Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Making Applications Mobile Under Linux

Cédric Le Goater, Daniel Lezcano, Clément Calmels
IBM France
{clg, dlezcano, clement.calmels}@fr.ibm.com

Dave Hansen, Serge E. Hallyn
IBM Linux Technology Center
{haveblue, serue}@us.ibm.com

Hubertus Franke
IBM T.J. Watson Research Center
frankeh@watson.ibm.com

Abstract

Application mobility has been an operating system research topic for many years. Many approaches have been tried and solutions are found across the industry. However, performance remains the main issue and all the efforts are now focused on performant solutions. In this paper, we will discuss a prototype which minimizes the overhead at runtime and the amount of application state. We will examine constraints and requirements to enhance performance. Finally, we will discuss features and enhancements in the Linux kernel needed to implement migration of applications.

1 Introduction and Motivation

Applications increasingly run for longer periods of time and build more context over time as well. Recovering that context can be time consuming, depending on the application, and usually requires that the application be re-run from the beginning to reconstruct its context. A few applications now provide the ability to checkpoint their data or context to a file, enabling that application to be restarted later in the case of a failure, a system upgrade, or a need to redeploy hardware resources. This ability to checkpoint context is most common in what is referred to as the High Performance Computing (HPC) environment, which is often composed of large numbers of computers working on a distributed, long running computation. The applications often run for days or weeks at a time, some even as long as a year.

Even outside the HPC arena, there are many applications which have long start up times, long periods of processing configuration files, pre-computing information and so on. Historically, emacs was built with a script which included `undump`—the ability to checkpoint the full state of emacs into a binary which could then be started much more quickly. Some enterprise class applications have thirty minute start up times, and those applications continue to build complex context as they continue to run.

Increasingly we as users tend to expect that our applications will perform quickly, start quickly, re-start quickly on a failure, and be always available. However, we also expect to be able to upgrade our operating system, apply security fixes, add components, memory, sometimes even processing power without losing all of the context that our applications have ac-

quired.

This paper discusses a generic mechanism for saving the state of an application at any point, with the ability to later restart that application exactly where it left off. This ability to save status and restart an application is typically referred to as checkpoint/restart, abbreviated throughout as CPR. This paper focuses on the key areas for allowing applications to be virtualized, simplifying the ability to checkpoint and later restart an application. Further, the technologies covered here would allow applications to potentially be restarted on a different operating system image than the one from which it was checkpointed. This provides the ability to move an application (or even a set of applications) dynamically from one machine or virtual operating system image to another.

Once the fundamental mechanisms are in place, this technology can be used for such more advanced capabilities such as checkpointing a cluster wide application—in other words synchronizing and stopping a coordinated, distributed applications, and restarting them. Cluster wide CPR would allow a site administrator to install a security update or perform scheduled maintainance on the entire cluster without impacting the application running.

Also, CPR would enable applications to be moved from host to host depending on system load. For instance, an overloaded machine could have its workload rebalanced by moving an application set from one machine to another that is otherwise underutilized. Or, several systems which are underloaded could have their applications consolidated to a single machine. CPR plus migration will henceforth be referred to as CPRM.

Most of the capabilities we've highlighted here are best enabled via application virtualization. Application virtualization is a means of abstracting, or virtualizing, the software resources of the system. These include such things as process id's, IPC ids, network connections, memory mappings, etc. It is also a means to contain and isolate resources required by the application to enable its mobility. Compared to the virtual machine approach, application virtualization approach minimizes the state of the application to be transferred and also allows for a higher degree of resource sharing between applications. On the other hand, it has limited fault containment, when compared to the virtual machine approach.

We built a prototype, called MCR, by modifying the Linux kernel and creating such a layer of containment. They are various other projects with similar goals, for instance VServer [8] and OpenVZ [7] and dated Linux implementation of BSD Jails [4]. In this paper, we will describe our experiences from implementing MCR and examine the many communalities of these projects.

2 Related Work in CPR

CPR is theoretically simple. Stop execution of the task and store the state of all memory, registers, and other resources. To restart, reload the executable image, load the state saved during the checkpoint, and restart execution at the location indicated by the instruction pointer register. In practice, complications arise due to issues like inter-processes sharing, security implications, and the ways that the kernel transparently manages resources. This section groups some of the existing solutions and reviews their shortcomings.

Virtual machines control the entire system state, making CPR easy to implement. The state of all memory and resources can simply be

stored into a file, and recreated by the machine emulator or the operating system itself. Indeed, the two most commonly mentioned VMs, VMware [9] and Xen [10], both enable live migration of their guest operating systems. The drawbacks of CPRM of an entire virtual machine is the increased overhead of dealing with all resources defining the VM. This can make the approach unsuitable for load balancing applications, since a requirement to add the overhead of a full VM and associated daemons to each migrateable application can have tremendous performance implications. This issue is further explored in Section 6.

A more lighter weight CPRM approach can be achieved by isolating applications, which is predicated on the safe and proper isolation and migration of its underlying resources. In general, we look at these isolated and migrateable units as containers around the relevant processes and resources. We distinguish conceptually *system containers*, such as VServer [8] or OpenVZ [7], and *application containers*, such as Zap [12] and our own prototype MCR. Since containers share a single OS instance, many resources provided by the OS must be specially isolated. These issues are discussed in detail in Section 5. Common to both container approaches is their requirement to be able to CPR an isolated set of individual resources.

For many applications CPR can be completely achieved from user space. An example implementation is `ckpt` [5]. Ckpt teaches applications to checkpoint themselves in response to a signal by either preloading a library, or injecting code after application startup. The new code, when triggered, writes out a new executable file. This executable reloads the application and resets its state before continuing execution where it left off. Since this method is implemented with no help from the kernel, there is state which cannot easily be stored, such as pending signals, or recreated, such as a process' original process id. This method is also potentially very inefficient as described in Section 5.2. The user space approaches also fall short by requiring applications to be rewritten and by exhibiting poor resource sharing.

CPR becomes much easier given some help from the kernel. Kernel-based CPR solutions include `zap` [12], `crak` [2], and our MCR prototype. We will be analyzing MCR in detail in Section 4, followed by the requirements for a consolidated application virtualization and migration kernel approach.

3 Concepts and principles

A user application is a set of resources—tasks, files, memory, IPC objects, etc.—that are aggregated to provide some features. The general concept behind CPR is to freeze the application and save all its state (in both kernel and user spaces) so that it can be resumed later, possibly on another host. Doing this transparently without any modification to the application code is quite an easy task, as long as you maintain a single strong requirement: *consistency*.

