Virtual Machines for Grid Computing

Erik Paulson
Department of Computer Sciences
University of Wisconsin, Madison
Madison, WI 53706
epaulson@cs.wisc.edu

ABSTRACT

The emerging computational grid infrastructure will provide users with access to orders of magnitude more computing power than they currently have available. These resources will be heterogeneous in type and implementation and independently controlled and administered. Users of grid resources will need mechanisms to account for these variations that must both be without a high performance overhead while providing resource owners strong safety assurances. We propose using virtual machines, a classic idea from the 1960's and 70's, as this mechanism.

In this work, we briefly explore some of the challenges that grid computing users will be faced with. We compare several different architectures for building virtual machines, and compare performance of several standard benchmarks under these different virtual machines. We examine some unique opportunities the additional layer of abstraction provides, such as checkpointing, split execution, and heterogenous process migration, all with unmodified user executables running on unmodified operating systems. We conclude by examining future work, and propose integrating this work with Condor, a Grid-Enabled High-Throughput Computing system developed at the University of Wisconsin, Madison.

Keywords

Checkpointing, Virtual Machines, User-Mode Linux

1. INTRODUCTION

The Grid[11] is the natural extension of the idea that while computers will become more and more pervasive in the future, it will be increasingly difficult to point to any one object and call it the "computer." Instead, it will be a collection of cheap and reliable devices embedded throughout the environment. Similarly, the large computing facilities characterized by the NSF-funded supercomputing centers of the 1980's and 90's will be replaced by a computationally-enhanced network ubiquitously available to the user. Much as the current electrical grid provides power for universal everyday use, the emerging computational grid will provide the additional computational power needed when the local resources are insufficient.

The Grid that exists today is not yet complete.[28] There certainly is today a large collection of computational power and a large collection of scientists and engineers who could use this power. In the future Moore's law will continue to deliver us with increased CPU power and the pool of users will expand to include more traditional and non-traditional users. Making effective use of these resources easy for users is then the focus of current Grid research.

Using this power is difficult for a number of reasons. Foremost amongst these reasons is that very little of this power is directly under your control. The owners of these resources may very well be perfectly willing to share or be compensated for letting you use these resources, but are unwilling to enter into an agreement which sacrifices their control over these resources. Additionally, these resources may not be able to be totally dedicated to a user, with availability subject to local demand. Finally, the resources may very well simply be heterogenous. Perhaps there are 100 CPU's available, but half of them are running Windows NT and the other half Linux. What results from all of these difficulties is a fragmented sea of compute power that the potential user is unable to navigate.

The Condor project[20] at the University of Wisconsin-Madison has been helping users make use of these resources for nearly 15 years. Condor is a high-throughput scheduling system designed to foster a community between the resource owners and resource users, and has delivered centuries of compute time to users over it's lifetime. Condor provides users with a friendly execution environment, trapping system calls[19] a job may make and forwarding them back to a known host where they are performed on the job's behalf. Condor also makes guarantees to the resource owners that Condor jobs will not interfere with the owner's use of the machine, and will checkpoint and migrate the currently running job to a different machine on the network. This feature both returns the workstation to the owner on demand, and preserves the work the process has completed up to that point. Condor is being enhanced to support the emerging Grid, and soon

This work is being done as part of course project in CS736 - Advanced Operating Systems at the University of Wisconsin, Madison. None of this work was supported by the Defense Advanced Research Projects Agency, or the National Science Foundation, or any other branch of the United States Government. The content of this paper does not necessarily reflect the position of the Condor Project at the University of Wisconsin-Madison. This dead space would not be here if I knew LaTeX a little better.

Condor users will be able to access local and national resources seamlessly and securely.

In order to use the checkpointing and remote system call facilities of Condor, users are required to relink their program with a modified version of the C library. This requirement is a result of both needing to inject the checkpointing code into the program as well as being able to reconstruct the kernel-state information of the program after the checkpoint. If mechanisms to do this were provided in the kernel, Condor could use them and eliminate this relinking requirement. However, few operating systems provide these facilities, and while Condor could distribute a series of kernel patches to provide them, it has long been the philosophy of the Condor project that system should work without requiring kernel modifications.

