	505	*Session*

(b) Resulting table

Figure 1: Simple process forest



(a) Scenario

PID	Creator	Status
500	init	
501	init	
997	500	*Dead*
502	997	
998	500	*Dead*
503	998	
504	503	*Session*
999	501	*Dead*
505	999	

(b) Resulting table

Figure 2: Process forest with deletions

its own SID. Its creator is set to be its parent, but we also mark its status field `Session`, indicating that at restart the parent will need to fork before (potentially) creating a new session. When an entry is marked with `Session`, it is necessary to propagate this attribute up its ancestry hierarchy until an entry with that session group is located. In the worst case, this would proceed all the way to the session leader. This is required for the SID to correctly descend via inheritance to the current entry. Note that going up the tree does not increase the runtime complexity of the algorithm because traversal does not proceed beyond entries that already possess the `Session` attribute.

If the traversal fails to find an entry with the same SID, it will stop at an entry that corresponds to a leader of another session. This entry must have formerly been a descendant of the original session leader. Its creator will have already been set `init`. Because we now know that it needs to pass the original SID to its own descendants, we re-parent the entry to become a descendant of the original session leader. This is done using a placeholder in a manner similar to how we handle orphans that are not session leaders.

6.1.2 Examples

Figure 1 illustrates the output of the algorithm on a simple process forest. Figure 1a shows the process forest at checkpoint time. Figure 1b shows the table generated by `DumpForest`. The algorithm first creates a table of seven entries corresponding to the seven processes, then proceeds to determine the creator of each entry. Processes 502, 505, and 506 have their `Session` attributes set, since they must be forked off *before* their parents' session identifiers are changed. Note that process 505 received this flag by propagating it up from its child process 506.

Figure 2 illustrates the output of the algorithm on a process forest with a missing process, 501, which exited before the checkpoint. Figure 2a shows the process forest at checkpoint time. Figure 2b shows the table generated by `DumpForest`. While the algorithm starts with six entries in the table, the resulting table has nine entries since three placeholder processes, 997, 998, and 999, were added to maintain proper process relationships. Observe that process 503 initially has its creator set to `init`, but is reparented to the placeholder 998 as part of propagating its child's `Session` attribute up the tree.

6.1.3 Supporting Linux Parent Inheritance

We now discuss modifications to the basic `DumpForest` algorithm for handling the unique parent inheritance feature in Linux which allows a process to create a sibling. To inherit session relationships correctly, parent inheritance must be accounted for in determining the creators of processes that are not session leaders.

If its session leader is alive, we can determine that a process was created by its sibling if its parent is the creator of the session leader. If the session leader is dead, this check will not work since its parent is now `init` and there is no longer any information about its original parent. After a process dies, there is no easy way to determine its original parent in the presence of parent inheritance.

To provide the necessary information, we instead record the session a process inherits when it is created, if and only if the process is created with parent inheritance and it is a sibling of the session group leader. This *saved-SID* is stored as part of the process's virtualization data structure so that it can be used later if the process remains alive when the forest needs to be saved. A process created with parent inheritance is a sibling of the session group leader if either its creator is the session group leader, or its creator has the same saved-SID recorded, since the sibling relationship is transitive.

To support parent inheritance, we modify Algorithm 1 by inserting a new conditional after the check for whether an entry's SID is equal to its parent's and before the final else clause in `FindCreator`. The conditional examines whether an entry's saved-SID has been set. If it has been set and there exists another entry in the process forest table whose PID is equal to this saved-SID, the entry's status field is marked `Sibling` so that it will be created with parent inheritance on restart. The entry's creator is set to the entry that owns that PID, which is leader of the

session identified by the saved-SID. Finally, the creator of this session leader is set to be the parent of the current process, possibly re-parenting the entry if its creator had already been set previously.

6.2 MakeForest Algorithm

Given the process forest data structure, the `MakeForest` algorithm is straightforward, as shown in Algorithm 2. It reconstructs the process hierarchy and relationships by executing completely in user mode using standard system calls, minimizing dependencies on any particular kernel internal implementation details. The algorithm runs in linear time with the number of entries in the forest. It works in a recursive manner by following the instructions set forth by the process forest data structure. `MakeForest` begins with a single process that will be used in place of `init` to fork the processes that have `init` set as their creator. Each process then creates its own children.