3.1 Consistency

The kernel ensures *consistency* for each resource's internal state and also provides system identifiers to user space to manipulate these resources. These identifiers are part of the application state, and as such are critical to the application's correct behavior. For example, a process waiting for a child will use the known process id of that child. If you were to resume a checkpointed application, you would recreate the child's pid to ensure that the parent would wait on the correct process. Unfortunately, Linux does not provide such control on

how pids, or many other system identifiers, are associated with resources.

An interesting approach to this problem is virtualization: a resource can be associated with a supplementary virtual system identifier for user space. The kernel can maintain associations between virtual and system identifiers and offer interfaces to control the way virtual identifiers are assigned. This makes it possible to change the underlying resource and its system identifier without changing the virtual identifier known by the application. A direct side effect is that such virtualized applications can be confused by virtual identifier collisions if they are not separated from one another. These conflicts can be avoided if the virtualization is implemented with resource containment features. For example, `/proc` should only export virtual pid entries for processes in the same virtualized container as the reading process.

3.2 Process subsystem

Processes are the essential resources. They offer many features and are involved in many relationships, such as parent, thread group, and process group. The pid is involved in many system calls and regular UNIX features such as session management and shell job control. Correct virtualization must address the whole picture to preserve existing semantics. For example, if we want to run multiple applications in different containers from the same login session, we will also want to keep the same system session identifier for the ancestor of the container so as to still benefit from the regular session cleanup mechanism. The consequence is that we need a system pid to be virtualized multiple times in different containers. This means that any kernel code dealing with pids that are copied to/from user space must be patched to provide containment and choose the correct virtualization space according to the implied context.

3.3 Filesystem and Devices

A filesystem may be divided into two parts. On one hand, global entries that are visible from all containers like `/usr`. On the other hand, local entries that may be specific to the containers like `/tmp` and of course `/proc`. `/proc` virtualization for numerical process entries is quite straightforward: simply generate a filename out of the virtual pid. Tagging the resultant `dentries` with a container id makes it possible to support conflicting names in the directory name cache and to filter out unrelated entries during `readdir()`.

The same tagging mechanism can be applied using files attributes for instance. Some user level administration commands can be used by the system administrator to keep track of files created by the containers. Devices files can be tagged to be visible and usable by dedicated containers. And of course, the `mknod` system call should fail on containers or be restricted to a minimal usage.

3.4 Network

If the application is network oriented, the migration is more complex because resources are seen from outside the container, such as IP addresses, communication ports, and all the underlying data related to the TCP protocol. Migration needs to take all these resources into account, including in-flight messages.

The IP address assigned to a source host should be recreated on the target host during the migration. This mobile IP is the foundation of the migration, but adds a constraint on the network. We will need to stay on the same network.

The migration of an application will be possible only if the communication channels are clearly isolated. The connections and the data associated with each application should be identified. To ensure such a containment, we need to isolate the network interfaces.

We will need to freeze the TCP layer before the checkpoint, to make sure the state of the peers is consistent with the snapshot. To do this, we will block network traffic for both incoming and outgoing packets. The application will be migrated immediately after the checkpoint and all the network resources related to the container will be cleaned up.

4 Design Overview

We have designed and implemented MCR, a lightweight application oriented container which supports mobility. It is discussed here because it is one of a few implementations which are relatively complete. It provides an excellent view on what issues and complexities arise. However, from our prototype work, we have concluded that certain functionality implemented in user space in MCR is best supported by the kernel itself.

An important idea behind the design of MCR is that it is kernel-friendly and does not do everything in kernel space. A balance needed to be struck between striving towards a minimal kernel impact to facilitate proper forward ports and ensuring that functional correctness and acceptable performance is achieved. This means using available kernel features and mechanisms where possible and not violating important principles which ensure that user space applications work properly.

With that principle in mind, CPR from user space makes your life much easier. It also enables some nifty and useful extensions like distributed CPR and system management.

4.1 Architecture

The following section provides an overview of the MCR architecture (Figure 1). It relies on a set of user level utilities which control the container: creation, checkpoint, restart, etc. New features in the kernel and a kernel module are required today to enable the container and the CPR features.

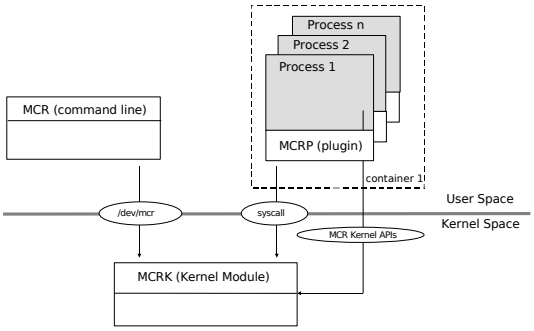

Figure 1: MCR architecture

The CPR of the container is not handled by one component. It is *distributed* across the 3 components of MCR depending on the locality of the resourse to checkpoint. The user level utility `mcr` (Section 4.1.2) invokes and orchestrates the overall checkpoint of a container. It also is in charge of checkpointing the resources which are global to a container, like SYSV shared memory for instance. The user level plugin `mcrp` (Section 4.1.4) checkpoints the resources at the process level, memory, and at the thread level, signals and cpu state. Both rely on the kernel module `mcrk` (Section 4.1.3) to access kernel internals.

4.1.1 Kernel enhancements and API

Despite a user space-oriented approach, the kernel still requires modifications in order to support CPR. But, surprisingly, it may no be as much as one might expect. There are three high level needs:

The biggest challenge is to build a container for the application. The aim here is neither security nor resource containment, but making sure that the snapshot taken at checkpoint time is *consistent*. To that end we need a container much like the VServer[8] context. This will isolate and identify all kernel resources and objects used by an application. This kernel feature is not only a key requirement to application mobility, but also for other frameworks in the security and resource management domains. The container will also virtualize system identifiers to make sure that the resources used by the application do not overlap with other containers. This includes resources such as processes IDs, threads IDs, SysV IPC IDs, UTS names, and IP addresses, among others.

The second need regards freezing a container. Today, we use the `SIGSTOP` signal to freeze all running tasks. This gives valid results, but for upstream kernel development we would prefer to use a container version of the `refrigerator()` service from `swsusp`. `swsusp` uses fake signals to freeze nearly all kernel tasks before dumping the memory.

Finally, MCR exposes the internals of different Linux subsystems and provides new services to *get* and *set* their state. These interfaces are by necessity very intrusive, and expose internal state. For an upstream migration solution, using a `/proc` or `/sysfs` interface which exposes more selective data would be more appropriate.

4.1.2 User level utilities

Applications are made mobile simply by being started under `mcr-execute`. The container is created before the `exec()` of the command starting the application. It is maintained around the application until the last process dies.