This requirement of relinking leaves a large class of applications unable to take advantage of Condor's checkpointing and remote system calls. Commercial software vendors are often unwilling to provide source or object code, and there are surprisingly large numbers of users who are unable to relink their jobs (oftentimes because all the user has is an executable.) Condor calls such applications "Vanilla jobs", and does it's best to run them. With kernel-level checkpointing, Condor could put these applications on equal footing with relinked jobs. There is a large demand for such features; for example, one research group at the University of Wisconsin consumed 4 years of CPU time for experiments for a single conference, and has pledged to consume as many CPU years as can be provided. This pledge is the original impetus for this work. This research group was limited to running on a small subset of available compute power, and while they were able to consume 4 years another 20 years sat idle because they were unable to effectively use it.

John Masefield in his poem "Sea Fever" wrote that all he asked was "a tall ship and a star to steer her by", and it is this observation that is the key to our work: a user program has two requirements - an appropriate platform on which to execute, and a set of services it can call upon. Traditionally, the platform is the native hardware and the services are provided by the established Operating System. However, by using Virtual Machine Monitors, a classic technique from yesteryear, it is possible to use the native hardware at full speed while providing an alternate set of operating system services without disrupting the native operating system. By using Virtual Machine Monitors, we can provide programs with the appropriate set of services they need to run, regardless if they were present or not present on whatever CPU we are able to get, as well as the additional functionality we need in order to perform remote system calls and checkpointing.

The remainder of the paper will be organized as follows. Section 2 will examine just what it means to be a virtual machine, and section 3 will gather some performance numbers which will establish that the overhead of a virtual machine is acceptable to users. Section 4 examines some of the additional services that we are now able to perform with the additional layer of abstraction the Virtual Machine provides. Section 5 examines related work, and Section 6 provides some conclusions and proposes future work.

2. VIRTUAL MACHINES

"Virtual Machine" is a very overloaded term. There are a number of different levels at which it is possible to construct a virtual machine. In this section we will briefly examine several different examples of virtual machines.

2.1 System Simulators

A virtual machine at this level exists totally in software, and is the lowest-level virtual machine we will examine. A virtual machine at this level simulates the original hardware, perhaps at a clock-tick granularity or at the level of each instruction. They are particularly useful for examining system performance or debugging the low-level parts of an operating system, since it is possible to easily and arbitrarily examine any part of the system. They are also useful for simulating hardware that may no longer exist, so it becomes possible to continue to run legacy applications on modern systems. This particular application has caught on in the video-game emulator circuit, with classic video games being brought back with systems such as MAME. Other examples of this sort of virtual machine include SimOS, SimICS[21], SimpleScalar, and SPIM. This detailed simulation is particularly expensive. Applications under these sorts of systems can see several orders of magnitude performance hits, which limits there usefulness to pure systems of study, and not for production services.

2.2 Synthetic Machines

At a slightly higher level, we find synthetic machines such as Java and UCSD Pascal. These systems take a high-level language and compile it into a bytecode that is meant to be interpreted at runtime. The bytecode is not tied to a specific architecture, which allows for full portability. The object of these virtual machines is to produce the correct results of the program, and not faithfully reproduce the hardware of the original machine. Because they only need to produce the correct answer, many steps in the simulation can be eliminated, and the program can be executed at as near to native speed as possible. This level of virtual machine is usually fast enough to use for interactive applications. However, there are two major disadvantages of using this approach. First, applications must be recompiled or translated into this bytecode, usually by porting the entire source code over from the native language and into Java or Pascal. Second, the performance hit can still substantial, and the techniques to improve them oftentimes sacrifice portability. Because of the perceived performance hit and need to translate, synthetic machines have not become the platform of choice for potential Grid users, even with the nearly-universal portability they offer.

