Process Hijacking

Victor C. Zandy Barton P. Miller Miron Livny
{zandy,bart,miron}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

Process checkpointing is a basic mechanism required for providing High Throughput Computing service on distributively owned resources. We present a new process checkpoint and migration technique, called process hijacking*, that uses dynamic program re-writing techniques to add checkpointing capability to a running program. Process hijacking makes it possible to checkpoint and migrate proprietary applications that cannot be re-linked with a checkpoint library, and it makes it possible to dynamically hand off an ordinary running process to a distributed resource management system such as Condor. We discuss the problems of adding checkpointing capability to a program already in execution: (1) loading new code into the running process, and (2) replacing functions of the process with calls to dynamically loaded functions. We use the DynInst API process editing library, augmented with a new call for replacing functions, to solve these problems.*

We discuss problems associated with migrating a hijacked process: (1) preserving the uncheckpointable operating system state of the hijacked process for migration, and (2) safely restoring the dynamically assembled address space of the hijacked process from a checkpoint. We preserve uncheckpointable operating system state by spawning a shadow process from the hijacked process. We have used process hijacking to migrate a variety of programs, including a running, unmodified Java VM. We show that the migration performance of hijacking is comparable to that of Condor.

1 Introduction

Process checkpointing contributes to the flexibility and power of a distributed resource management system by allowing processes to be migrated to other hosts during their execution[8]. Systems such as Condor[7] and Codine[5] use checkpointing to dynamically schedule long-running programs. A characteristic of the checkpoint techniques employed by these systems is that the application must be re-linked with a checkpoint library before it can be submitted. This requirement prevents the submission of proprietary executables, since re-linking depends on access to the original object files of the program. In addition, a program already in execution cannot be handed over to a resource manager, since that would require the ability to re-link a running process. We present a new process checkpointing technique, called *process hijacking*, that enables these types of programs to be submitted to a resource management system. It includes support for remote I/O to allow hijacked processes to be migrated to foreign administrative domains.

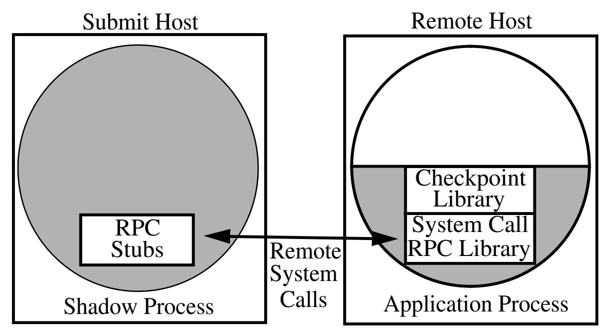

Figure 1: A Hijacked Process
The shadow process remains on the original submit host. The application process can be migrated to a remote host.

Process hijacking uses dynamic program re-writing techniques to add checkpointing capability and remote I/O to ordinary processes. It creates an execution context similar to that created by Condor. A hijacked process is split into two processes (Figure 1): the *application process*, and the *shadow process*. The application process is the original process to which the hijacker adds checkpoint and remote

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

system call support. It originally runs on the *submit host*, and after it is hijacked, it can be migrated to a *remote host*. The shadow process is started on the submit host to provide a stable context for the application process. When the application makes a context-sensitive system call, the call is executed by the shadow via remote procedure call. The RPC stubs for the application are contained in a library called the *remote system call library* that is dynamically loaded into the application by the hijacker. The hijacker rewrites the application to use the RPC stubs instead of the standard system calls. The most important type of context-sensitive system calls are those related to I/O (calls that involve file descriptors). Remotely executing all I/O operations on the shadow allows the application process to be migrated to hosts that do not have the file system resources or security credentials of the submit host. To add support for migration, the hijacker also loads a *checkpoint library* into the application. Checkpointing is later triggered by sending a signal to the application. When the process is restarted, it resumes communication with the shadow process, transmitting remote system calls from its new host.

