A Survey of Checkpoint/Restart Implementations

Eric Roman
Lawrence Berkeley National Laboratory
Berkeley, CA

Abstract

In this paper we evaluate candidates for a checkpoint/restart implementation against a common set of requirements. Overall characteristics of the two main classes of checkpoint systems, library and system, are discussed followed by specific examples from existing systems. A detailed description of two system implementations is presented. We conclude that no single publically available implementation meets all requirements for a checkpoint/restart system for Linux clusters.

1.Introduction

Checkpoint/restart is a feature found in most platforms used for high performance technical computing. Many technical computing users use the Linux platform; yet Linux does not have a fully featured checkpoint/restart implementation. A number of checkpoint/restart implementations have been developed; no existing implementation is able to meet the full range of requirements for a production system. To identify the state of the art in checkpoint/restart, we conducted a survey of existing implementations. This paper presents the results of this survey.

The rest of this paper is organized as follows. Section 2 describes the motivation for checkpoint/resart, and characterizes the major differences between implementations. Section 3 describes some general requirements and discusses how these requirements are often implemented. Section 4 summarizes much of the relevant work on checkpoint/restart. Section 5 is a detailed study of two checkpoint/resatrt systems. Section 6 lists our conclusions and describes some further work on checkpoint/restart.

2.Background

Checkpoint/restart is an operating system feature that creates a file describing a running process; The operating system can later reconstruct the process from the contents of this file. The checkpoint file contains the stack, heap, and registers of the process. The checkpoint file may also contain the status of pending signlas, signal handlers, accounting records, and terminal state. During restart, the operating system recreates the process and any objects desicrbed in the checkpoint file. This enables the process to continue execution at the point where the checkpoint was taken.

Checkpoint/restart provides two main functions for an operating system. First, checkpoint/restart is a mechanism for fault tolerance. Applications may be checkpointed periodically (or based on notifications from fault monitors). Once the application state has been committed to stable storage, the application may be restarted and reconfigured to work around the fault. Second, checkpoint/restart may be used as a mechanism for preemption. This form of preemption is useful in environments where virtual memory is scarce. In such environments, the operating system is unable to allocate sufficient memory to hold all runnable processes, so checkpoint/restart is used to save the application state to the file system.

Checkpoint/restart may be implemented in three ways: by an application itself, through a library linked with the application, or within the operating system kernel. Application-implemented checkpoint promises the highest efficiency, since the application programmer has exact knowledge of which data structures must be saved, and which may be discarded. Application-implemented checkpoint also has many drawbacks. It may not be possible to change the application source code. This is a major obstacle due to the prevalence of third party scientific applications and old dusty deck applications. Another drawback of application-implemented checkpoint is that the application may place very strong restrictions on when checkpoints may be taken. Commonly, checkpoints may only be taken at the end of a lengthy time step. These restrictions limit the usefulness of checkpoint/restart, since there is a potentially long delay between the time a checkpoint is requested, and the time the application decides to write the checkpoint. Another serious drawback of this technique is the lack of a common restart mechanism. It may be impossible for an automated system to restart an arbitrary application.

Library implementations address some of these deficincies. Libraries may not require source modifications (or even a relink) to the underlying application. Library implementations typically use a signal handler to accomplish checkpointing. This may eleminate time restrictions placed on checkpointability. Library implementations generally have a common restart procedure. Most library implementations look for a special flag passed through the command line to the application. There is one key drawback to using a library implementation: The library imposes restrictions on which system calls the application may use. All forms of interprocess communication (e.g. pipes) are generally forbidden. As a result shell scripts and most parallel applications may not be checkpointed. The restriction against parallel applications is the most severe, since checkpoint/restart has proven most useful for technical computing.

System implementations promise to lift these restrictions. A system implmented checkpoint/restart adds special support to the operating system kernel. Since most data structures are accessible to the kernel there are very few restrictions on the scope of restartable applications. System implementations typically allow applications to be checkpointed at any time. Due to the difficulty of implementitng checkpoint/restart at the system level, system implementations have only been provided on a few current systems.