2.3 Virtual Machine Monitors

A Virtual Machine Monitor overcomes the two disadvantages of the synthetic architectures of the last section by running code natively on the underlying hardware. This, of course, limits portability to the architecture on which the

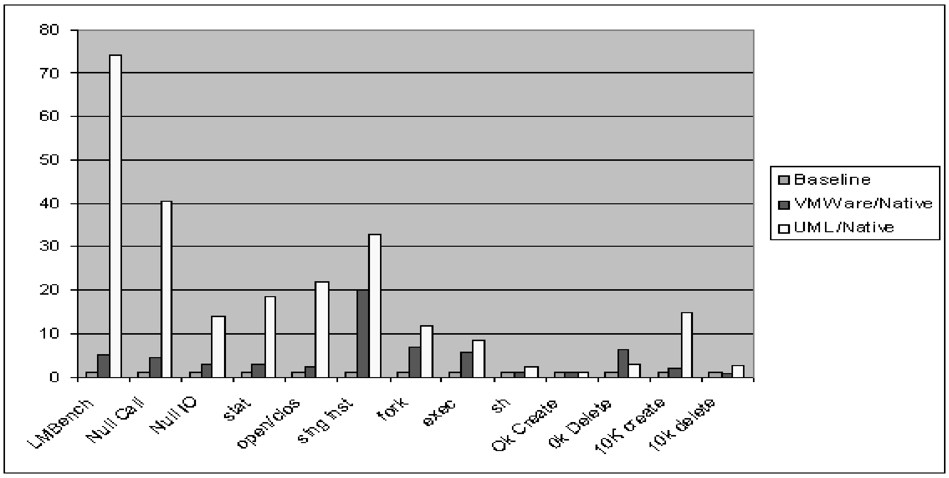

Figure 1: Standard Operating System operations under lmbench (lower is better.) These operations are considerably more expensive under the Virtual Machines than they are on the native operating system.

program to be run is compiled for, but the lack of a user-code speed penalty and ability to to run any binary can outweigh the advantages of portability. Virtual Machine Monitors are possible because most computer systems maintain a notion of "User" mode and "Supervisor" mode, in which any code running in "User" mode is unable to interfere with any activity on the system other than its own. Attempting to execute instructions which require "Supervisor" privileges cause the CPU to trap into the Operating System, which can then examine the instruction the "User" code was attempting to access, and determine what steps next to take. A Virtual Machine Monitor handles these traps, either by running as the base-level operating system and passing these traps back to an operating system running beneath it, or by running underneath an already-established operating system and requesting that any such faults a program it may be monitoring be passed to it. Because only "Supervisor" instructions can view other parts of the machine, and all such instructions are emulated by the Monitor, Virtual Machine Monitors are able to run unmodified binaries on unmodified operating systems, with neither the user job or operating system aware that anything is amiss.

Virtual Machine Monitors are not new ideas. By 1974 they were mature enough to be making a "comeback", and the definitive guide to the early Virtual Machine Monitors can be found in Goldberg's survey paper.[13]. IBM is certainly the best-known vendor of Virtual-Machine capable systems, with the historical VM/370 system being described by Creasy [7], and their modern S/390 systems running Linux more recently.[31]. Virtual Machine Monitors were a cold topic through much of the 80's as the Operating System world traveled off on a Microkernel and Userspace adventure, but the idea resurfaced in the 1990's with Disco[6], which was later commercialized in the form of VMWare. In order to provide a perfect Virtual Machine Monitor, all instructions which affect the state of the system must be trapped and handled by the Monitor. Sadly, not all architectures provide this needed mechanisms. Hall[14] describes the challenges of virtualizing the VAX architecture. Robin[27] describes the challenges of virtualizing the Intel architecture with an eye towards security, and Lawton[17] describes the Intel architecture with an eye towards multiple operating systems.

2.4 API Level

It is also possible to provide a virtual machine at the API level. The best example of this is the WINE system, which provides Win32 applications with the needed services to run on UNIX machines. By many definitions, including that of the WINE developers, these sorts of systems are not virtual machines, and we mention them only for completeness and do not consider them further.

3. PERFORMANCE

The flexibility provided by Virtual Machine Monitors is not without cost. Each privileged instruction must be caught first by the monitor, evaluated and possibly emulated, and only then have controlled passed back to either the operating system or the monitored application. While this overhead is only paid on those privileged instructions, codes making frequent use of them will run noticeably slower than those systems which can natively handle all instructions. In our evaluation, we compared the performance of two microbenchmark suites and two production Condor pool codes to gain insight as to what sort of overhead users of Virtual Machine Monitors will be faced with. The two benchmarks we evaluated were lmbench[22] and the SPEC series of CPU-intensive codes.