Process hijacking is interesting for three main reasons:

- ❏ It expands the scope of checkpointing. Users with already running or proprietary programs can now benefit from the checkpointing services offered by distributed resource management systems. Process migration without the need to re-link in advance makes migration a viable alternative to program termination or suspension when unanticipated changes in a resource availablity arise. For example, a process running on a machine that is close to exhausting its swap space can be hijacked and checkpointed to convert its swap space usage into disk usage.
- ❏ It introduces technology that can generalize access to metacomputing. Process hijacking replaces some of the system call functions of the hijacked process with dynamically loaded substitutes. In effect, the process is linked again while it is running, allowing programs already in execution to be dynamically modified to redirect I/O, access authentication services, and interact with a scheduling service.
- ❏ It is a concrete demonstration of the power of runtime code modification. Adding checkpointing capability to a program while it runs requires the ability to modify its code. With a toolkit that makes editing running programs simple, as found in the DynInst API[6], these modifications are as easy as ordinary data manipulation. Process hijacking shows that runtime program rewriting, previously shown to be useful for debugging and performance profiling[10], is also useful for resource management.

To hijack a process, and to support migration of a hijacked process, we have overcome four main technical challenges:

- ❏ Dynamically inserting code into and controlling the execution of the hijacked process: We use the DynInst API runtime code re-writing library to modify and control the process. This library provides an architecture-independent interface for splicing new code sequences into a running process, and for installing new code libraries (see Section 2).
- ❏ Replacing the system call functions of a running process: We developed a new extension to the DynInst API, called replaceFunction, to replace the original system call functions with the remote system call RPC stubs. replaceFunction provides a general mechanism for re-linking a running program (see Section 3.2).
- ❏ Migrating a process with a dynamically loaded checkpoint library: A checkpointed process has an arbitrarily arranged address space. The restart code must be careful to place itself out of the way when it reconstructs the address space (see Section 4).
- ❏ Preserving the operating system state of the hijacked process for migration. Process state that is hidden in the operating system, such as open file state, cannot be checkpointed directly. We preserve this state in the shadow process, and use remote system calls to allow system calls involving this state to be serviced (see Section 3.1).

We have implemented process hijacking in a tool called the Process Hijacker that runs on UltraSPARC workstations running Solaris 2.6. The remote system call and checkpointing libraries are derived from the libraries used in Condor. Excluding DynInst and the libraries, the Hijacker is approximately 1000 lines of C and C++. In Section 2 we briefly describe the DynInst API, the runtime program re-writing library that we used to implement the Process Hijacker. The modifications performed to hijack a process are explained in Section 3. The aspects of migration unique to process hijacking are described in Section 4. Section 5 describes the performance of the Process Hijacker, showing a breakdown of costs.

2 DynInst API

DynInst is an architecture-independent API for making on-the-fly modifications to a running program. Figure 2 shows the organization of a DynInst API application. A program, called the *mutator*, is linked with the DynInst API library and makes API calls to control and modify the application program, called the *mutatee*. The mutator attaches to the mutatee with the usual process debugging interface provided by the operating system, such as ptrace

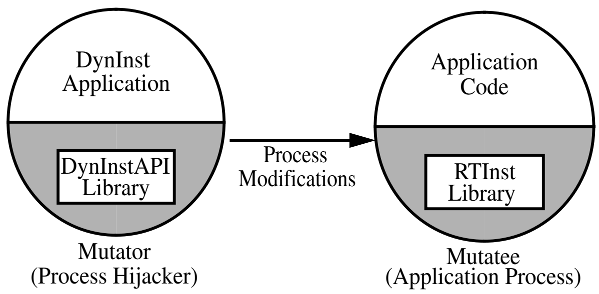

Figure 2: DynInst API Operation

or `/proc` on Unix or the process control API on Windows/NT. Some operations, such as reading and writing the memory of the mutatee, are performed by DynInst directly through this interface. Other more complex operations, such as allocating memory, are executed by a library installed by DynInst in the mutatee, called the *run-time instrumentation library* (RTInst). DynInst can operate on any dynamically linked executable. It does not require special preparation of the executable such as re-compiling or re-linking, and it is not necessary for the executable to contain debugging symbols.