Process migration is closely related to checkpoint/restart. Process migration and checkpoint/restart must both arrange to save the heap, registers, and stack of a process. Checkpoint restart may be used to implement process migration, and vice versa. When process migration implements checkpoint/restart, the checkpoint is often treated as "migration to disk". When checkpoint/restart implements process migration a shared file system, file transport protocol, or open socket is used to transfer the checkpoint file to a remote node for an immediate restart. There are two large differences between checkpoint/restart and process migration. First, a checkpoint file is usually written so that it is valid after a crash or reboot of the node where the application was started (often called a home node). In contrast, a process migration scheme may leave residual data on the home node, so that the migrated process still requires the home node to be alive. The second major difference concerns the number of instances of the application assumed to be present. Migration schemes may assume the application executes successfully (i.e. without any crashes). A checkpoint/restart scheme must be able to roll back an application to a known state after a crash. It is likely that a long running applications will be rolled back more than once, due to multiple crashes.

3.Motivation

This section introduces the requirements and features of a checkpoint/restart system. We discuss the overall structure and motivation for many of these requirements. We also describe techniques commonly used to implement these requirements. For a complete statement of requirements for a checkpoint/restart system see [REQS02].

Transparency is the parent of most requirements for checkpoint/restart. User applications must be restartable without modifications to the application source code, or requiring the application to be statically linked to an outside library. Applications interact with the kernel through system calls. These system calls modify or create data structures within the kernel. The extent to which a given implementation can recover these data structures determines the range of applications for which perfect transparency is achievable. If checkpoint/restart is not to restrict the applications which may be checkpointable, it must allow the application to use the full range of system calls.

Parallel application checkpoint is the next major requirement. Checkpoint/restart is very important for users of large clusters running parallel scientific applications. These users use checkpoint/restart for either fault tolerance, to implement preemption, or for process migration. There are two distinct classes of parallel application: Multprocess applications and multinode applications. Multiprocess applications use multiple processes executing on a single node that communicate through IPC channels such as shared memory segments, pipes, threads, or local domain sockets. During a checkpoint, the operating system must suspend all processes and save the relevant process state and IPC state. When the application restarts, the kernel reconstructs the processes and the IPC channels. The distinguishing feature of multiprocess applications is that the kernel has consistent access to all process state and IPC state. Since the IPC state is known, no interaction with the processes to be checkpointed or another node is necessary.

Multinode application checkpoint requires the active involvement of the checkpointing processes or coordination with a remote kernel. Instances of the application on separate nodes must cooperate with each eother to ensure that

their saved state is consistent (i.e. All messages known to be received on one node are marked as sent by another node). If communication channels are reliable, this synchronization must also guarantee that all sent messages have been received or buffered, otherwise messages will be lost when the processes are restarted.

Application control over when checkpoints may be taken is desirable. By allowing applications to temporarily block checkpoints during certain regions of code, a broader range of applications are checkpointable. Applications use this mechanism to prevent checkpoints from occurring during calls to unchekpointable libraries. A checkpoint implementation may use a signal handler to accomplish this. The application may simply mask and unmask the checkpoint signal through the signal or sigprocmask system calls.

As an optimization, some checkpoint libraries allow the application to designate "unimportant" data areas. These areas are marked with a system call, and are not saved during a checkpoint. By designating a data area as unimportant, the application spares checkpoint/restart from saving information unnecessary for proper recovery of the application. This optimization is very useful in technical computing, where applications allocate tremendous amounts of memory. This optimization allows checkpoint/restart to ignore large temporary arrays, application caches, or other data structures that may be easily reconstructed (or completely unnecessary) after a restart. By ignoring these sections checkpoint/restart saves disk space and time by not writing unnecessary sections to disk.