Our evaluation systems were PentiumIII-class workstations at the University of Wisconsin running the RedHat Linux distribution version 6.2. We evaluated the microbenchmarks under three different conditions. The first was running the benchmarks natively with no Virtual Machine involved. We then ran under VMWare 2.0, with RedHat 6.2 as both the host and guest operating system. As a final experiment, we ran under User-Mode-Linux[8], which is a port of Linux to it's own system call interface. UML and VMWare perform

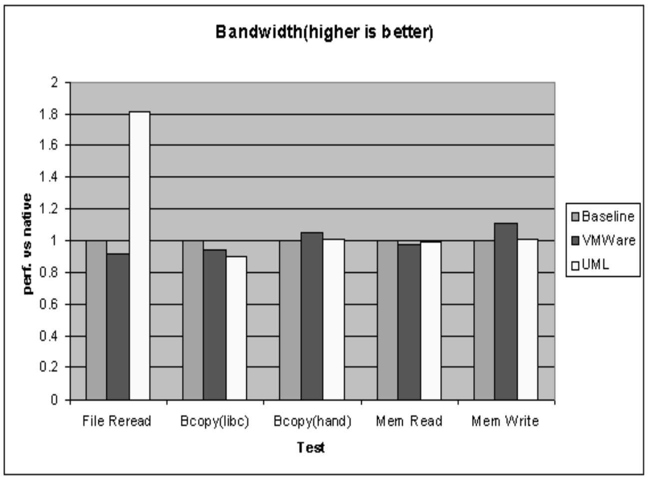

Figure 2: lmbench memory bandwidth tests. These rely less on operating system features and more on raw processor performance, so they perform considerably closer to native speed

the virtualization at different levels. VMWare presents an entire set of abstracted hardware directly to a guest operating system, which does not realize that it is not talking to a real Ethernet or VGA card. UML uses the `ptrace(2)` interface to perform it's virtualization, much like the Catcher interface in the UFO[1] system. What makes UML interesting is that it has linked the Linux Kernel into a regular application, and uses the architecture-dependent subsystems of the code to trap privileged instructions, and the architecture-independent parts of the code to handle them. This results in a regular user-level application that when run starts up an entire virtual machine in an ELF binary, unaware that it is really nothing more than a regular user-level process.

Our first experiment is shown in figure 1, with the performance of the two virtual machines against a normalized native version of the same test. In the first experiment, we compare some standard operating system functions such as file creation and process forking. Not surprisingly, these features are considerably more expensive in the the Virtual Machine Monitors than they are running natively. User-Mode-Linux is consistently a poorer performer than VMWare, sometimes by as much as an order of magnitude. This is probably a result of the mechanisms used to construct the Virtual Machine Monitor. VMWare uses a loadable-kernel module in the native operating system, whereas the `ptrace` debugging interface used by UML is notorious for poor performance. In the null OS call test, for example, UML was 74 times slower than the native test. Clearly, a system that makes heavy use of these facilities will be penalized for running under a Virtual Machine Monitors.

However, in our second set of experiments we discover that not all aspects of Virtual Machines are slower. This experiment, shown in figure 2, ran several of the memory bandwidth tests from lmbench and in all cases had comparable performance to the native machines. The lone exception to

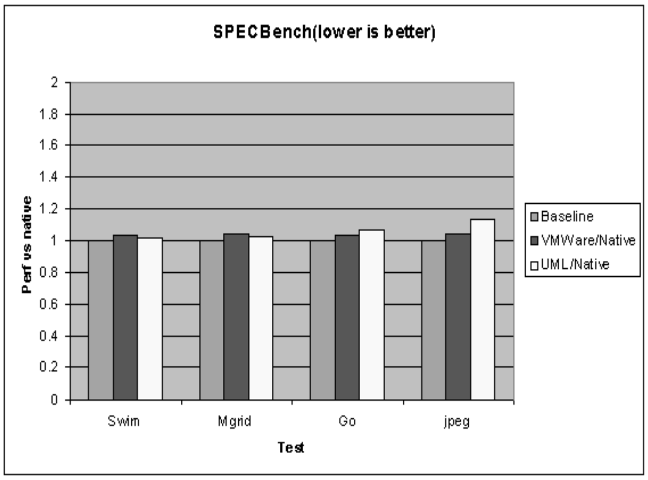

Figure 3: SPEC95 Performance of 4 benchmarks under Virtual Machines. Again, these codes stress the CPU, not the OS and perform quite close to native speeds