With DynInst, the mutator can splice *code patches*, sequences of machine instructions, at the entry, exit, or call sites of a function in the mutatee. DynInst provides an architecture-independent mechanism for specifying code patches in terms of familiar program data and control flow operations, including assignment, logic and arithmetic, branching, and function calls. The mutator can also make an inferior RPC into the mutatee to cause it to asynchronously execute a code patch.

When a DynInst mutator attaches to a process, it first parses all of the code in the process to find instrumentation points. We have modified DynInst to parse only the functions that will be instrumented by the hijacker, to reduce the parsing time when the hijacker is started.

3 Hijacker Operation

We describe and motivate the operation of the Process Hijacker. Figure 3 shows a process before and after it is hijacked. Before (Figure 3a), the application is an ordinary process linked with the system call functions of the standard C library. After (Figure 3b), four things are different: (1) it has loaded the checkpoint library and remote system call library, (2) it has a new signal handler (not shown) that triggers a checkpoint, (3) its context-sensitive system calls have been replaced with remote system calls, and (4) a shadow process is running to handle the remote system calls.

Currently system calls cannot be safely replaced if a system call function is active on the stack(s), because resuming such a system call may invalidate the state of the shadow process. The Hijacker can detect this situation by inspecting the stack. It can be handled by intercepting the return from the system call, and then proceeding with the hijacking. Since we inherit the limitations of Condor's checkpointing mechanism, processes that spawn children, communicate with other processes, or run with multiple kernel-level threads currently cannot be migrated. The hijacker assumes that all system calls generated by the application process are handled by the standard C library system call interface; system calls that bypass this interface are not forwarded to the shadow process.

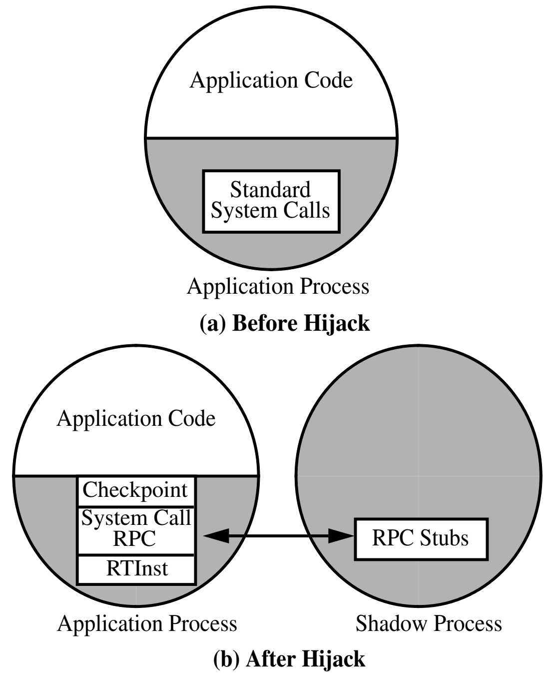

Figure 3: A Process Before and After Hijack

To hijack a process, the Process Hijacker performs the following operations through calls to DynInst (see Figure 4):

1. Attaches to the application process, stops it, and loads the RTInst library;
2. Loads the checkpoint and replacement system call libraries into the application;
3. Saves the open file state of the application process in the shadow process by forking the application process (Section 3.1);
4. Replaces the application's system calls (Section 3.2);
5. Initializes the shadow communication and installs the checkpoint signal handler;
6. Restarts the application process and detaches.