`mcr-checkpoint` and `mcr-restart` invoke and orchestrate the overall checkpoint or restart of a container. These commands also perform the *get* and *set* of the resources which are global to a container, as opposed to those local to a single process within the container. For instance, this is where the SYSV IPC and file descriptors are handled. They rely on the kernel module (Section 4.1.3) to manage the container and access kernel internals.

4.1.3 Kernel module

The kernel module is the container manager in terms of resource usage and resource virtualization. It maintains a real time definition of the container view around the application which ensures that a checkpoint will be consistent at any time.

It is in charge of the global synchronization of the container during the CPR sequence. It freezes all tasks running in the container, and maps into the process a user level plugin (see Section 4.1.4). It provides the synchronization barriers which unrolls the full sequence of the checkpoint before letting each process resume its execution.

It also acts as a *proxy* to capture the states which cannot be captured directly from user space. Internal states handled by this module include, for example, the process' memory page mapping, socket buffers, clone flags, and AIO states.

4.1.4 User level plugin

When a checkpoint or a restart of a container is invoked, the kernel module maps a plugin into each process of the container. This plugin is run in the process context and is removed after completion of the checkpoint. It serves 2 purposes. The first is synchronization, which it orchestrates with the help of the kernel module. Secondly, it performs *get* and *set* of states which can be handled from user space using standard syscalls. Such states include sigactions, memory mapping, and rlimits.

When all threads of a process enter the plugin, a *master* thread, not necessarily the main thread, is elected to handle the checkpoint of the resources at process level. The other threads only checkpoint the resources at thread level, like cpu state.

4.2 Linux subsystems CPR

The following subsections describe the checkpoint and the restart of the essential resources without doing a deep dive in all the issues which need to be addressed. The following section 5 will delve deeper into selected issues for the interested reader.

4.2.1 CPU state

Checkpointing the cpu state is indirectly done by the kernel because the checkpoint is signal oriented: it is saved by the kernel on the top of the stack before the signal handler is called. This stack is then saved with the rest of the memory. At restart, the kernel will restore the cpu state in `sigreturn()` when it jumps out of the signal handler.

4.2.2 Memory

The memory mapping is checkpointed from the process context, parsing `/proc/self/maps`. However, some `vm_area` flags (i.e. `MAP_GROWSDOWN`) are not exposed through the `/proc` file system. The latter are read from the kernel using the kernel module. The same method is used to retrieve the list of the mapped pages for each `vm_area` and reduce significantly the size of the snapshot.

Special pages related to POSIX shared memory and POSIX semaphores are detected and skipped. They are handled by the checkpoint of resources global to a container.

4.2.3 Signals

Signal handlers are registered using `sigaction()` and called when the process gets a signal. They are checkpointed and restarted using the same service in the process context. Signals can be sent to the process or directly to an invidual thread using the `tkill()` syscall. In the former, the signal goes into a shared sigpending queue, where any thread can be selected to handle it. In the latter, the signal goes to a thread private sigpending queue. To guarantee correct signal ordering, these queues must be checkpointed and restored separately using a dedicated kernel service.

4.2.4 Process hierarchy

The relationships between processes must be preserved across CPR sequences. Groups and sessions leaders are detected and taken into account. At checkpoint, each process and thread stores in the snapshot its execution command using `/proc/self/exe` and its pid and ppid

using `getpid()` and `getppid()`. Threads also need to save their tid, ptid, and their stack frame. At restart time, the processes are recreated by `execve()` and immediately killed with the checkpoint signal. Each process then jumps into the user level plugin (See Section 4.1.4) and spawns its children. Each process also respawns its threads using `clone()`. The process tree is recreated recursively. On restart, attention must be paid to correctly setting the pid, pgid, tid, tgid for each newly created process and thread.

4.2.5 Interprocess communication

The contents and attributes of SYSV IPCs, and more recently POSIX IPCs, are checkpointed as resources global to a container, excepting semundos.

Most of the IPC resource checkpoint is done at the user level using standard system calls. For example, `mq_receive()` to drain all messages from a queue, and `mq_send()` to put them back into the queue. The two main drawbacks to such an approach are that access time to resources are altered and that the process must have read and write access to them. Some functionalities like `mq_notify()` are a bit trickier. In these cases, the kernel sends notification cookies using an `AF_NETLINK` socket which also needs to be checkpointed.

4.2.6 Threads

Every thread in a process shares the same memory, but has its own register set. The threads can dump themselves in user context by asking the kernel for their properties. At restart time, the main thread can read other threads' data back from the snapshot and respawn each with its original tid.

The thread local storage contains the thread specific information, data set by `pthread_setspecific()` and the `pthread_self()` pointer. On some architectures it is stored in a general purpose register. In that case it is already covered in the signal handler frame. But on some other architectures, like Intel, it is stored in a separate segment, and this segment mapping must be saved using a dedicated call to kernel.

4.2.7 Open files

File descriptors reference open files, which can be of any type, including regular files, pipes, sockets, FIFOs, and POSIX message queues.

CPR of file descriptors is not done entirely in the process context because they can be shared. Processes get their fd list by walking `/proc/self/fd`. They send this list, using ancillary messages, to a helper daemon running in the container during the checkpoint. Using the address of the `struct file` as a unique identifier, the daemon checkpoints the file descriptor only once per container, since two file descriptors pointing to the same opened file will have the same `struct file`.

File descriptors 0, 1, and 2 are considered special. We may not want to checkpoint or restart them if we do the restart on another login console for example. The file descriptors are tagged and bypassed at checkpoint time.

4.2.8 Asynchronous I/O

Asynchronous I/Os are difficult to handle by nature because they can not easily be frozen. The solution we found is to let the process reach a quiescence point where all AIOs have completed before checkpointing the memory. The

other issue to cover is the ring buffer of completed events which is mapped in user space and filled by the kernel. This memory area needs to be mapped at the same address when the process is restarted. This requires a small patch to control the address used for the mapping.

5 Zooming in

This section will examine in more detail three major aspects of application mobility. The first topic covers a key requirement in process migration: the ability to restart a process keeping the same pid. Next we will discuss issues and solutions to VM migration, which has the biggest impact on performance. Finally, we will address network isolation and migration of live network communications.

5.1 Process Virtualization

A `pid` is a handle to a particular task or task group. Inside the kernel, a `pid` is dynamically assigned to a task at fork time. The relationship is recorded in the pid hash table ($pid \rightarrow task$) and remains in place until a task exits. For system calls that return a pid (e.g. `getpid()`), the pid is typically extracted straight out of the task structure. For system calls that utilize a user provided pid, the task associated with that pid is determined from the pid hash table. In addition various checks need to be performed that guarantee the isolation between users and system tasks (in particular during the `sys_kill()` call).