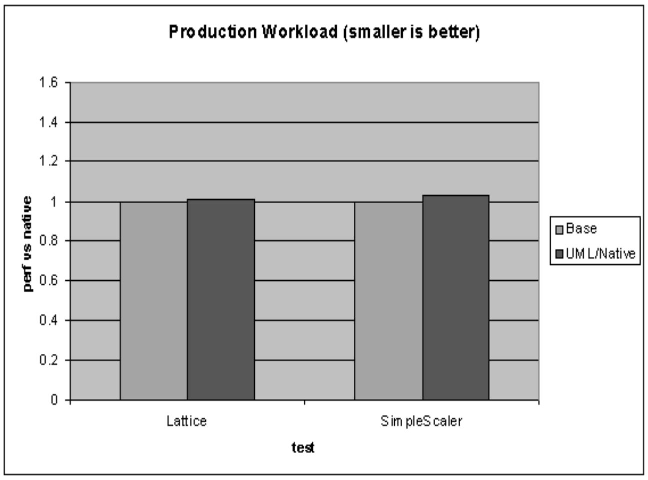

Figure 4: Performance of production-codes on native hardware vs the same code under a Virtual Machine Monitor. Both of these codes are heavily CPU-bound, so they perform virtually identically under both the native hardware and the virtual machine

this was the file-reread test under User-Mode-Linux, which actually performed nearly twice the speed of the native test. This is probably an artifact of the double-buffering that occurs between the User-Mode kernel and the native kernel, but is an area for future experiments.

Our third set of experiments, shown in figure 3, ran 4 benchmark codes from the SPEC95 suite (two floating point and two integer.) These codes are designed to test CPU performance, and are very similar to the sorts of codes Condor and early Grid users run. As expected, performance of these codes under the virtual machine is nearly identical to that of the native machine. Swim and Mgrid are both simulation codes and run for about 10-20 minutes, and show a 1-3 percent speed penalty. Go and JPEG, which play a game of Go and compress image files, respectively, run for about 1 minute and have a 6 and 12 percent speed penalty. This increase is probably a result of their short runtimes, and they have not had time to amortize the expensive file operations they perform at their startups. Finally, in our fourth set of experiments in figure 4, we run two production codes currently running in the Condor pool at the University of Wisconsin. The first is lattice, which is a computation chemistry code from UCLA. The group that developed this code has a nearly unbounded CPU need, and at the last Supercomputer allocations meeting received more time than all other users combined. The other example is SimpleScalar, a low-level virtual machine that consumes between 10 and 30 percent of the daily Condor cycles. Both of these examples are long-running codes, and both have a negligible performance penalty for running under the User-Mode-Linux kernel.

4. PROVIDING EXTENDED SERVICES

Now that we have demonstrated the performance degradation of using Virtual Machine Monitors is minimal, we examine three new services that we can provide to Grid applications that were previously difficult or impossible to efficiently and correctly provide for arbitrary applications.

4.1 Split Execution

Split execution is a model of providing a friendly execution environment for an application to run under on a foreign machine. By "friendly" we mean that all of the runtime libraries it requires are present, and the program has access to the data it will need during its lifetime. Condor provides this friendly environment by the remote system-calls alluded to earlier in the paper, but leaves it up to the site administrator to ensure that the needed libraries are present on all machines. For those jobs that cannot be relinked to take advantage of remote system calls Condor cannot provide anything for. Bypass[29] uses the dynamic linker to intercept POSIX calls, and through this can construct and provide a rich set of stackable alternatives implementations of the calls the application made. Bypass does not address the problem of missing libraries (it can help for those libraries explicitly loaded via `dlopen()`, but cannot help for libraries loaded immediately by the host operating system.) It also is unable to help applications that are either statically linked or do not use the POSIX interface the system.