The bulk of the algorithm loops through the list of children of the current process three times during which the children are forked or cleaned up. Each child that is forked executes the same algorithm recursively until all processes have been created. In the first pass through the list of children, the current process spawns children that are marked `Session` and thereby need to be forked before the current session group is changed. The process then changes its session group if needed. In the second pass, the process forks the remainder of the children. In both passes, a child that is marked `Thread` is created as a thread and a child that is marked `Sibling` is created with parent inheritance. In the third pass, terminated processes and temporary placeholders are cleaned up. Finally, the process either terminates if it is marked `Dead` or calls `RestoreProcessState()` which does not return. `RestoreProcessState()` restores the state of the process to the way it was at the time of checkpoint.

Algorithm 2 MakeForest

```
 1: Procedure MakeForest
 2: for all entries ent in the table do
 3:    if ent.creator == init then
 4:       call ForkChildren(ent)
 5:    end if
 6: end for
 7: End
 8:
 9: Procedure ForkChildren(ent)
10: for all children cld of ent do
11:    if (cld.status & Session) then
12:       call ForkChild(cld)
13:    end if
14: end for
15: if (ent.sid == ent.pid) then
16:    call setsid()
17: end if
18: for all children cld of ent do
19:    if ¬(cld.status & Session) then
20:       call ForkChild(cld)
21:    end if
22: end for
23: for all children cld of ent do
24:    if (cld.status & Dead) then
25:       call waitpid(cld.pid)
26:    end if
27: end for
28: if (ent.status & Dead) then
29:    call exit()
30: else
31:    call RestoreProcessState()
32: end if
33: End
34:
35: Procedure ForkChild(cld)
36: if (cld.status & Thread) then
37:    pid = fork_thread()
38: else if (cld.status & Sibling) then
39:    pid = fork_sibling()
40: else
41:    pid = fork()
42: end if
43: if pid == 0 then
44:    call ForkChildren(cld)
45: end if
46: End
```

7 Shared Resources

After the process hierarchy and relationships have been saved or restored, we process operating system resources that may be shared among multiple processes. They are either *globally shared* at the pod level, such as IPC identifiers and pseudo terminals, or *locally shared* among a subset of processes, such as virtual memory, file descriptors, signal handlers and so forth. As discussed in Section 4, globally shared resources are processed first, then locally shared resources are processed. Shared resources may be referenced by more than one process, yet their state need only be saved once. We need to be able to uniquely identify each resource, and to do so in a manner independent of the operating system instance to be able to restart on another instance.

Every shared resource is represented by a matching kernel object whose kernel address provides a unique identifier of that instance within the kernel. We represent each resource by a tuple of the form $\ll$`Address`,`Tag`$\gg$, where `address` is its kernel address, and `tag` is a serial number that reflects the order in which the resources were encountered (counting from 1 and on). Tags are, therefore, unique logical identifiers for resources. The tuples allow the same resource representation to be used for both checkpoint and restart mechanisms, simplifying the overall implementation. During checkpoint and restart, they are stored in an associative memory in the kernel, enabling fast translation between physical and logical identifiers. Tuples are registered into the memory as new resources are discov-

ered, and discarded once the entire checkpoint (or restart) is completed. This memory is used to decide whether a given resource (physical or logical for checkpoint or restart respectively) is a new instance or merely a reference to one already registered. Both globally and locally shared resources are stored using the same associative memory.

During checkpoint, as the processes within the pod are scanned one by one, the resources associated with them are examined by looking up their kernel addresses in the associative memory. If the entry is not found (that is, a new resource has been detected) we allocate a new (unique) tag, register the new tuple and record the state of that resource. The tag is included as part of that state. On the other hand, if the entry is found, it means that the resource is shared and has been already dealt with earlier. Hence it suffices to record its tag for later reference. Note that the order of the scan is insignificant.