CPU register state must be saved during a checkpoint. The CPU state includes the instruction pointer, stack pointer, general purpose registers, floating point registers and any other available registers. A library implemented checkpoint/restart may use a signal handler to capture the CPU registers as follows: When the signal is received, the kernel stores the process registers on the process stack. The signal handler uses a setjmp call to save the stack, and during the restart invokes longjmp to restore it. Therefore, as long as a checkpoint library can restore the stack, the registers are restored automatically. A system implementation simply reads the process registers from the relevant process data structure.

The process address space contains the largest quantity of process state. The address space consists of several sections: The initialized data and unitialized data sections, the heap, the stack, and any mapped regions. A library implementation may obtain the address for the start and end of most regions through system calls and some kernel specific knowledge of how the address space is allocated. A library implementation will have trouble saving mapped regions, since there is no standard way of identifying the addresses, lengths, protection (set through mprotect), or pin status (set through mlock) of mapped regions. It is not possible to access this information by redirecting the mmap system call, because the loader will map regions before the checkpoint library is initialized. The library implementation must access mapped region information using some nonstandard mechanism, such as the /proc filesystem. A system implementation has direct access to the data structures describing the mapped region.

Signal handlers and pending signal state must be saved and restored. The library implementation may obtain signal handler state through the sigaction or signal system calls, since theses system calls return the signal handler state to the caller. The list of pending signals is available through the sigpending system call. A kernel implementation saves the signal handler and pending signal data structures directly.

Most checkpoint/restart implementations do not, or cannot, save and restore process credentials (e.g. The effective UID). This restriction is not terribly severe if checkpoint/restart is only used for user applications and the application does not require any special priveleges. However, if checkpoint/restart is used to checkpoint system daemons or the applications use setuid binaries, this restriction becomes limiting.

Shells are an important class of application for two reasons: First, many applications use shell script wrappers around a compiled binary (e.g. MPICH's mpirun). Second, jobs submitted to batch systems are generally shell scripts. Thus, if checkpoint/restart is to work at the level of batch jobs, it must be able to restart shell scripts and the entire corresponding UNIX session. Checkpoint/restart must save and restore the state of all process groups within the session, as well as restore the PIDs of applications within the session. Session checkpointing has been implemented for system checkpoint/restart on several commercial platforms, but never within a library. Unfortunately, the source for these system implementations is closed, and there are no papers describing the techniques used.

Resource limits and accounting information should be saved and restored. A library implementation may save resource limits by calling getrlimit and restored through setrlimit. The library may save resource usage by saving

the return of the getrusage system call, but there is no corresponding setrusage system call to restore this information. The kernel may simply save and restore the relevant rlimit and rusage data structures directly, so it may properly restore accounting data to a process.

Files and file descriptors pose several challenges for checkpoint/restart. These two abstractions are fundamental to the correct execution of most applications. All input/output paths are built on files and file descriptors. Since these are so ubiquitious, it is critical for checkpoint/restart to handle these correctly. Since file operations occur frequently, performance is an important concern. Files may be modified between a checkpoint and the corresponding restart (possibly by the application itself if it is restarted twice from the same checkpoint, or from a periodic checkpoint after a fault). Worse still, there are no straightforward ways to know what an application has done to the filesystem in the interval between checkpoints. If a file descriptor has been closed, or a file has been unlinked, there are no data structures available to recover the state of the file! Partial solutions to this problem exist, but they involve saving hidden copies of all files when they are opened, and also when the process is checkpointed. Even here, correct execution is impossible to guarantee. No existing checkpoint/restart implementation has adequately addressed the file problem so as to guarantee correct execution under all circumstances. There are a few techniques that may prove practical.

File descriptors are the only link between a process and the corresponding file. File descriptors associated with normal files should be reattatched to those files when the application is restarted. File descriptors associated with a terminal (for standard IO) should be attatched to a the terminal where the restart is requested. Any flags set for the file descriptor, the access mode, and the file offset must be restored after restart. Library implementations resort to system call redirection to maintain a separate copy of the kernel's file descriptor table for this purpose. In some cases, such as read-only access, the file descriptor table may simply be restored during restart. In others, such as general random read/write access, it may be necessary to save the contents of the file with the checkpoint.