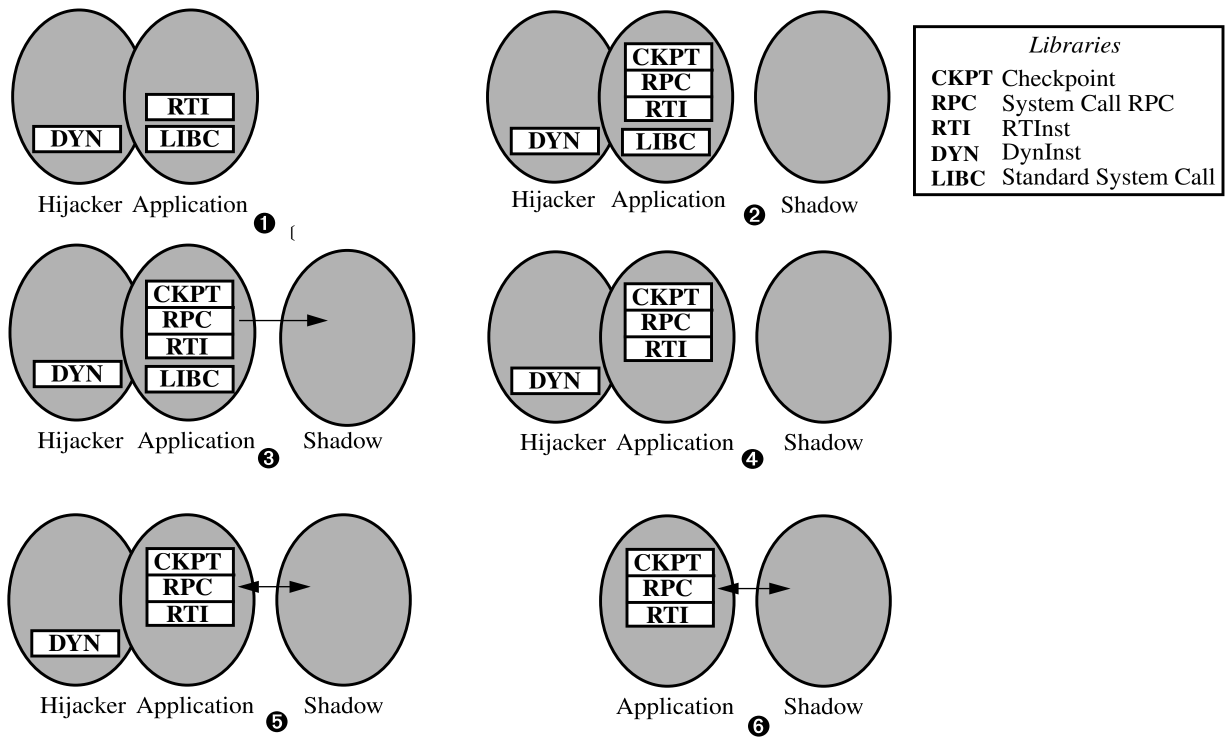

Figure 4: The Steps of Hijacking

(1) The hijacker attaches to the application; (2) The application loads the checkpoint and remote system call libraries; (3) The application spawns the shadow process; (4) The hijacker replaces the application system calls; (5) The application initializes its shadow connection; (6) The hijacker detaches.

Steps 1 and 6 use the primitive DynInst operations for attaching and detaching to a mutatee. Step 2 is performed with the DynInst operation loadLibrary, which causes the mutatee to load a new library. Step 5 is performed with oneTimeCode, the DynInst operation for making inferior RPC into the mutatee. The most interesting steps are creation of the shadow process (Step 3), and replacement of the system calls (Step 4). The remainder of this section explains these two steps.

3.1 Preserving Process State with the Shadow

The shadow process serves two main functions: (1) as in Condor, it executes context-sensitive system calls in the submit host context, and (2) it preserves open-file state between a checkpoint and subsequent restart of the application. We discuss the motivation for using the shadow process for preserving open-file state.

A running application program may have file descriptors that were opened before it was hijacked. These descriptors may include open files, devices, pipes or sockets, and processes. The same file may be open more than once with separate position pointers (from separate open calls) or with a shared position pointer (from dup calls). Checkpointing this state makes severe demands on an operating system. Since open works with file names, the names of files opened in the application process must be known to re-open them. However, finding the name of a file from a descriptor requires an inode search of the file system, since the operating system only maps descriptors to inodes, not file names. If the file happened to be unlinked after the descriptor was created, it is not possible to open it again (it will be deleted when it is closed). Even if it can be re-opened, the descriptor must be set to its previous position and possibly duplicated. Although lseek can be used to determine the position and duplication state of a descriptor, not all file types support seeking. Device files may have special close semantics (such as rewind or eject on closing) that must be undone before the device is re-opened.