Running under a Virtual Machine Monitor allows us to construct an operating system which both provides a mechanism to perform remote system calls on the applications behalf, regardless of how that system call was triggered. We are also able to provide a consistent environment in which the application will run in. This has the immediate benefit to the user in that they can establish exactly what their application will need and have available ahead of time, and can completely eliminate a source of very frustrating bugs and other roadblocks to distributed computing. For example, perhaps the user wants to guarantee a certain version of the math library their code will be linked with (perhaps they know there was a bug in an older version but do not know if every site on the Grid has upgraded yet.)

4.2 Checkpointing

Checkpointing is committing to stable storage all the state necessary to recreate a given process exactly as it was at some point in the future. Dijkstra observed many years ago that the only thing that has meaning in a computer program is the logical succession of states, and not at the speed at which those successions take place. Checkpointing is this idea taken to the extreme. In a distributed system such as Condor, checkpointing is most often used to provide for process migration and fault-tolerance. A checkpoint file is really nothing more than an annotated core file, and is described in the report by Litzkow et all[18]. As mentioned earlier, in order to take advantage of checkpointing, users are required to relink their programs with the Condor libraries, and those applications which are not able to be relinked are not able to be checkpointed by Condor. This requirement stems from needing a small bit of kernel state that is associated with each process, such as the current file position of each open file. It would be trivial to add an extension to the kernel which provided this information in `/proc`, but a kernel patch of any kind is not an option.

Modifying the kernel the virtual machine provides is an option, of course, and therefore we propose to add checkpointing to this kernel. At this point providing this service remains a work in progress. Several kernel-level checkpointing patches have been made available, but all suffer from some deficiencies. Common to all of them are their age; most were designed for the 2.0 kernel and none of them work under the modern 2.4 kernel. The best-known of these patches is epckpt[25]. epckpt works by adding two new system calls, `checkpoint()` and `restore()`. The checkpoint code works by patching the signal-handling code of a process, and the system call uses this modified signal handler to write out the current state of process. epckpt currently only works under kernels 2.2.1 and earlier. Another package is CRAK[15], which is based on epckpt. CRAK is interesting because it implements checkpointing not as a kernel patch but as a device, `/dev/ckpt`. Reading from this device gets you a checkpoint, writing to it restores a process. The advantage of this approach is that it requires no source changes to the kernel - a privileged user can simply load a module and have is able to checkpoint a process. CRAK has two major problems. First, it incorrectly handles floating point state. Second, because it is implemented as a device, it performs the checkpointing code in kernelspace but in a different process-context. This means that the virtual memory maps may be incorrect. CRAK works around this by walking the in-kernel virtual memory structures itself, but this is problematic when the page has been swapped out. It becomes necessary in that case to essentially implement all of the virtual memory subsystem in the device, which is not practical. There is a kernel helper-function which could solve all these problems, but it requires a single line patch to expose it to the device level, and this patch has twice been rejected for inclusion in the main kernel tree. A user would therefore have to install this patch manually, at which point the advantages of not having to patch the kernel just to use this device becomes moot.

We are currently implementing a new checkpointing interface for the Linux kernel with several goals. The first is that it is implemented in such a way as to separate the architecture dependent features from the architecture independent higher-level portions of the kernel. All current checkpointing patches are not structured this way, and as such have met with opposition from the core kernel development team whenever they are submitted for consideration. While the goals of this work are to provide this feature for users in a kernel that we are free to modify, the engineering work needed to complete it is considerable. There are also many applications and users who would be interested in seeing such functionality in the mainstream kernel, and the author has always wanted to be a kernel hacker. The second goal is to make any such interface conform as closely as possible to the POSIX[16] specification that exists for checkpointing.

4.3 Heterogenous Process Migration

We have shown that we can execute code at native speed while providing an alternative set of operating system services. It follows, therefore, that we should be able to use the checkpointing services provided in the previous section and the friendly environment provided for in the section before that to migrate processes between heterogenous systems, so long as they share the same underlying hardware. For example, a program should be able to run on an Intel/WindowsNT machine, then migrate to an Intel/Linux machine, and the move back to an Intel/Windows machine and it should all work at native speeds. While nothing new technically is offered in this section, having this work in the future would be perhaps the strongest vindication that this project is a valid and interesting effort. What is needed yet is a sufficient VM implementation under Windows NT, and the best hope for this currently is LINE[33] combined with a Cygwin port of the rest of the Linux kernel to produce a Windows version of User-Mode-Linux.