During restart, the algorithm restores the state of the processes and the resources they use. The data is read in the same order as has been written originally, ensuring that the first occurrence of each resource is accompanied with its actual recorded state. For each resource identifier, we examine whether the tag is already registered, and if not we create a new instance of the required resource, restore its state from the checkpoint data, and register an appropriate tuple, with the address field set to the kernel address that corresponds to the new instance. If a tuple with the specified tag is found, we locate the corresponding resource with the knowledge of its kernel address as taken from the tuple.

Nested shared objects Nested sharing occurs in the kernel when a common resource is referenced by multiple distinct resources rather than by processes. One example are objects that represent a `FIFO` in the filesystem, as a `FIFO` is represented by a single inode which is in turn pointed to by file descriptors of reader and writer ends. Another example is a single backing file that is mapped multiple times within distinct address spaces. In both examples shared objects—file descriptors and address spaces respectively—refer to a shared object, yet may themselves be shared by multiple processes.

Nested sharing is handled similarly to simple sharing. To ensure consistency we enforce an additional rule, namely that a nested object is always recorded prior to the objects that point to it. For instance, when saving the state of a file descriptor that points to a `FIFO`, we first record the state of the `FIFO`. This ensures that the tuples for the nested resource exist in time for the referring object.

Compound shared objects Many instances of nested objects involve a pair of coupled resources. For example, a single pipe is represented in the kernel by two distinct inodes that are coupled in a special form, and Unix domain sockets can embody up to three disparate inodes for the listening, accepting and connecting sockets. We call such objects *compound objects*. Unlike unrelated resources, compound objects have two or more internal elements that are created and interlinked with the invocation of the appropriate kernel subroutine(s) such that their lifespans are correlated, e.g. the two inodes that constitute a pipe.

We consistently track a compound object by capturing the state of the entire resource including all components at once, at the time it is first detected, regardless of through which component it was referred. On restart, the compound object will be encountered for the first time through some component, and will be reconstructed in its entirety, including all other components. Then only the triggering component (the one that was encountered) will need to be attached to the process that owns it. The remaining components will linger unattached until they are claimed by their respective owners at a later time.

The internal ordering of the elements that compose a compound object may depend on the type of the object. If the object is symmetric, such as `socketpairs`, its contents may be saved at an arbitrary order. Otherwise, the contents are saved in a certain order that is particularly designed to facilitate the reconstruction of the object during restart. For example, the order for pipes is first the read-side followed by the write-side. The order for Unix domain sockets begins with the listening socket (if it exists), followed by the connecting socket and finally the accepting socket. This order reflects the sequence of actions that is required to rebuild such socket-trios: first create a listening socket, then a socket that connects to it, and finally the third socket by accepting the connection.

Memory sharing Since memory footprint is typically the most dominant factor in determining the checkpoint image size, we further discuss how recording shared resources is done in the case of memory. A memory region in a process's address space can be classified along two dimensions, one is whether it is mapped to a backing file or anonymous, and the other is whether it is private to some address space or shared among multiple ones. For example, text segments such as program code and shared libraries are mapped and shared, IPC shared memory is anonymous and shared, the data section is mapped and private, and the heap and stack are anonymous and private.

Memory sharing can occur in any of these four cases. Handling regions that are shared is straightforward. If a region is mapped and shared, it does not need to be saved since its contents are already on the backing file. If a region is anonymous and shared, it is treated as a normal shared object so that its contents are only saved once. Handling regions that are private is more subtle. While it appears contradictory to have memory sharing with private memory regions, sharing occurs due to the kernel's COW optimization. When a process forks, the kernel defers the creation of a separate copy of the pages for the newly created process until one of the processes sharing the common