Directories may also be accessed through file descriptors. None of the implementations studied in this survey can recover the state of these descriptors. Open directories pose a bit of a problem for checkpoint/restart, since the directory may be modified while the application is checkpointed. Working with other directory information is possible. For example, the current working directory of a process must be saved and restored.

Sockets have always posed a serious problem for checkpoint/restart. Unlike the areas covered so far, a standard approach to socket checkpoint/restart has never emerged, even though a number have been developed. Most implementations choose to ignore sockets entirely. In some cases, the application is given a chance to shut down and restart it's sockets through callbacks run at checkpoint and restart time. Zandy and Miller [ZAND02] use a scheme where a message buffering and a separate control connection are used to maintain a heartbeat signal and recover lost messages after a TCP connection failure. They report coupling their scheme with a checkpoint/restart library to checkpoint MPI applications. Zhong and Nieh [ZHON01] use a scheme where checkpointed sockets are sent into a TCP_TIME_WAIT state to avoid connection shutdown. At restart, they create a new socket and modify the kernel socket data strure to point to the remote end point. The remote socket is then modified, through an out of band mechanism, to point to the newly created socket.

4.Library Implementations

In this section we describe some existing implementations of checkpoint/restart. We discuss kernel checkpoint/restart schemes for Linux, library implementations, and parallel checkpoint/restart schemes. There is some discussion of process migration and commercial checkpoint/restart.

4.1.Library Implementations

libckpt [PLAN95] is one of the first library implementations for UNIX. Libckpt has all the usual limitations of a library checkpoint/restart, but provides a number of special optimizations to reduce the size of checkpoint files. Memory exclusion is a feature that allows the application to note whether a section of its address space is unused or will not be modified. During a checkpoint, unused pages are not written, and pages that will not be modified are only written to the checkpoint file immediately after they are marked. Incremental checkpointing uses the mprotect system call to track modifications to pages inside the address space. Modified pages are marked as dirty; when a checkpoint is taken, only the dirty pages are written to the checkpoint file. Forked checkpointing is a performance optimization that allows the checkpointed process to continue while the checkpoint is being written. When it is

time to checkpoint, libckpt calls fork to make a copy of the process, and the checkpoint proceeds in the forked process, while the original process resumes its computation. Synchronous checkpointing, an optional feature, allows the application to suggest to libckpt that checkpoints be taken at particular points during execution. If a sufficient time has elapsed since the last checkpoint, a checkpoint is taken. Libckpt requires a modification to the application source code. The main routine of the application must be renamed, and the application must be recompiled and statically linked to libckpt. Libckpt can checkpoint and restart applications using shared libraries, but can not restore segments mapped in by the application through mmap.

Condor's checkpoint/restart [LITZ97] implements process migration for the Condor load balancing system. Unlike libckpt, Condor supports applications using memory mapped segments. Condor also supports applications linked with shared libraries. Mapped segments and dynamic libraries are read through the /proc filesystem. Condor uses this information to locate dynamic libraries and mapped segments in the process address space. Condor requires applications to be linked with a special checkpoint library; no recompilation is necessary. This has the advantage that no source modifications are necessary to enable checkpointing, but prevents Condor from checkpointing applications for which object files are unavailable (such as most commercial binaries).

Libckp [WANG95] is one of the only library implementations of checkpoint/restart developend for fault tolerance. Libckp is unique in that it treats the contents of open files as part of the process state. Libckp makes shadow copies of open files, and copies of all files removed with the unlink() system call. Libckp is able to handle a much broader class of file behavior than libraries that simply reposition the file pointer at restart time. Libckp also provides calls, similar to setjmp and longjmp, that allow an application to initiate a checkpoint, or return to a previously checkpointed state. These allow libckp to be used in fault tolerance applications where applications want to explicitly force a roll back when failures occur.

libtckpt [DIET01]checkpoints multithreaded applications using LinuxThreads or Solaris threads. Libtckpt adds a checkpoint thread to the application. During checkpoint and restart operations, this checkpoint thread is used to synchronize the other application threads and invoke user callbacks. The user may install callbacks to be invoked just before a checkpoint is taken, immediately after a checkpoint is taken, and immediately after a restart is performed. By using these callbacks, the application has a chance to save state that libckpt cannot. Libtckpt requires some modification to application source code; the application must include a special header file and call libtckpt's initialization function.