5. RELATED WORK

This work appears to be quite similar to that of Nasika, Boyd, and Dasgupta [24][3][4] at Arizona State University. From the published work it does not appear that they have a production system at this point. Their work also uses virtualization to provide new services to applications, and they have been developing their system on Windows NT. They claim to be building a Single-System-Image federation of machines on top of commodity operating systems.

Process migration and checkpointing has been a feature in many operating systems developed over the years, though few have been successful. One of the best-known was Sprite[9], and a survey of several techniques can be found in Milojicic[23]. The Flux project at the University of Utah[32][10] has been creating microkernels that provide native virtual-machine techniques, as well as exporting enough information from the kernel to allow user-level checkpointing to be possible. It differs from our work in that it requires that at the lowest-level the Flux system be running, whereas ours works on unmodified, commodity operating systems. Other examples of checkpointing systems include Libckpt[26] and Theimer[30]. Both of these systems require either sourcelevel, compile-time, or link-time changes. Mosix[2] is a popular system aimed at Clusters of Workstations, and is able to checkpoint and migrate unmodified binaries. It requires the user install a set of kernel patches. An interesting fusion of this work with Mosix would be to boot a Mosix-capable kernel on top of a Virtual Machine Monitor, bringing Mosix to unmodified systems. Process Hijacking[35] brings checkpointing to unmodified binaries on unmodified operating systems by using the binary-rewriting tools of the Paradyn project. It has an even smaller performance overhead, but is not capable of the heterogenous process migration that a Virtual Machine Monitor could provide. Finally, Virtual Machine techniques have been proposed [5][34][12] as a security and fault-tolerance tool.

6. CONCLUSIONS AND FUTURE WORK

We have presented Virtual Machine Monitors as a solution to the difficulties encountered in running on large collections of heterogenous, distributed computer systems such as those found in existing Condor pool and in future Computational Grids. We have examined the performance overhead of running the compute-bound jobs typical of current Condor and future Computational Grid users and found it to be minimal. We propose using Virtual Machine Monitors to construct new mechanisms to enable new opportunities for users to take advantage of CPU power that is currently available in but in an unusable form.

Future work will concentrate most on completing the checkpointing system described in section 4.2. Until this system is complete this work is not particularly relevant. There is also much work to be done with User-Mode-Linux. There are a number of virtualization techniques that still need to be applied to User-Mode-Linux in order to provide a more complete and secure system. There are also a number of performance concerns which must be more firmly established and mitigated. There is also an emerging trend of "Data-Intensive Computing" in the Grid Community, and the impact of running such an workload must be examined in a Virtual Machine environment. Finally, the Condor system should be extended to understand how to deal with running user jobs under a Virtual Machine Monitor, both in the local pool and in Condor-G, the Grid-Enabled version of Condor.

7. ACKNOWLEDGEMENTS

The author would like to thank Remzi Arpaci-Dusseau his helpful suggestions, comments, and flexibility; Miron Livny for providing a flexible work environment that allowed for meeting the obligations of both this work and his regular work; Todd Tannenbaum, Derek Wright, Peter Couvares, and Peter Keller for understanding when the author missed a staff meeting or a group lunch; his friends for putting up with repeated explanations as to why Virtual Machines were the answer to whatever problem they were facing at the moment, even if they didn't have a problem at the moment; Doug Thain, Nick Coleman, and Jim Basney for many helpful comments, and John Bent for encouragement, comments, suggestions, and explaining that Yellow is the Power Color.

8. REFERENCES

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. ACM Transactions on Computer Systems, 16(3):207–233, August 1998.

[2] A. Barak, O. La'adan, and A. Shiloh. Scalable cluster computing with mosix for linux. In Proc. 5-th Annual Linux Expo, pages 95–100, May 1999.

[3] T. Boyd and P. Dasgupta. Injecting distributed capabilities into legacy applications through cloning and virtualization. In The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), July 2000.