Name	Description
apache	apache 2.0.55 with 50 threads (default) loaded w/ `httperf 0.8` (rate=1500, num-calls=20)
make	compilation (make -j 5) of Linux kernel tree
mysql	MySQL 4.2.21 loaded w/ standard `sql-bench`
volano	VolanoMark 2.5 w/ Blackdown Java 1.4.2
UML	User Mode Linux w/ 128 MB and Debian 3.0
gnome-base	Gnome 2.8 session with THINC server
gnome-firefox	gnome-base and Firefox 1.04 with 2 browser windows and 3 open tabs in each
gnome-mplayer	gnome-base and MPlayer 1.0pre7-3.3.5 playing an MPEG1 video clip
Microsoft-office	gnome-base and CrossOver Office 5.0 running Microsoft Office XP with 2 Word documents and 1 Powerpoint slide presentation open

Table 2: Application scenarios

memory attempts to modify it. During checkpoint, each page that has been previously modified and belongs to a private region that is marked COW is treated as a nested shared object so that its contents are only saved once. During restart, the COW sharing is restored. Modified pages in either anonymous and private regions or mapped and private regions are treated in this manner.

8 Experimental Results

To demonstrate the effectiveness of our approach, we have implemented a checkpoint-restart prototype as a Linux kernel module and associated user-level tools and evaluated its performance on a wide range of real applications. We also quantitatively compared our prototype with two other commercial virtualization systems, OpenVZ and Xen. OpenVZ provides another operating system virtualization approach for comparison, while Xen provides a hardware virtualization approach for comparison. We used the latest versions of OpenVZ and Xen that were available at the time of our experiments.

The measurements were conducted on an IBM HS20 eServer BladeCenter, each blade with dual 3.06 GHz Intel Xeon^{TM} CPUs, 2.5 GB RAM, a 40 GB local disk, and QLogic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and linked through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre Channel hard drives. Each blade used the GFS cluster filesystem [25] to access a shared SAN. Unless otherwise indicated, the blades were running Debian 3.1 distribution and the Linux 2.6.11.12 kernel.

Table 2 lists the nine application scenarios used for our experiments. The scenarios were running an Apache web server, a kernel compile, a MySQL database server, a volano chat server, an entire operating system at user-level using UML, and four desktop applications scenarios run using a full Gnome X desktop environment with an XFree86 4.3.0.1 server and THINC [1] to provide remote display access to the desktop. The four desktop scenarios were running a baseline environment without additional applications, a web browser, a video player, and a Microsoft Office suite using CrossOver Office. The UML scenario shows the ability to checkpoint and restart an entire operating system instance. The Microsoft Office scenario shows the ability to checkpoint and restart Windows applications using CrossOver Office on Linux.

We measured checkpoint-restart performance by running each of the application scenarios and taking a series of ten checkpoints during their execution. We measured the checkpoint image sizes, number of processes that were checkpointed, checkpoint times, and restart times, then averaged the measurements across the ten checkpoints for each application scenario. Figures 3 to 8 show results for our checkpoint-restart prototype.

Figure 3 shows the average total checkpoint image size, as well as a breakdown showing the amount of data in the checkpoint image attributable to the process forest. The total amount of state that is saved is modest in each case and varies according to the applications executed, ranging from a few MBs on most applications to tens of MBs for graphical desktop sessions. The results show that the total memory in use within the pod is the most prominent component of the checkpoint image size, accounting for over 99% of the image size.

An interesting case is UML, that uses memory mapping to store guest main memory using an unlinked backing file. This file is separate from memory and amounts to 129 MB. By using the optimization for unlinked files as discussed in Section 4 and storing the unlinked files separately on the filesystem, the UML state stored in the checkpoint image can be reduced to roughly 1 MB. The same occurs for CrossOver Office, which also maps additional 16 MB of memory to an unlinked backing file.

Figure 4 shows the average number of processes running within the pod at checkpoints for each application scenario. On average the process forest tracks 35 processes in most scenarios, except for `apache` and `volano` with 169 and 839 processes each, most of which are threads. As Figure 3 shows the process forest always occupies a small fraction of the checkpoint, even for `volano`.