4.2.Parallel Checkpoint/Restart

Score [SCOR00] has a library implementation that checkpoints parallel applications. Score implements checkpoint/restart in the low level communication and run time libraries. This allows Score to checkpoint most parallel applications without any modifications to the application source. Further, very few changes are necessary to high level communications layers, such as MPI. Score checkpoints processes to avoid node failures when running long parallel applications. Score can also store parity data on a remote node. This parity data is used to reconstruct a checkpoint file stored on a failed node, so that applications may be recovered even if a checkpoint file is lost.

CoCheck [PRUY96] implements checkpoint/restart for parallel processes under the Condor checkpoint/restart system. CoCheck uses marker messages piggybacked on top of PVM [PRUY96] or MPI [STEL96] messages to trigger checkpoints. A separate resource manager process is used to help synchronize the network state and facilitate a restart of the parallel application. To avoid network and file server bottlenecks, CoCheck uses special checkpoint servers to store checkpoint files.

5.System Implementations

In this section we will describe the operations of Vmadump and contrast it with CRAK. Both of these systems implement checkpoint/restart for Linux. Since CRAK is similar to vmadump, only the most significant differences will be discussed.

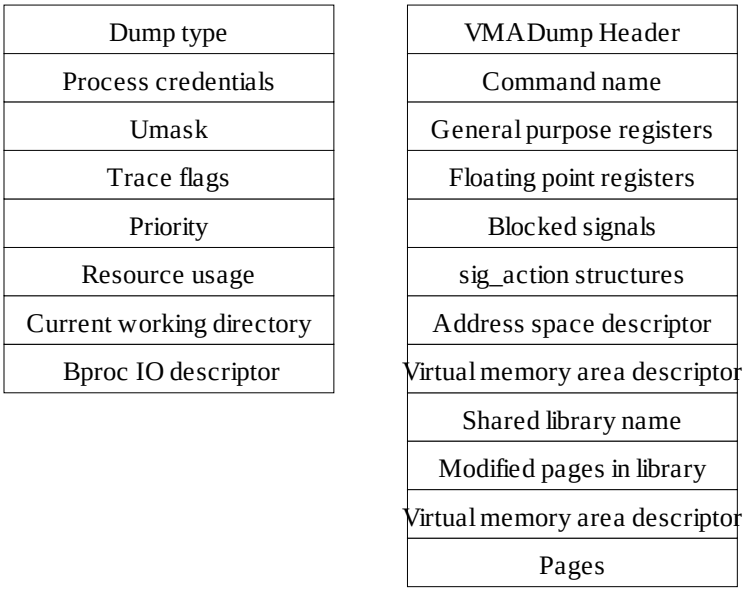
5.1.VMADump

VMADump [HEND00] is a part of Scyld's Bproc system. Bproc presents a cluster-wide PID space to a master

node. Bproc provides operations to spawn processes on remote nodes, migrate processes, and send signals to remote processes. Remote spawn and process migration are implemented through VMADump. VMAdump writes process state to a file descriptor. In Bproc, this file descriptor is a socket connected to a remote node. The other end of this file descriptor reads the process state as it is sent over the socket, and reconstructs the migrating process. VMADump is designed mainly for this style of process migration. However, since VMADump can also be used for checkpoint/restart (simply by passing a descriptor to an open file as an argument), it will be discussed here.