To avoid addressing these problems directly, the Hijacker uses the fact that fork copies the file descriptors of the parent process to the child to create a complete copy of the application's open file state in the shadow process. An advantage created by this approach is that it allows the migration of distributed applications that communicate with sockets, since the shadow process can be a fixed location for the communication ports. A disadvantage of this approach is that the shadow process must live until the application process terminates. The shadow process is quite small, so it has trivial performance impact on the submit host, but if the submit host crashes, the application must be restarted.

The Condor shadow, in contrast, can be restarted because the Condor remote system calls are linked with the application process from the beginning of its execution. The Condor remote system calls record parameters passed to context-senstive system calls, such as filenames used in open calls and descriptor numbers used in dup calls, to allow the Condor shadow to reconstruct the file descriptor state when it restarts. The Condor shadow is more flexible, as it can survive a crash of the submit host, but it cannot be used with unmodified or already-running processes. To provide the same level of fault-tolerance to hijacking shadows would require an operating system mechanism for allowing handles to open files to persist across process termination and even system shutdown.

3.2 Replacing System Calls

We added a new method to the DynInst API, called replaceFunction, to replace the original system call functions of the hijacked process with the RPC stubs contained in the remote system call library. replaceFunction inserts a code patch at the entry point of the original function that contains a jump to the replacement function. The jump is designed so that the replacement function returns directly to the caller of the original function, not the replaced function. replaceFunction understands various code optimizations that affect function calls, such as abbreviated state saving and tail-calls. Figure 5 shows the operation of replaceFunction.

There are more powerful applications of replaceFunction than process hijacking. The code patch it inserts can contain arbitrary code, including code that references the parameters of the replaced function. A patch could thus select one of multiple replacement functions to call depending on the parameter values, which would be a useful mechanism for calling dynamically specialized functions. It is also a general mechanism for re-linking a program after it has started running. Conventional dynamic linking techniques can set the binding time of symbols to definitions at the start of execution, but not after. replaceFunction can be applied to change a function binding at any time during execution.

The costs of replaceFunction are a small amount of memory for the new code and a small time penalty for each call to jump from the original function to the replacement. The memory required does not depend on the number of potential callers of the function. The time penalty is negligible because usually the replacement function call makes an RPC to the shadow process, the time for which easily dominates the cost of the control transfer.

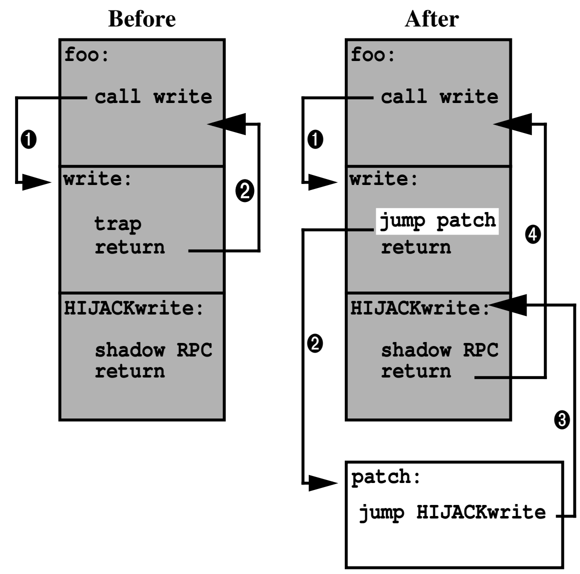

Figure 5: Function Replacement
The function write *is replaced with* HIJACKwrite.

There are other ways to dynamically replace functions, but the method we chose appears the most general. The point of modification, the entry point of the replaced function, is at the convergence of all control paths to the function. Only one modification per function is required, and it never needs to be repeated. Other techniques include modifying every point where the function is called, or modifying the dynamic linkage data structures. These techniques require that multiple modifications be made for each replaced function: one for every control path to the function (modifying call points), or one for every dynamic library containing a call to the function (modifying linkage data). They also require that the process be permanently monitored, so that the modifications can be applied again to new code dynamically loaded by the process. Our technique allows the Hijacker to attach to the process, rewrite it, and then disappear.