Figure 5 shows the average *total* checkpoint times for each application scenario, which is measured from when the pod is quiesced until the complete checkpoint image is written out to disk. We also show two other measures. Checkpoint *downtime* is the time from when the pod is quiesced until the pod can be resumed; it is the time to record the checkpoint data without committing it to disk. *Sync* checkpoint time is the total checkpoint time plus the time to force flushing the data to disk. Average total checkpoint times are under 600 ms for all application scenarios and as small as 40 ms, which is the case for UML. Comparing

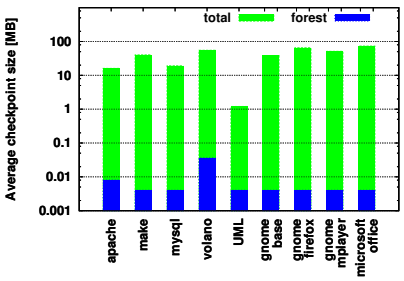


Figure 3: Average checkpoint size

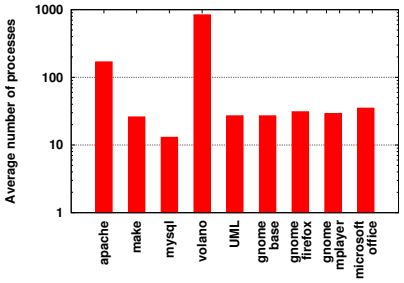


Figure 4: Average no. of processes

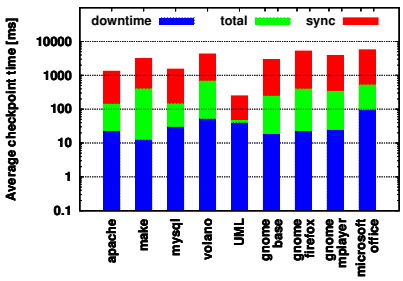


Figure 5: Average checkpoint time

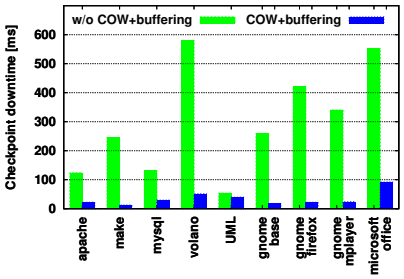


Figure 6: COW and buffering impact

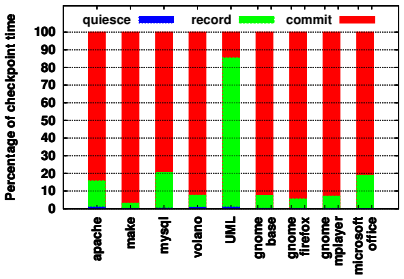


Figure 7: Checkpoint time breakdown

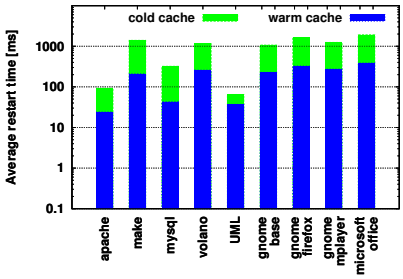


Figure 8: Average restart time

with Figure 3, the results show that both the total checkpoint times and the sync times are strongly correlated with the checkpoint sizes. Writing the filesystem, particularly with forced flushing of the data to disk, is largely limited by the disk I/O rate. For example, `gnome-base` has an average checkpoint size of 39 MB and an average sync checkpoint time of just under 3 s. This correlates directly with the sustained write rate for GFS, which was roughly 15 MB/s in our measurements.

Perhaps more importantly, checkpoint downtimes in Figure 5 show that the average time to actually perform the checkpoint without incurring storage I/O costs is small, ranging from 12 ms for a kernel make to at most 90 ms for a full fledged desktop running Microsoft Office. Though an application is unresponsive while it is quiesced and being checkpointed, even the largest average checkpoint downtimes are less 100 ms. Furthermore, the average checkpoint downtimes were less than 50 ms for all application scenarios except Microsoft Office.

Figure 6 compares the checkpoint downtime for each application scenario with and without the memory buffering and COW mechanisms that we employ. Without these optimizations, checkpoint data must be written out to disk before the pod can be resumed, resulting in checkpoint downtimes that are close to the total checkpoint times shown in Figure 5. The memory buffering and COW checkpoint optimization reduce downtime from hundreds of milliseconds to almost always under 50 ms, in some cases even as much as an order of magnitude.