Because `pids` might be cached at the user level, processes should be restarted with their original pids. However, it is difficult if not impossible to ensure that the same `pid` will always be available upon restart of a checkpointed application, as another process could already have been started with this pid. Hence, `pids` need to be *virtualized*. Virtualization in this context can be and is interpreted in various manners. Ultimately the requirement, that an application consistently sees the same `pid` associated with a task (process/thread) across CPR, must be satisfied.

There are essentially three issues that need to be dealt with in any solution:

1. container init process visibility,
2. where in the kernel the virtualization interception will take place,
3. how the virtualization is maintained.

Particularly 1. is responsible for the non-trivial complexities of the various prototypes. It stems from the necessity to "rewrite" the pid relationships between the top process of a container (short `cinit`) and its parent. `cinit` essentially lives in both contexts, the creating container and the created container. The creating container requires a pid in its context for `cinit` to be able to deploy regular `wait()` semantics. At the same time, `cinit` must refer to its parent as the perceived system init process ($vpid = 1$).

5.1.1 Isolation

Various solutions have been proposed and implemented. Zap [12] intercepts and wraps all pid related system calls and virtualizes the pids in the interception layer through a $pid \leftrightarrow vpid$ lookup table associated with the caller's container either before and/or after calling the original syscall implementation with the real pids. The benefit of this approach is that the kernel

does not need to be modified. However, the ability of overwriting the syscall table is not a direction Linux embraces.

MCR, presented in greater detail in Section 4, pushes the interception further down into the various syscalls itself, but also utilizes a *pid* ↔ *vpid* lookup function. In general, the calling task provides the context for the lookup.

The isolation between containers is implemented in the lookup function. Tasks that are created inside a container are looked up through this function. For global tasks the `pid == vpid` holds. In both implementations the `vpid` is not explicitly stored with the task, but is determined through the *pid* ↔vpid lookup each and every time. On restart tasks can be recreated through the `fork(); exec()` sequence and only the lookup table needs to record the different `pid`. The `cinit` parent problem mentioned earlier is solved by mapping `cinit` twice, in the created context as vpid=1 and in the creating container contexts with the assigned vpid. The lookup function is straight forward, essentially we need to ensure that we identify any `cinit` process and return the vpid/task associated with it relative to the provided container context.

The OpenVZ implementation [7] provides an interesting, yet worthwhile optimization that only requires a lookup for tasks that have been restarted. OpenVZ relies on the fact that tasks do have a unique pid when tasks are in their original incarnation (not yet C/R'd). The lookup function, which is called at the same code locations as the MCR implementation, hence only has to maintain the isolation property. In the case of a restarted task the uniqueness can no further be guaranteed, so the pid must be virtualized. Common to all three approaches is the fact that virtual pids are all relative to their respective containers and that they are translated into system-wide unique pids. The guts of the pidhash have not changed.

A different approach is taken by the namespace proposal [6]. Here, the container principle is driven further down into the kernel. The pidhash now is defined as a $(\{pid, container\} \rightarrow task)$ function. The namespace approach naturally elevates the container as a first class kernel object. Hence minor changes were required to the pid allocation which maintains a pidmap for each and every namespace now. The benefit of this approach is that the code modifications clearly highlight the conditions where container boundaries need to be crossed, where in the earlier virtualization approach these crossings came implicitly through the results of the lookup function. On the other hand, the namespace approach needs special provisioning for the `cinit` problem. To maintain the ability to `wait()` on the cinit process from the cinit's parent (child_reaper), a `task->wid` is defined, that reflects the pid in the parent's context and on which the parent needs to wait. There is no clear recommendation between the namespace and virtualization approach that we want to give in this paper; both the OpenVZ and the namespace proposal are very promising.

5.1.2 CPR on Process Virtualization

The CPR of the Process Virtualization is straight forward in both cases. In the ZAP, MCR and OpenVZ case, the lookup table is recreated upon restart and populated with the vpid and the real pid translations, thus requiring the ability to select a specific vpid for a restarted process. Which real pid is chosen is irrelevant and is hence left to the pidmap management of the kernel. In the namespace approach since the pid selection is pushed into the kernel a function requires that a task can be forked at a specific pid with in a container's pidmap. Ultimately, both approaches are very similar to each other.

5.2 CPR on the Linux VM

At first glance, the mechanism for checkpointing a process's memory state is an easy task. The mechanism described in section 2 can be implemented with a simple ptrace.

This approach is completely in user space, so why is it not used in the MCR prototype, nor any commercial CPR systems? Or in other words, why do we need to push certain functionalities further down into the kernel?

5.2.1 Anonymous Memory

One of the simplest kind of memory to checkpoint is anonymous memory. It is never used outside the process in which it is allocated.

However, even this kind of memory would have serious issues with a ptrace approach.

When memory is mapped, the kernel does not fill it in at that time, but waits until it is used to populate it. Any user space program doing a checkpoint could potentially have to iterate over multiple gigabytes of sparse, entirely empty memory areas. While such an approach could consolidate such empty memory after the fact, simply iterating over it could be an incredibly significant resource drain.

The kernel has intimate knowledge of which memory areas actually contain memory, and can avoid such resource drains.

5.2.2 Shared Memory

The key to successful CPR is getting a consistent snapshot. If two interconnected processes are checkpointed at different times, they may become confused when restarted. Successful memory checkpointing requires a consistent quiescence of all tasks sharing data. This includes all shared memory areas and files.

5.2.3 Copy on Write

When a process forks, both the forker and the new child have exactly the same view of memory. The kernel gives both processes a read-only view into the same memory. Although not explicit, these memory areas are shared as long as neither process writes to the area.

The above proposed ptrace mechanism would be a very poor choice for any processes which have these copy-on-write areas. The areas have no practical bounds on their sizes, and are indistinguishable from normal, writable areas from the user's (and thus ptrace's) perspective.

Any mechanism utilizing the ptrace mechanism could potentially be forced to write out many, many copies of redundant data. This could be avoided with checksums, but it causes user space reconstruction of information about which the kernel already explicitly knows.

In addition, user space has no way of explicitly recreating these copy-on-write shared areas during a resume operation. The only mechanism is fork, which is an awfully blunt instrument by which to recreate an entire system full of processes sharing memory in this manner. The only alternative is restoring all processes and breaking any sharing that was occurring before the checkpoint. Breaking down any sharing is highly undesirable because it has the potential to greatly increase memory utilization.