4 Process Migration

After a process has been hijacked, it is ready to be migrated. Process migration has two parts: checkpointing and restarting. A process checkpoints itself when a signal handler contained in the checkpoint library is triggered. The process is checkpointed (and later restarted) entirely within the signal handler context. With the exception of the use of the shadow to preserve file state, our checkpoint mechanism is identical to Condor's and will not be discussed further.

Program	Hijack Time	Checkpoint Time				Restart Time			
		Condor (chkpt server)	Hijack (chkpt server)	Hijack (local file)	Hijack (AFS)	Condor (chkpt server)	Hijack (chkpt server)	Hijack (local file)	Hijack (AFS)
tiny	1,094	2,009	3,192	48	3,641	1,801	3,378	55	3,498
big	1,364	12,876	12,977	221	14,406	14,635	14,702	209	15,330
kaffe	1,357	16,951	17,491	216	21,314	18,072	19,488	279	21,597
ss	1,342	3,168	4,507	67	5,328	2,922	5,004	77	5,250
path	2,088	48,490	48,791	5,254	76,357	54,505	53,334	5,130	82,825

Table 1: Hijack and Migration Times for Condor and the Hijacker
All times in milliseconds.

Program	Condor (MB)	Hijacker (MB)
tiny	1.6	3.1
big	12.1	13.6
kaffe	15.9	18.4
ss	2.7	4.6
path	50.1	52.2

Table 2: Checkpoint Sizes for Condor and the Hijacker

A checkpointed process is restarted by a process, called the *Starter*, that transforms itself into a continuation of the checkpoint. A delicate stage of this procedure occurs when the Starter replaces its own address space with the checkpointed address space. The Starter process contains a library, called the *Restart library*, that contains the code and data used to perform the address space replacement. If any region of the checkpointed address space coincides with the location of the Restart library, the Restart library will overwrite itself as it replaces that region of the address space, possibly crashing the restart. To prevent this, the Restart library must be loaded in a region that is disjoint from every region used by the checkpointed process.

We delay loading the Restart library until it is certain that it will be loaded in a safe place. We use the Solaris dlopen function to load it at runtime. Before it is loaded, we pre-allocate the regions that are allocated in the checkpointed address space (and not already allocated for the Starter). Then we load the Restart library. Since the regions of checkpointed address space are already allocated, the loader is forced to load it in a region from which it is safe to copy the checkpointed address space. A later stage of the restart procedure unloads the Restart library so that copies of it do not accumulate over multiple checkpoints.

Our restart mechanism requires two services from the operating system: (1) a means to discover the address space of a process (provided by `/proc` on Solaris) and (2) the ability to allocate specific regions of virtual memory (provided by mmap with the `MAP_FIXED` flag).

The restart code and data location problem does not occur in systems, such as Condor, that statically link the Restart library with the application. In these systems the application executable itself is used to restart a checkpoint (a command-line parameter determines whether the application should run normally or restart a checkpoint). Since static linking loads the Restart library at the same address each time the application is run, the checkpointed address space will contain a copy of the Restart library at the same address as the process that restarts the checkpoint. The Restart library will thus be preserved when this region of the address space is replaced. This arrangement would not be possible for the Process Hijacker without creating a custom Starter for each checkpoint that is statically linked with the Restart library in a safe location.

5 PERFORMANCE

We measured two aspects of the performance of the Process Hijacker: *hijack time* and *migrate time*. Hijack time is the total running time of the Process Hijacker, including the time for inserting the checkpoint and remote system call libraries and replacing the system calls. Migration time is measured in two parts: *checkpoint time* and *restart time*. Checkpoint time begins when the application process receives the checkpoint signal and ends when checkpoint is written. Restart time begins when the restart program is executed and ends just before the restart program returns to the application code of the checkpointed process.