Figure 7 shows the breakdown of the total checkpoint time (excluding sync) for each application scenario, as percentage of the total time attributable to different steps: *quiesce*—the time to quiesce the pod, *record*—the time to record the checkpoint data, and *commit*—the time to commit the data by writing it out to storage. The commit step amounts to 80-95% of the total time in almost all application scenarios, except for UML where it amounts to only 15% due to a much smaller checkpoint size. Quiescing the processes took less then 700 µs for all application scenarios except `apache` and `volano`, which took roughly 1.5 ms and 5 ms, respectively. The longer quiesce times are due to the large number of processes being executed in `apache` and `volano`. The time to generate and record the process forest was even smaller, less than 10 µs for all applications except `apache` and `volano`, which took 30 µs and 336 µs respectively. The time to record globally shared resources was under 10 µs in all cases.

Figure 8 presents the average total restart times for each application scenario. The restart times were measured for two distinct configurations: *warm cache*—restart was done with a warm filesystem cache immediately after the checkpoint was taken, *cold-cache*—restart was done with a cold filesystem cache after the system was rebooted, forcing the system to read the image from the disk. Warm cache restart times were less than .5 s in all cases, ranging from 24 ms for `apache` to 386 ms for a complete Gnome desktop running Microsoft Office. Cold cache restart times were longer as restart becomes limited by the disk I/O rate. Cold cache restart times were less than 2 s in all cases, ranging from 65 ms for UML to 1.9 s for Microsoft Office. The cold restart from a checkpoint image is still noticeably faster than the checkpoint to the filesystem with flushing

because GFS filesystem read performance is much faster than its write performance.

To provide a comparison with another operating system virtualization approach, we also performed our experiments with OpenVZ. We used version 2.6.18.028stab on the same Linux installation. Because of its lack of GFS support, we copied the installation to the local disk to conduct experiments. Since this configuration is different from what we used with our prototype, the measurements are not directly comparable. However, they provide some useful comparisons between the two approaches. We report OpenVZ results for `apache`, `make`, `mysql` and `volano`; OpenVZ was unable to checkpoint the other scenarios. Table 3 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We ignore sync checkpoint times and cold cache restart times to reduce the impact of the different disk configurations used.

Scenario	Checkpoint [s]	Restart [s]	Size [MB]
apache	0.730	1.321	7.7
make	2.230	1.376	53
mysql	1.793	1.288	22
volano	2.036	1.300	25

Table 3: Checkpoint-restart performance for subset of applications that worked on OpenVZ

The results show that OpenVZ checkpoint and restart times are significantly worse than our system. OpenVZ checkpoint times were 5.2, 5.6, 12.4, and 3.0 times slower for `apache`, `make`, `mysql` and `volano`, respectively. OpenVZ restart times were 55.0, 6.6, 29.9, and 5.0 times slower for `apache`, `make`, `mysql` and `volano`, respectively. OpenVZ checkpoint sizes were .48, 1.3, 1.2, and .46 times the sizes of our system. The difference in checkpoint sizes was relatively small and does not account for the huge difference in checkpoint-restart times even though different filesystem configurations were used due to OpenVZ's lack of support for GFS. OpenVZ restart times did not vary much among application scenarios, suggesting that container setup time may constitute a major component of latency.

To provide a comparison with a hardware virtualization approach, we performed our experiments with Xen. We used Xen 3.0.3 with its default Linux 2.6.16.29. We were unable to find a GFS version that matched this configuration, so we used the local disk to conduct experiments. We also used Xen 2.0 with Linux 2.6.11 because this configuration worked with GFS. In both cases, we used the same kernel for both "dom0" and "domU". We used three VM configurations with 128 MB, 256 MB, and 512 MB of memory. We report results for `apache`, `make`, `mysql`, UML, and `volano`; Xen was unable to run the other scenarios due to lack of support for virtual consoles. Table 4 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We report a single number for each configuration instead of per application since Xen results were directly correlated with the VM memory configuration and did not depend on the applications scenario. Checkpoint image size was determined by the amount of RAM configured. Checkpoint and restart times were directly correlated with the size of the checkpoint images.