5.2.4 Anonymous Shared

Anonymous shared memory is that which is shared, but has no backing in a file. In Linux there is no true anonymous shared memory.

The memory area is simply backed by a pseudo file on a ram-based filesystem. So, there is no disk backing, but there certainly is a file backing.

It can only be created by an `mmap()` call which uses the `MAP_SHARED` and `MAP_ANONYMOUS`. Such a mapping is unique to a single process and not truly shared. That is, until a `fork()`.

No running processes may attach to such memory because there is no handle by which to find or address it, neither does it have persistence. The pseudo-file is actually deleted, which creates a unique problem for the CPR system.

Since the "anonymous" file is mapped by some process, the entire addressable contents of the file can be recovered through the aforementioned ptrace mechanism. Upon resume, the "anonymous" areas can be written to a real file in the same ram-based filesystem. After all processes sharing the areas have recreated their references to the "anonymous" area, the file can be deleted, preserving the anonymous semantics. As long as the process performing the checkpoint has ptrace-like capabilities for all processes sharing the memory area, this should not be difficult to implement.

5.2.5 File-backed Shared

Shared memory backed by files is perhaps the simplest memory to checkpoint. As long as all dirty data has been written back, requiring filesystem consistency be kept between a checkpoint and restart is all that is required. This can be done completely from user space.

One issue is with deleted files. However, these can be treated in the same way as "anonymous" shared memory mentioned above.

5.2.6 File-backed Private

When an application wants a copy of a file to be mapped into memory, but does not want any changes reflected back on the disk, it will map the file `MAP_PRIVATE`.

These areas have the same issues as anonymous memory. Just like anonymous memory, separately checkpointing a page is only necessary after a write. When simply read, these areas exactly mirror contents on the disk and do not need to be treated differently from normal file-backed shared memory.

However, once a write occurs, these areas' treatment resembles that of anonymous memory. The contents of each area must be read and preserved. As with anonymous memory, user space has no detailed knowledge of specific pages having been written. It must simply assume that the entire area has changed, and must be checkpointed.

This assumption can, of course, be overridden by actually comparing the contents of memory with the contents of the disk, choosing not to explicitly write out any data which has not actually changed.

5.2.7 Using the Kernel

From the ptrace discussions above, it should be apparent that the he various kinds of memory mappings in Linux can be checkpointed from userspace while preserve many of their important pre-checkpoint attributes. However, it should now be apparent that user space lacks the detailed knowledge to do these operations efficiently.

The kernel has exact knowledge of exactly which pages have been allocated and populated. Our MCR prototype uses this information to efficiently create memory snapshots.

It walks the pagetables of each memory area, and marks for checkpoint only those pages which actually have contents, and have been touched by the process being checkpointed. For instance, it records the fact that "page 14" in a memory area has contents. This solves the issues with sparsely populated anonymous and private file-backed memory areas, because it accurately records the process's actual use of the memory.

However, it misses two key points: the simple presence of a page's mapping in the page tables does not indicate whether its contents exactly mirror those on the disk.

This is an issue for efficiently checkpointing the file-backed private areas because the page may be mapped, but it may be either a page which has been only read, or one to which a write has occurred. To properly distinguish between the two, the `PageMappedToDisk()` flag must be checked.

5.2.8 File-backed Remapped

Assume that a file containing two pages worth of data is `mmap()`ed. It is mapped from the beginning of the file through the end. One would assume that the first page of that mapping would contain the first page of data from the disk. By default, this is the behavior. But, Linux contains a feature which invalidates this assumption: `remap_file_pages()`.

That system call allows a user to remap a memory area's contents such that the n^{th} page of a mapping does not correspond to the n^{th} page on the disk. The only place in which the information about the mapping is stored is in the pagetables. In addition, the presence of one of these areas is not openly available to user space.

Our user space ptrace mechanism could likely detect these situations by double-checking that each page in a file-backed memory area is truly backed by the contents on the disk, but that would be an enormous undertaking. In addition, it would not be a complete solution because two different pages in the file could contain the same data. Userspace would have absolutely no way to uniquely identify the position of a page in a file, simply given that page's contents.

This means that the MCR implementation is incomplete, at least in regards to any memory area to which `remap_file_pages()` has been applied.

5.2.9 Implementation Proposal

Any effective and efficient checkpoint mechanism must implement, at the least:

1. Detection and preservation of sharing of file-backed and other shared memory areas for both efficiency and correctness.
2. Efficient handling of sparse files and untouched anonymous areas.
3. Lower-level visibility than simply the file and contents for `remap_file_pages()` compatibility (such as effective page table contents).

There is one mechanism in the kernel today which deals with all these things: the swap code. It does not attempt to swap out areas which are file backed, or sparse areas which have not been populated. It also correctly handles the nonlinear memory areas from `remap_file_pages()`.

We propose that the checkpointing of a process's memory could largely be done with a synthetic swap file used only by that container.

This swap file, along with the contents of the pagetables of the checkpointed processes, could completely reconstruct the contents of a process' memory. The process of checkpointing a container could become very similar to the operation which `swsusp` performs on an entire system.

The swap code also has a feature which makes it very attractive to CPR: the swap cache. The swap cache allows a page to be both mapped into memory **and** currently written out to swap space. The caveat is that, if there is a write to the page, the on-disk copy must be thrown away.

Memory which is very rarely written to, such as the file-backed private memory used in the jump table in dynamically linked libraries, has the most to gain from the swap cache. Users of this memory can run unimpeded, even during a checkpoint operation, as long as they do not perform writes.

Just as has been done with other live cross-system migration[11] systems, the process of moving the data across can be iterative. First, copy data in several passes until, despite the efforts to swap them out, the working set size of the applications ceases to decrease.

The application-level approach has the potential to be at least marginally faster than the whole-system migration because it is **only** concerned with application data. Xen must deal with the kernel's working set in addition to the application. This **must** increase the amount of data which must be migrated, and thus **must** increase the potential downtime during a migration.

5.3 Migrating Sockets

In Section 3.4, the needs for a network migration were roughly defined. This section focuses on the four essential networking components required for container migration: network isolation, network quiescent points, network state access for a CPR, and network resource clean-up.

5.3.1 Network isolation

The network interface isolation consists of selectively revealing network interfaces to containers. These can be either physical or aliased interfaces. Aliased interfaces are more flexible for managing the network in the containers because different IP addresses can be assigned to different containers with the same physical network interface. The `net_device` and `in_ifaddr` structures have been modified to store a list of the containers which may view the interface.