We report the hijack time and migrate time under the Hijacker and Condor for five programs: (1) tiny, a small compute-intensive program, (2) big, a program similar to tiny, but with a larger data area (10MB), (3) kaffe, a Java virtual machine (1.6MB text, 8.2MB data) running a 4400 line compute-intensive Java program, (4) ss, a CPU simulator used by architecture researchers at the University of Wisconsin (1.2MB text, 1MB data), and (5) path, a mixed complementary solver used by mathematical programming researchers at the University of Wisconsin (2MB text, 46.6MB data). The performance results are summarized in Table 1. The checkpoint sizes for the measured programs are shown in Table 2. The processes were hijacked on and migrated between 250MHz Sun UltraSPARC 30s with 128MB of memory running Solaris 2.6.

Hijack time is between 1 and 2 seconds. Table 3 shows a cost breakdown of the major hijacking stages for the ss program. Real time includes the time spent executing inferior RPCs in the application process, while virtual time reflects only the CPU time (user and system) of the Hijacker. The stages are listed in the order they occur (see Figure 4). Attaching to the application is the most expensive stage, during which DynInst loads the RTInst library into the application process, scans the executable and shared libraries of the application to locate its functions, and parses the code of the replaced system call functions to identify their entry, exit, and call sites. The entry sites are the points at which replaceFunction laters inserts jumps to the RPC stubs. The next most expensive stage is loading and scanning the checkpoint and remote system call libraries. Replacing the system calls and initializing the shadow connection is the third most expensive stage. Starting the shadow process requires only one inferior RPC from the Hijacker process, but it results in significant activity by other processes when the application process executes the RPC and creates the shadow process, and when the shadow starts. Detaching from the application process is an inexpensive stage in which the Hijacker makes the application runnable and then exits. Since none of the stages read or write the data of the application process, data size is not a factor in hijack time, excluding paging effects.

For the migration time of hijacked processes, we compare the performance of three methods for checkpoint file I/O: (1) writing to a local disk, (2) writing to the AFS distributed file system, and (3) writing to the Condor checkpoint server. The time to transfer a checkpoint file to a new host is not included in the times for local disk I/O, but it is included in the times for AFS and the checkpoint server, since they both make the checkpoint file globally available.

The checkpoint and restart times using the checkpoint server are comparable to the times of Condor. Our times are slightly higher because our checkpoints are bigger than Condor's (due to large data areas in the DynInst RTInst library). A user who wants to hijack and migrate a medium-sized process (10MB) using a checkpoint server can expect it to take about 30 seconds.

A local disk provides fast checkpoint I/O, and is thus convenient for users who want to use checkpointing to yield machine resources, however it requires a user who wants to migrate the process to manually transfer the checkpoint to the destination host. Although AFS is a convenient mechanism for transferring checkpoint files, it is significantly slower than using a network checkpoint server. The performance of the Condor checkpoint server falls between that of the other two methods.

Hijack Stage	Real Time	Virtual Time
Attach	683	417
Load libraries	389	352
Start shadow	108	41
Replace syscalls	157	104
Detach	35	0

Table 3: Breakdown of Hijacking Costs for ss
All times in milliseconds.

6 Related Work

Process migration has been an operating system function since the early 1980s. It was a feature in DEMOS/MP[11] (1983), LOCUS[9] (1984), V[15] (1985), and Sprite[3] (1991). These research systems had the advantage that they could implement kernel primitives for accessing process state, making checkpointing and migration transparent to application processes. These systems were limited to migration between kernels (hosts) that were under the same administrative control.

More recent systems provide user-level checkpointing and migration on commodity kernels. These systems require the user to prepare the executable for checkpointing by linking with special libraries, inserting calls to checkpointing libraries in the source code, or using a special compiler. Condor[7], Codine[5], CoCheck[13,14], and MIST[1] require the application to be re-linked with a checkpoint library. Codine is a distributed resource management system that provides checkpoint and migration service to applications re-linked with a checkpoint library. CoCheck adds the Condor checkpoint library to PVM and MPI applications. MIST allows migration of PVM tasks by providing a special version of the PVM library that includes checkpoint fuctionality.