Process migration in bproc is voluntary, or application-initiated. It is not possible to force a process to migrate from one node to another without explicit cooperation from the process. Migration begins when a bproc library function, such as bproc_move, is called to move the process through the global pid space. The bproc library then calls into the kernel, arranges for a migration to the remote node, calls vmadump_freeze_proc to move the process. Upon return from bproc_move, the application is running on a different node.

VMADump may also be invoked directly by an application, through a special system call. This system call takes a file descriptor as an argument, and writes the state of the calling process to the file descriptor. A similar system call may be used to read the process image back from a file, and restore the process. This system call is similar to exec (), in that if it is successful, it will never return to the caller. VMADump also allows process images to be executed directly through exec.



VMADump does not write all of the process state to the file descriptor. Much of the process state is not transferred at all, and some may be handled by another part of Bproc. Bproc sends some state immediately before invoking VMADump. Bproc first sends the process credentials. The credentials correspond directly to fields in the task structure (task_struct) describing the process. These fields describe the current, effective, saved, and filesystem UIDs and GIDs for the process. Also present are group membership, process capabilities, thread identifiers, and whether the process is dumpable. The umask is sent next, followed by ptrace status, priority information, rlimits, resource usage, and the current working directory. Bproc then sends a data structure that allows it to reconnect the standard I/O stream.

Once this information has been sent, the remainder of the process is sent through the socket. There are four main sections, a header, CPU state, signals, and memory. The header contains information used for simple sanity checking. The process receive code scans the header to make sure that the architecture, format versions, and kernel

versions match on both sides. If there is a mismatch, the restore is aborted.

The registers are saved next. Since Linux saves the general purpose registers in a pt_regs structure on the kernel stack, this structure is easily accessible. Floating point and debugging registers are saved on a structure of type thread_struct, which is a member of the task structure. Some care must be taken when restoring the general registers, since a malicious user may tamper with the registers described in the file to change registers that are ordinarily not modifiable by applications.

Signal handlers and blocked signals are then saved. The set of blocked signals is available from the blocked field of the current process descriptor. Since this field is a bitmap, it is quite simple to save and restore. Current signal handlers are stored as k_sigaction structures. These structures contain no references outside the process space, so the data structures may be saved directly to disk. When manipulating signal data structures, VMADump obtains the necessary locks before modifying these structures. When restored, these data structures are read back from disk, and attatched to the process structure. A check is made to ensure that the user cannot override default behavior for SIGSTOP and SIGKILL, which normally cannot be modified by user processes.

Memory is saved last. While an executable is loading, the loader maps shared libraries directly into the process address space. Due to the use of shared libraries, even trivial processes can consume large amounts of memory. As an optimization, VMADump sends the name of the shared library to the remote node, rather than the contents of the shared library. On the restarting node, the shared library is mapped back into the process' memory, thus avoiding the direct transfer of the shared library contents. Any pages modified through copy-on-write are transferred next. If the region is not backed by a file, the all nonzero pages are transferred directly.

VMADump meets only some of our requirements for checkpoint/restart. Since VMADump is application-initiated, it is not sufficiently transparent to meet our requirements. VMADump ignores file contents and file descriptors, so restarting processes will not continue correct execution in most cases, unless it arranges to reopen its files. VMADump can checkpoint only individual single-threaded processes, not sessions, process groups, or multithreaded applications, since it lacks any synchronization mechanism.

5.2.CRAK

CRAK [ZHON01] is a kernel module that implements checkpoint/restart for Linux. CRAK is designed for process migration. CRAK has two main design goals. First, CRAK implements checkpoint/restart as a kernel module. Second, CRAK reconnects checkpointed sockets to their remote end points. CRAK seeks to implement a checkpoint/restart that requires no modifications to existing code, and minimal modifications to the operating system kernel. CRAK can also recreate IPC mechanisms (pipes) between processes after a checkpoint.