The isolation ensures that each container uses its own IP address. But any return packet must also go to the right interface. If a container connects to a peer without specifying the source address, the system is free to assign a source address owned by an another container. This must be avoided. The `tcp_v4_connect()` `udp_sendmsg()` functions are modified in order to choose a source address associated with an interface visible from the source container.

The network isolation ensures that network traffic is dispatched to the right container. Therefore it becomes quite easy to drop the traffic for any specific container.

5.3.2 Reaching a quiescent point

As the processes need to reach a quiescent point in order to stop their activities, the network must reach this same point in order to retrieve network resource states at a fixed moment.

This point is reached by blocking the network traffic for a specified container. The filtering mechanism relies on netfilter hooks. Every packet is contained in a `struct skbuff`. This structure has a link to the `struct sock` connection which has a record of the owner container. Using this mechanism, packets related to a container being checkpointed can be identified and dropped.

For dropping packets, iptables is not directly suitable because each drop rule returns a `NF_DROP`, which interacts with the TCP stack. But, we need the TCP stack to be frozen for our container. So a kernel module has been implemented which drops the packets but returns `NF_STOLEN` instead of `NF_DROP`.

This of course relies on the TCP protocol's retransmission of the lost packets. However, traffic blocking has a drawback: if the time needed for the migration is too large, the connections on the peers will be broken. The same will occur if the TCP keep alive time is too small. This encourages any implementation to have a very short downtime during a migration. However, note that, when both sides of a connection are checkpointed simultaneously, there are no problems with TCP timeouts. In that case the restart could occurs years later.

5.3.3 Socket CPR

Now that we have successfully blocked the traffic and frozen the TCP state, the CPR can actually be performed.

Retrieving information on UDP sockets is straightforward. The protocol control block is simple and the queues can be dropped because UDP communication is not reliable. Retrieving a TCP socket is more complex. The TCP sockets are classified into two groups: `SS_UNCONNECTED` and `SS_CONNECTED`. The former have little information to retrieve because the PCB (Protocol Control Block) is not used and the send/receive queues are empty. The latter have more information to be checkpointed, such as information related to the socket, the PCB, and the send/receive queues. Minisocks and the orphan sockets also fall in the connected category.

A socket can be retrieved from `/proc` because the file descriptors related to the current process are listed. The `getpeername()`, `getsockname()` and `getsockopt()` can be directly used with the file descriptors. However, some information is not accessible from user space, particularly the list of the minisocks and the orphaned sockets, because no fd is associated with them. The PCB is also unaccessible because it is an internal kernel structure. MCR adds several accessors to the kernel internals to retrieve this missing information.

The PCB is not completely checkpointed and restored because there is a set of fields which need to be modified by the kernel itself. For example, the round time trip values.

5.3.4 Socket Cleanup

When the container is migrated, the local network resources remaining in the source host should be cleaned up in order to avoid duplicate resources on the network. This is done using the container identifier. The IP addresses can not be used to find connections related to a container because if several interconnected containers are running on the same machine, there is no way to find the connection owner.

5.3.5 CPR Dynamics

The fundamentals for the migration have been described in the previous sections. Figure 2

illustrates how they are used to move network resources from one machine to an another.

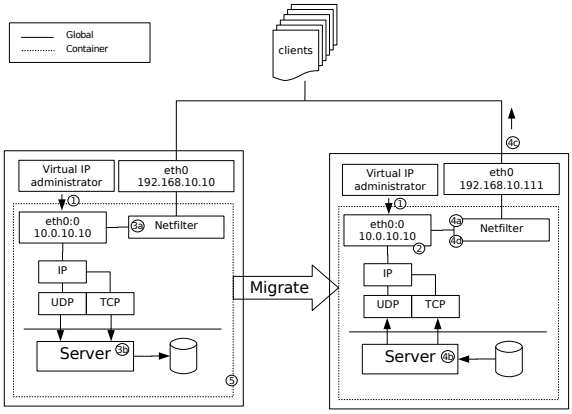

Figure 2: Migrating network resources

1. Creation

   A network administration component is launched; it creates an aliased interface and assigns it to a container.

2. Running

   The IP address associated with the aliased interface is the only one seen by the applications inside the container.

3. Checkpoint

   (a) All the traffic related to the aliased interface assigned to the container is blocked.

   (b) The network resources are retrieved for each kind of socket and saved: addresses, ports, multicast groups, socket options, PCB, in-flight data and listening points.

4. Restart

   (a) The traffic is blocked

   (b) The network resources are set from the file to the system.

   (c) An ARP (adress request package) response is sent to the network in order to boost up and ensure correct $mac \leftrightarrow ip$ address association.

   (d) The traffic is unblocked.

5. Destruction

   The traffic is blocked, the aliased interface is destroyed and the sockets related to the container are removed from the system.

6 What is the cost?

This section presents an overview of the cost of virtualization in different frameworks. We have focused on VServer, OpenVZ, and our own prototype MCR, which are all lightweight containers. We have also included Xen, when possible, as a point of reference in the field of full machine virtualization.

The first set of tests assesses the virtualization overhead on a single container for each above mentioned solution. The second measures scalability of each solution by measuring the impact of idle containers on one active container. The last set provides performance measures of the MCR CPR functionality with a real world application.

6.1 Virtualization overhead

At the time of this writing, no single kernel version was supported by each of VServer, OpenVZ, and MCR. Furthermore, patches against newer kernel versions come out faster than we can collect results, and clearly by the time of publication the patches and base kernel used in testing will be outdated anyway. Hence for each virtualization implementation we present results normalized against results obtained from the same version vanilla kernel.

6.1.1 Virtualization overhead inside a container

The following tests were made on quad PIII 700MHz running Debian Sarge using the following versions:

- VServer version vs2.0.2rc9 on a 2.6.15.4 kernel with `util-vserver` version 0.30.210
- MCR version 2.5.1 on a 2.6.15 kernel

We used *dbench* to measure filesystem load, *LMbench* for microbenchmarks, and a kernel build test for a generic macro benchmark. Each test was executed inside a container, with only one container created.

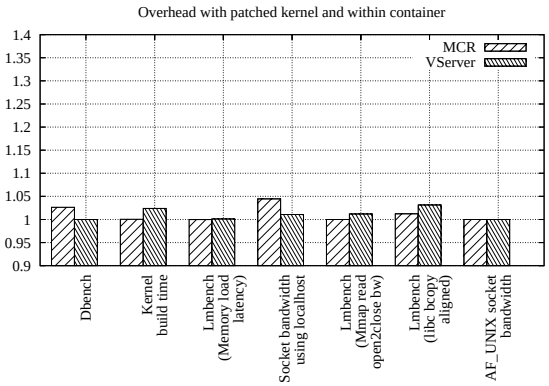

Figure 3: Various tests inside a container

The results shown in figure 3 demonstrate that the overhead is hardly measurable. OpenVZ, being a full virtualized server, was not taken into account.