Xen Config.	Checkpoint [s]		Restart [s]		Image Size [MB]
	Xen 3	Xen 2	Xen 3	Xen 2	
128 MB	3.5	5.5	1.6	0.8	129
256 MB	10.3	12	13.4	6.6	257
512 MB	25.9	19	27.3	12	513

Table 4: Checkpoint-restart performance for Xen VMs

The results show that Xen checkpoint and restart times are significantly worse than our system. Xen 3 checkpoint times were 5.2 (`volano` on 128 MB) to 563 (UML on 512 MB) times slower. Xen 3 restart times were 6.2 (`volano` on 128 MB) to 1137 (`apache` on 512 MB) slower. Xen results are also worse than OpenVZ; both operating system virtualization approaches performed better. Restart times for the 256 MB and 512 MB VM configurations were much worse than the 128 MB VM because the images ended up being too large to be effectively cached in the kernel, severely degrading warm cache restart performance. Note that although precopying can reduce application downtime for Xen migration [4], it will not reduce total checkpoint-restart times.

9 Conclusions

We have designed, implemented, and evaluated a transparent checkpoint-restart mechanism for commodity operating systems that checkpoints and restarts multiple processes in a consistent manner. Our system combines a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for transparent checkpoint-restart. We have introduced novel algorithms for saving and restoring extended process relationships and for efficient handling of shared state across cooperating processes. We have implemented a checkpoint-restart prototype and evaluated its performance on real-world applications. Our system generates modest checkpoint image sizes and provides fast checkpoint and restart times without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Comparisons with two commercial systems, OpenVZ and Xen, demonstrate that our prototype provides much faster checkpoint-restart performance and more robust checkpoint-restart functionality than these other approaches.

Acknowledgments

Dan Phung helped with implementation and experimental results. Eddie Kohler, Ricardo Baratto, Shaya Potter and Alex Sherman provided helpful comments on earlier drafts of this paper. This work was supported in part by a DOE Early Career Award, NSF ITR grant CNS-0426623, and an IBM SUR Award.

References

[1] R. Baratto, L. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 277–290, Brighton, UK, Oct. 2005.

[2] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[3] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, pages 273–286, Boston, MA, May 2005.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[6] P. Gupta, H. Krishnan, C. P. Wright, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-1, Dept. of Computer Science, Stony Brook University, Jan. 2004.

[7] Y. Huang, C. Kintala, and Y. M. Wang. Software Tools and Libraries for Fault Tolerance. *IEEE Bulletin of the Tecnnical Committee on Operating System and Application Environments*, 7(4):5–9, Winter 1995.

[8] L. V. Kale and S. Krishnan. CHARM++: a Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 91–108, Washington, DC, Sept. 1993.

[9] B. A. Kingsbury and J. T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 355–364, San Diego, CA, Jan. 1989.

[10] C. R. Landau. The Checkpoint Mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, Dourdan, France, Sept. 1992.

[11] Linux Software Suspend. `http://www.suspend2.net`.

[12] Linux VServer. `http://www.linux-vserver.org`.

[13] M. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[14] Network Appliance, Inc. `http://www.netapp.com`.

[15] OpenVZ. `http://www.openvz.org`.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, Dec. 2002.

[17] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Dept. of Computer Science, University of Tennessee, July 1997.

[18] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, Jan. 1995.

[19] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[20] D. Price and A. Tucker. Solaris Zones: Operating Systems Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference*, pages 241–254, Atlanta, GA, Nov. 2004.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 235–248, Brighton, UK, Oct. 2005.

[22] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[23] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 300.2, Washington, DC, Apr. 2005.

[24] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Stanford University, Aug. 2000.

[25] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems*, pages 319–342, College Park, MD, Sept. 1996.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference, General Track*, pages 29–44, Boston, MA, June 2004.

[27] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1993.

[28] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, 20(227):40–48, Feb. 1995.

[29] VMware, Inc. `http://www.vmware.com`.