Other migration systems require access to the application source code. CLIP[2] is a checkpoint system for Intel Paragon multi-computers that requires insertion of calls to the checkpoint library into source code. Tui[12] and the Process Introspection Project[4] are heterogeneous process migration systems that provide special compilers to generate checkpoint capable programs. Tui supports heterogeneous process migration of programs written in common languages such as ANSI C. It uses a custom compiler that identifies properties of program data and execution points for the checkpoint system. The Process Introspection Project defines a platform-independent design pattern for expressing checkpointing information within the source code of an application. A special library and compiler automatically generate checkpoint capable executables from programs written in compliance with the pattern.

7 Conclusions

Process migration is one of the basic mechanisms required to provide High Throughput Computing service on distributively owned resources[8]. In this paper we report on a new checkpointing tool, the Process Hijacker, that expands the scope and range of process migration. With the help of the Hijacker, users with long-running proprietary programs can now benefit from the checkpointing and migration services offered by distributed resource management systems. They are freed from the requirement to re-link their application in advance and can pass at any time the control over a process that was started interactively to a resource management system. The Hijacker splits the running process into two processes: the original application process augmented with checkpoint and remote system call libraries, and a shadow process that remains in the original execution environment to preserve the I/O state of the application process and to remotely execute its future I/O system calls. The Hijacker employs dynamic program re-writing techniques to insert the checkpoint and remote system call libraries into the application and to replace the standard I/O system calls of the process with remote system calls to the shadow. We have used the Hijacker to dynamically migrate several ordinary programs, including a Java VM. We have shown that process hijacking is a reasonably inexpensive operation, and that the cost of migrating a hijacked process is comparable to the costs incurred by Condor when migrating a pre-linked process.

Acknowledgments

We thank Jim Basney, Todd Tannenbaum, and Derek Wright of the Condor Team for their assistance with Condor, and Bryan Buck, Tia Newhall, Christopher Serra, Ariel Tamches, Brian Wylie, and Zhichen Xu for their assistance with DynInst. Jim Basney and Jin Zhang came up with the idea of restart library relocation for safely restarting checkpoints of hijacked processes.

References

[1] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems* **8**, 2, Spring 1995, pp. 171-216.

[2] Y. Chen, J.S. Plank, and K. Li. CLIP: A Checkpointing Library for Intel Paragon. *SuperComputing* '97, San Jose, CA, 1997.

[3] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience* **21**, 8, August 1991, pp. 757-785.

[4] A.J. Ferrari, S.J. Chapin, and A.S. Grimshaw. Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Technical Report CS-97-05, Department of Computer Science, University of Virginia.

[5] GENIAS Software. Codine. `http://www.genias.de/products/codir`

[6] J.K Hollingsworth and B. Buck. DynInstAPI Programmer's Guide. `http://www.cs.umd.edu/projects/dyni API.`

[7] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report #1346, Computer Sciences Department, University of Wisconsin, April 1997.

[8] M. Livny and R. Raman. High-Throughput Resource Management. **The Grid: Blueprint for a New Computing Infrastructure**, Morgan Kaufmann, 1999, pp. 331-337.

[9] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. **Distributed Computing Systems: Concepts and Structures**, IEEE Computer Society Press, 1992, pp. 145-164.

[10] B. P. Miller, M.D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* **28**, 11, November 1995, pp 37-46.

[11] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. *9th ACM Symposium on Operating System Principles*, October 1983.

[12] P. Smith and N.C. Hutchinson. Heterogeneous Process Migration: The Tui System. *Software Practice and Experience* **28**, 6, May 1998.

[13] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. *10th International Parallel Processing Symposium*, Honolulu, HI, 1996.

[14] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. *2nd European PVM User Group Meeting*, Lyon, France, 1995.

[15] M.M. Theimer, K.A. Lantaz, and D.R. Cheriton. Preemptable Remote Execution Facilities for the V-System. *10th ACM Symposium on Operating System Principles*, Orcas Island, WA, December 1985.