6.1.2 Virtualization overhead within a virtual server

The next set of tests were run in a full virtual server rather than a simple container. For these tests, the nodes used were 64bit dual Xeon 2.8GHz (4 threads/2 real processors). Nodes were equipped with a 25P3495a IBM disk (SATA disk drive) and a Tigon3 gigabit ethernet adapter. The host nodes were running RHEL AS 4 update 1 and all guest servers were running Debian Sarge. We ran tbench and a 2.6.15.6 kernel build test in three environments: on the system running the vanilla kernel, on the host system running the patched kernel, and inside a virtual server (or guest system). The kernel build test was done with warmed up cache.

- VServer version 2.1.0 on a 2.6.14.4 kernel with `util-vserver` version 0.30.209
- OpenVZ version 022stab064 on a 2.6.8 kernel with `vzctl` utilities version 2.7.0-26
- Xen version 3.0.1 on a 2.6.12.6 kernel

The results are shown in the figures 4 and 5.

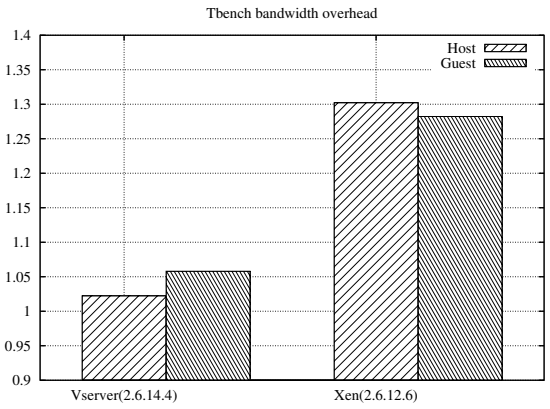

Figure 4: tbench results regarding a vanilla kernel

The OpenVZ virtual server did not survive all the tests. tbench looped forever and the kernel build test failed with a virtual memory allocation error. As expected, lightweight containers outperform a virtual machine. Considering the

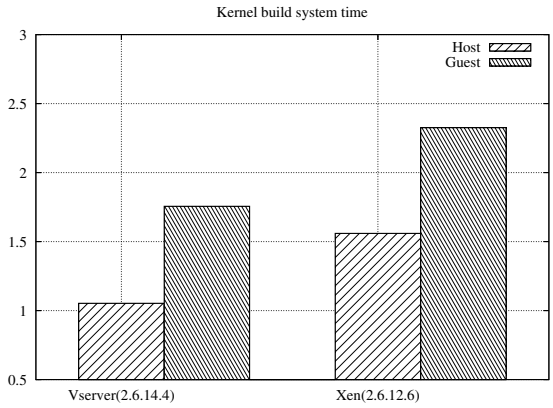

Figure 5: System time of a kernel build regarding a vanilla kernel

level of containment provided by Xen and the configuration of the domain, using file-backed virtual block device, Xen also behaved quite well. It would surely have better results with a LVM-backed VBD.

6.2 Resource requirement

Using the same tests, we have studied how performance is impacted when the number of containers increases. To do so, we have continously added idle containers to the system and recorded the application performance of the reference test in the presence of an increasing number of idle containers. This gives some insight in the resource consumption of the various virtualization techniques and its impact on application performance. This set of tests compared:

- VServer version 2.1.0 on a 2.6.14.4 kernel with `util-vserver` version 0.30.209
- OpenVZ version 022stab064 on a 2.6.8 kernel with `vzctl` utilities version 2.7.0-26
- Xen version 3.0.1 on a 2.6.12.6 kernel
- MCR version 2.5.1 on a 2.6.15 kernel

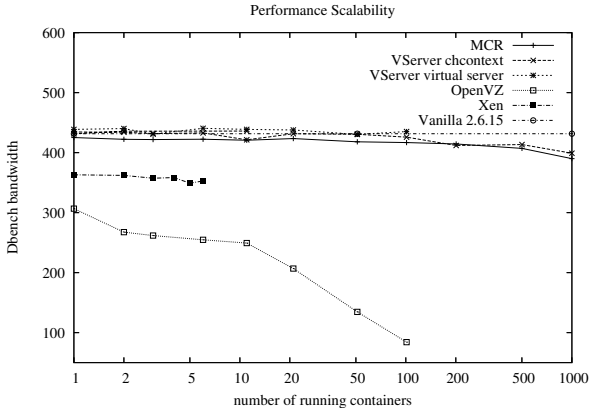

Figure 6: dbench performance with an increasing number of containers

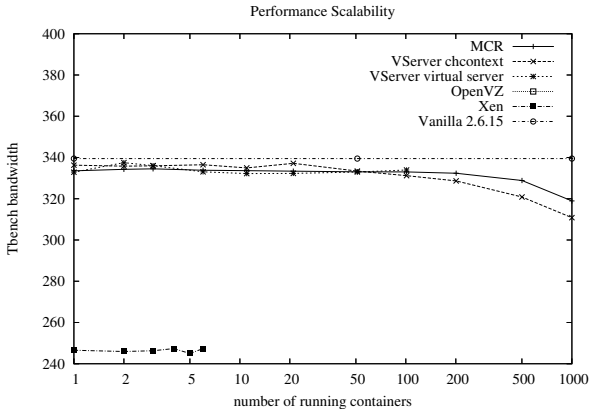

Figure 7: tbench performance with an increasing number of containers

The results are shown in the figures 6, 7, and 8. Lightweight containers are not really impacted by the number of idle containers. Xen overhead is still very reasonable but the number of simultaneous domains we were able to run stably was quite low. This issue is a bug in current Xen, and is expected to be solved. OpenVZ performance was poor again and did not survive the tbench test not the kernel build. The tests should definitely be rerun with a newer version.