CRAK's checkpoint/restart is split between user space and kernel space. User space is responsible for identifying the set of processes to be checkpointed, and for reconnecting open file descriptors and pipes. CRAK allows the user three ways to specify which processes to checkpoint, a single process may be checkpointed, all children of a process may be checkpointed, or a process and all of its children may be checkpointed. After identifying the set of processes to be checkpointed, these processes are sent a SIGSTOP signal. Since all processes are suspended, they should form a consistent state. Once the processes are suspended, the kernel code is invoked to write the checkpoint files. Once the checkpoint files are written, CRAK sends the user processes a SIGCONT signal to allow them to continue, or optionally kill the processes.

CRAK saves process credentials, virtual memory, signal handlers and pending signals, and the current working directory in a manner quite similar to VMADump, with only a few major exceptions. When VMADump is called, it is called by the process to be checkpointed, so the kernel may use the current pointer to access the relevant task structure. The page tables of the current (checkpointing) process are also valid. In CRAK's checkpoint/restart, the kernel executes as the process requesting the checkpoint, not the checkpointed process. This means that CRAK must walk the task list to find the process descriptor corresponding to a checkpointing process. Since the page tables do not reflect the checkpointing process, CRAK must manually walk the page tables of the checkpointing process to find the location of a checkpointing process' memory.

CRAK can save file descriptors attatched to sockets, unnamed pipes, and regular files. CRAK recognizes duplicate file descriptors and arranges for them to be saved and restored. CRAK's support for regular files is similar to the library implementations. During a checkpoint, the file name, access mode, flags, and position are saved, and during

a restart the file is reopened, and the relevant fields are restored. Pipes between processes are reconnected in user space. Any data undelivered in pipes is restored in kernel space. Sockets are restored in three phases. First, a new socket is created in user space. Next, when the process is in kernel space, the local socket data structure is modified to reflect the original endpoint. Finally, the remote socket data structure is modified to reflect the restarting address.

CRAK meets most of our requirements for checkpoint/restart. CRAK is system-initiated, so no modifications are necessary to user code. CRAK can restore file descriptors. CRAK can also checkpoint multiple processes. CRAK cannot restart multithreaded processes, since it does not understand shared virtual memory areas. Also, CRAK does not allow the user to register checkpoint handlers, or block checkpoints from occurring. While CRAK may work fine for some applications, it is far from a general purpose checkpoint/restart.

6.Conclusions

6.1.Summary

Table I summarizes the work to date on checkpoint/restart implementations. While signal handlers, open file descriptors, the process address space, and registers are fairly well covered by most implementations, most other areas are not. Checkpoint handlers are supported only by very modern implementations. A truly complete implementation of checkpoint/restart would address all of them.

Name	***Type***	***Scope***	***File Data***	***Resource Usage***	***Credentials***	***Checkpoint Handlers***	***Signals***	***File Descriptors***	***Address Space***	***Registers***
libckp	lib	Process	-	-	-	-	-	❍	❍	●
libckpt	lib	Process	❍	-	-	-	-	❍	❍	●
Condor	lib	Process	-	-	-	Δ	●	❍	❍	●
libtckpt	lib	Thread	-	-	-	❍	●	❍	❍	●
CRAK	sys	Child	-	-	❍	Δ	●	❍	❍	●
BPRoc	sys	Process	-	❍	Δ	-	●	-	❍	●
Score	lib	Parallel	-	-	-	❍	●	❍	❍	●
CoCheck	lib	Parallel	-	-	-	Δ	●	❍	❍	●
- = Missing. Δ = Weak. ❍ = Good. ● = Complete										

6.2.Other Work

SGI has implemented checkpoint/restart for the IRIX operating system. The IRIX checkpoint/restart is closed source, and part of the IRIX kernel. Cray has implemented checkpoint/restart for the Unicos and Unicos/mk operating systems. The Unicos/mk implementation is still in use at some production computing centers. Finally, IBM is developing checkpoint/restart for the AIX operating system. The IBM checkpoint/restart is not complete. The Cray and SGI implementations are widely believed to be the most robust implementations in existence. We are unable to provide more information on these systems since there are no publically available documents describing their implementation.