[4] T. Boyd and P. Dasgupta. Virtualizing operating systems for seamless distribution. In 12th IASTED International Conference on Parallel and DistributedComputing Systems (PDCS 2000), November 2000.

[5] C. Brenton. Honeynets. http://www.ists.dartmouth.edu/IRIA/.

[6] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, 1997.

[7] R. J. Creasy. The origin of the VM/370 time-sharing system. IBM Journal of Research and Development, 25(5):483–490, Sept. 1981. Comment: In Keith Lantz's course notes.

[8] J. Dike. A user-mode port of the linux kernel. In Proceedings of the 4th Annual Linux Showcase, 2000.

[9] F. Douglis and J. Ousterhout. Process migration in the sprite operating system. In Proceedings of the 7th International Conference on Distribu ted Computing Systems, pages 18–25, September 1987.

[10] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, S. Goel, and S. Clawson. Microkernels meet recursive virtual machines. Technical Report UUCS-96-004, Department of Computer Science, University of Utah, May 1996.

[11] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infastructure. Morgan Kaufmann, San Fransico, CA, 1999.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In Proceedings of the 6th USENIX Security Symposium, July 1996.

[13] R. P. Goldberg. Survey of virtual machine research. IEEE Computer, 7(6):34–45, June 1974.

[14] J. S. Hall and P. T. Robinson. Virtualizing the vax architecture. In Proc. 18th Annual Symposium on Computer Architecture, pages 38–389, August 1991.

[15] H. Z. Hau Zhong. CRAK: Linux Checkpoint/Restart as a Kernel Module. http://www.cs.columbia.edu/ huaz/english/research/crak.htm, December 2000.

[16] IEEE. POSIX P1003.1m, 1998.

[17] K. Lawton. Running Multiple Operating systems on an IA32 PC using virtualization techniques. http://www.plex86.org/research/paper.txt, November 1999.

[18] M. Litzkow, T. Tanenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, Department of Computer Sciences, University of Wisconsin-Madison, April 1997.

[19] M. J. Litzkow. Remote unix: Turning idle workstations into cycle servers. In Proc. of the 1987 Usenix Summer Conference, pages 381–384, 1987.

[20] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In Proceedings of the 8th International Conference on Distributed Computing Systems, pages 104–111, 1988.

[21] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm, and B. Werner. Simics/sun4m: A virtual workstation. In Proceedings of USENIX Annual Technical Conference, June 1998.

[22] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In Proceedings of the 1996 USENIX Technical Conference, January 1996.

[23] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. Technical Report HPL-1999-21, HP Labs.

[24] R. Nasika and P. Dasgupta. Transparent migration of distributed communicating processes. In 13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS-2000), August 2000.

[25] E. Pinheiro. Truly-Transparent Checkpointing of Parallel Applications. http://www.cs.rutgers.edu/ edpin/epckpt/, December 2000.

[26] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In USENIX Winter 1995 Technical Conference, January 1995.

[27] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In Proceedings of the 9th USENIX Security Symposium, August 2000.

[28] J. M. Schopf and B. Nitzberg. Grids: The top ten questions. Technical Report CS-00-05, Department of Computer Science, Northwestern University, 2000.

[29] D. Thain and M. Livny. Bypass: A tool for building split execution systems. In Proceedings of the Ninth IEEE Internation Symposium on High Performance Distributed Computing, pages 79–86, August 2000.

[30] M. M. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In IEEE 11th International Conference on Distributed Computing Systems, pages 18–25, May 1991.

[31] A. Thorton. Linux on the system/390. In Proceedings of the 4th Annual Linux Showcase, October 2000.

[32] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In Proc. Fifth International Workshop on Object Orientation in Operating Systems, pages 85–89, October 1996.

[33] M. Vines. LINE: Line is not an Emulator. http://neomueller.org/ isamu/line/, December 2000.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Grahm. Efficent software-based fault isolation. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203–216, December 1993.

[35] V. C. Zandy, B. P. Miller, and M. Livny. Process hijacking. In Proceedings of the Eighth IEEE Internation Symposium on High Performance Distributed Computing, August 1999.