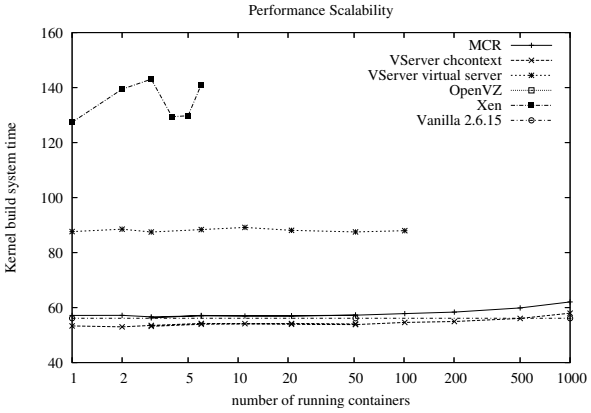

Figure 8: Kernel build time with an increasing number of containers

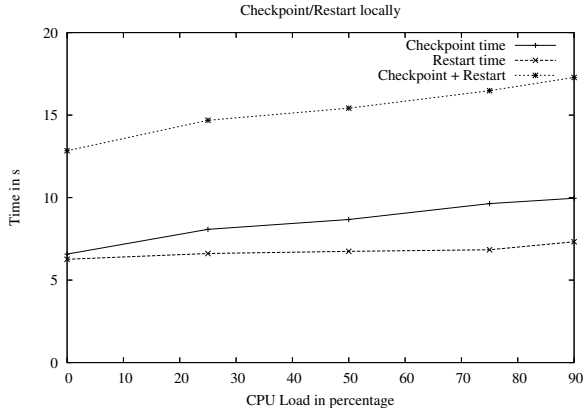

Figure 9: Oracle CPR time under load on local disk

6.3 Migration performance

To illustrate the cost of a migration, we have set up a simple test case with Oracle (`http://www.oracle.com/index.html`) and Dots, a database benchmark coming from the Linux Test Project (`http://ltp.sourceforge.net/`). The nodes used in our test were dual Xeon 2.4GHz HT (4 cpus/2 real processors). Nodes were equipped with a ST380011A (ATA disk drive) and a Tigon3 gigabit ethernet adapter. Theses nodes were running a RHEL AS 4 update 1 with a patched 2.6.9-11.EL kernel. We used Oracle version 9.2.0.1.0 running under MCR 2.5.1 on one node and Dots version 1.1.1 running on another node (nodes are linked by a gigabit switch). We measured the duration of checkpoint, the duration of restart and the size of the resulting snapshot with different Dots cpu workloads: no load, 25%, 50%, 75%, and 100% cpu load. The results are shown in Figures 9 and 10.

The duration of the checkpoint is not impacted by the load but is directly correlated to the size of the snapshot (real memory size). Using a swap file dedicated to a container and incremental checkpoints in that swap file (Section 5.2) should improve dramatically the checkpoint time. It will also improve service downtime when the application is migated from a node to another.

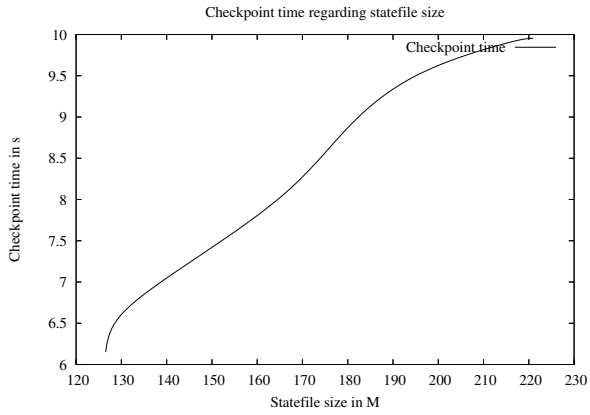

Figure 10: Time of checkpoint according to the snapshot size

At the time we wrote the paper, we were not able to run the same test with Xen, their migration framework not yet being available.

7 Conclusion

We have presented in this paper a motivation for application mobility as an alternative to

the heavier virtual machine approach. We discussed our prototype of application mobility using the simple CPR approach. This prototype helped us to identify issues and work on solutions that would bring useful features to the Linux kernel. These features are isolation of resources through containers, virtualization of resources and CPR of the kernel subsystem to provide mobility. We also went through the various other alternatives projects in that are currently persued within community and exemplified the many communalities and currents in this domain. We believe the time has come to consolidate these efforts and drive the necessary requirements into the kernel. These are the necessary steps that will lead us to live migration of applications as a native kernel feature on top of containers.

8 Acknowledgments

First all, we would like to thank all the former Meiosys team who developed the initial MCR prototype, Frédéric Barrat, Laurent Dufour, Gregory Kurz, Laurent Meyer, and François Richard. Without their work and expertise, there wouldn't be much to talk about. A very special thanks to Byoung-jip Kim from the Department of Computer Science at KAIST and Jong Hyuk Choi from the IBM T.J. Watson Research Center for their contribution to the s390 port, tests, benchmarks and this paper. And, most of all we need to thank Gerrit Huizenga for his patience and his encouragements. *Merci, DonkeySchön*!

9 Download

Patches, documentations and benchmark results will be available at `http://lxc.sf.net`.

References

[1] William R. Dieter and James E. Lumpp, Jr. *User-level Checkpointing for LinuxThreads Programs,* Department of Electrical and Computer Engineering, University of Kentucky

[2] Hua Zhong and Jason Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module,* Department of Computer Science, Columbia University, Technical Report CUCS-014-01, November, 2001.

[3] Duell, J., Hargrove, P., and Roman., E. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart,* Berkeley Lab Technical Report (publication LBNL-54941).

[4] Poul-Henning Kamp and Robert N. M. Watson, *R. N. M. Jails: Confining the omnipotent root,* in Proc. 2nd Intl. SANE Conference (May, 2000).

[5] Victor Zandy, *Ckpt—A process checkpoint library,* `http://www.cs.wisc.edu/~zandy/ckpt/`.

[6] Eric Biederman, *Code to implement multiple instances of various linux namespaces,* `git://git.kernel.org/pub/scm/linux/kernel/git/ebiederm/linux-2.6-ns.git/`.

[7] SWSoft, *OpenVZ: Server Virtualization Open Source Project*, `http://openvz.org`, 2005.

[8] Jacques Gélinas, *Virtual private servers and security contexts*, `http://www.solucorp.qc.ca/miscprj/s_context.hc?prjstate=1&nodoc=0`, 2004.

[9] VMware Inc, *VMware*, `http://www.vmware.com/`, 2005.

[10] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer, *Xen and the art of virtualization,* in Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, *L*ive Migration of Virtual Machines, In Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05), May, 2005, Boston, MA.

[12] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. *The design and implementation of zap: A system for migrating computing environments,* in Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation, pp. 361–376, December, 2002.

[13] Daniel Price and Andrew Tucker. "Solaris Zones: Operating System Support for Consolidating Commercial Workloads." From *Proceedings of the 18th Large Installation Systems Administration Conference* (USENIX LISA '04).

[14] Werner Almesberger, *Tcp Connection Passing,* `http://tcpcp.sourceforge.net`

Copyright © 2006 IBM.

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provided "AS IS," with no express or implied warranties. Use the information in this document at your own risk.