MOSIX and Compaq's SSI project can both migrate processes between nodes. Unlike Bproc, where process migration is loosely coupled to the rest of the run time, MOSIX and SSI process migration is very closely coupled to the runtime. In MOSIX, stub processes are left on the original node, and system calls are forwarded back to the original node. In SSI, stub information is kept on the original node for open files and sockets, but for no other process state. Neither of these systems have a process migration that may be used for a general purpose checkpoint/restart, so they were not included in this survey.

Carothers and Szymanski [CARO02] describe a system-implemented checkpoint for multithreaded programs

running on Linux. Under their scheme a new system call, sys_checkpoint, is introduced. sys_checkpoint creates a copy of a multithreaded application using a copy-on-write mechanism similar to that used by the fork system call. The new processes remain in memory, so checkpoint files do not need to be written to disk. Their scheme has the advantage that it can be quite fast, but since checkpoint files are not written, checkpoints may not persist through a reboot of the node, or be migrated to a remote node. Though they have implemented a checkpoint system call, there is no corresponding restart system call. While this implementation is potentially of use to those needing fast checkpoints to protect against process failures, due to the discussed restrictions this implementation does not meet our basic requirements for a checkpoint/restart facility.

Epckpt implements kernel level process migration for Linux. Epckpt is used in GNU Queue for checkpoint/restart, and was used in the Linux Kernel Hot Swap Project. Since CRAK is derived from Epckpt, we chose not to discuss it in this survey. StarFish implements checkpoint/restart for programs running under the StarFish runtime system. StarFish can only checkpoint applications written in Ocaml. Further, StarFish requires the user to perform any synchronization between processes required for checkpoint/restart; i.e. StarFish does not ensure that checkpoints are consistent.

6.3.Future Work

We plan to implement a generic checkpoint/restart facility for the Linux operating system. This study was conducted to understand the state of practice in checkpointing, and identify code or results that we could apply to our work. We will have full support for multithreaded processes, sessions, and parallel processes communicating through MPI.

Bibliography

[REQS02] Duell, J., Hargrove, P., and Roman, E.. Requirements for Linux Checkpoint/Restart,

[ZAND02] Victor C. Zandy and Barton P. Miller. Reliable Network Connections. ACM MobiCom 2002, Atlanta. September, 2002.

[ZHON01] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint / Restart As a Kernel Module. Technical Report CUCS-014-01. Department of Computer Science. Columbia University, November 2002

[PLAN95] J. S. Plank, M. Beck, G. Kingsley, and K. Li.. Libckpt: Transparent Checkpointing Under UNIX. Conference Proceedings, Usenix Winter 1995. Technical Conference, pages 213-223. January 1995.

[LITZ97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed System. http://www.cs.wisc.edu/condor/doc/ckpt97.ps. .

[WANG95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. Proceedings of the International Symposium on. Fault-Tolerant Computing. pages 22-31, June 1995.

[DIET01] William R. Dieter, and James E. Lumpp, Jr.. User-level Checkpointing for LinuxThreads Programs. FREENIX Track: USENIX 2001 Annual Technical. Conference. pp. 81-92. June, 2001

[SCOR00] T. Takahashi, S. Sumimoto, A. Hori, H. Harada and Y. Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network. Environments. http://www.sc2000.org/techpapr/papers/pap.pap205. .pdf. 2000

[PRUY96] Jim Pruyne and Miron Livny. Managing Checkpoints for Parallel Programs. Job Scheduling Strategies for Parallel Processing. IPPS 96 Workshop v. 1162 pp. 140-154. 1996

[STEL96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. Proceedings of the 10th International Parallel. Processing Consortium (IPPS 96). 1996

[HEND00] Erik Hendriks. VMADump. http://bproc.sourceforge.net. . 2002

[CARO02] Christopher D. Carothers and Boleslow K. Szymanski. Checkpointing Multithreaded Programs. Dr. Dobb's Journal. August 2002.