Checkpointing in CosMiC: a User-level Process Migration Environment

P. Emerald Chung, Yennun Huang and Shalini Yajnik
Bell Labs, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{emerald,yen,shalini}@bell-labs.com

Glenn Fowler, Kiem-Phong Vo and Yi-Min Wang
AT&T Labs, Research
180 Park Avenue
Florham Park, NJ 07932
{gsf,kpv,ymwang}@research.att.com

Abstract

The CosMiC *system is a user-level process migration environment. Process migration is defined as the mechanism to checkpoint the state of an unfinished process, transfer the state from one machine to another, and resume process execution on the new machine. The main purposes of process migration are (1) to utilize the CPU power and balance load on all machines in an environment; (2) to provide fault-tolerance by migrating a process from a failed machine to another machine.*

CosMiC *provides an extensible architecture to allow an application to choose its own checkpointing mechanism. It is equipped with four checkpoint libraries, namely,* `libckp`, `libfcp`, `libft` *and* `libst`. *They provide different strategies for state saving and restoring.* `Libckp` *is a transparent checkpoint library, it checkpoints the entire process state. It requires minimum user involvement and no modifications to the source code.* `Libfcp` *is a file checkpoint library that saves and restores file contents.* `Libft` *is a critical data checkpoint library. Users select critical data to be checkpointed using a set of application programming interfaces (APIs).* `Libst` *is a strong-type checkpoint library. It saves and restores architecture-independent checkpoints in a heterogeneous environment. In this paper, we describe our experience in incorporating these different checkpointing mechanisms into the* CosMiC *system.*

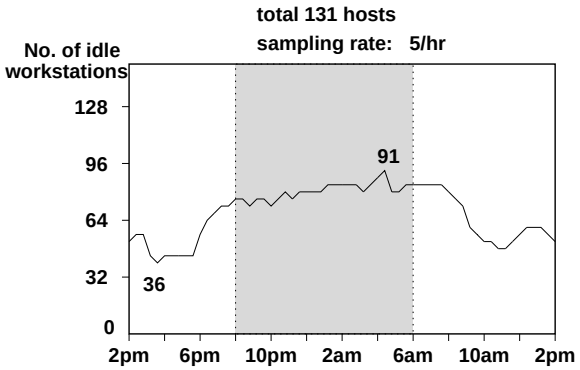

Figure 1. Numbers of idle machines in a day

1. Introduction

In the scientific computing community, many tasks require a large amount of computation time. Examples include model parameter estimations from a large set of data and solving a large-dimensional linear equation.

These jobs used to be carried out on centralized supercomputers. With the trend of desktop computing, many modern laboratory computing environments are now populated with workstations sitting on individuals' desks and sharing network file systems. There is a need to run the computation-intensive jobs effectively in this type of distributed environments.

It has been observed that the CPU power in a workstation environment is usually under-utilized. We did a measurement in a local computing environment, which consists of 131 SUN and SGI workstations, and found that, on average, 1/3 of the machines were idle during the days and 2/3 of

them were idle during the nights and weekends. Figure 1 shows a typical day. The situation was disturbing because many users were asking for more computing power while a significant percentage of machines were idling.

To take advantage of these under-utilized computer cycles, many systems have been developed to distribute jobs automatically and to balance the load on all the machines. A typical setting includes a centralized or distributed job queue, and a resource monitor running on each machine. Resource monitors periodically collect status information such as load, idle time, or number of users. The job queue dispatches a task to a certain machine based on some load-balancing algorithm.

An important feature in these systems is *process migration* [13], i.e., the capability to transfer the state of an unfinished process from one node to another node, and to resume process execution. With process migration, a system is able to move jobs from an overloaded node to an under-loaded one, to move jobs away from a node being interactively used by users, or to recover jobs which fail to finish due to a node failure.

The techniques for process migration can be categorized into *OS-built-in process migration* and *user-level process migration*. OS-built-in process migration has been implemented in several operating systems. Examples include MOSIX [3], V [5], Accent [16], Sprite [14], Mach [2] and OSF/1 AD [22]. The user-level process migration is typically implemented as checkpoint-restart systems running on UNIX or NT operating systems. Examples include Condor [12] and Load Sharing Facility (LSF) [23]. Although the OS-built-in migration systems are often more efficient, they have not been widely used due to the slowness in accepting new operating systems in the user community.

There are also parallel systems that utilize the computation power from a network of workstations, such as *Linda-Piranha* [9] and *PVM* [7]. Also, the *CMU Dome* project [17] implements high-level architecture checkpoint to support heterogeneous process migration in *PVM*. Unlike the previous migration mechanisms which are applied to general UNIX processes, these systems mainly provide services to parallel programs written in specific programming paradigms.

We developed a user-level process migration system, called CosMiC, to address the needs from a signal processing research group. In this paper, we focus our discussions on the checkpoint-restart mechanisms supported in the CosMiC system, and our experience in building and deploying the system.

2. CosMiC System Architecture

CosMiC stands for **Cos**hell [6], **Mi**gration and **C**heckpointing. The overall architecture of CosMiC is shown in Fig. 2. We briefly describe the Coshell service and the migration mechanism in this section. The checkpoint-restart techniques are described in more details in the next section.

2.1. Coshell Service

Coshell is a general purpose service that automatically executes jobs on lightly loaded machines in a local network. There is no centralized job queue; each user is served by a separate Coshell daemon with its own job queue. A Coshell daemon uses `rsh` to establish a shell process on a remote machine, without requiring any special privilege. To minimize the overhead of establishing network shell connections, a Coshell daemon may reuse a previous connection.

Coshell daemons are responsible for accepting user requests for job submissions and status inquiries, and for sending jobs to idle machines in the network. Each Coshell daemon maintains and schedules its own job queue. The selection of a target machine is based on a ranking function that combines both *static machine attributes* and *dynamic status information*, to be described shortly. The target machine is selected randomly from a number of top-ranked machines to reduce the chance of multiple Coshell daemons scheduling jobs on the same idle machine.

Static machine attributes are maintained in a central host description file as shown in Fig. 3. The `type` attribute differentiates different processor types; `rating` specifies the relative mips rating; `idle` is the minimum idle time of any interactive users before jobs can be scheduled on the machine (the default is 15 minutes); `mem` describes the size (in Mbytes) of the main memory; `puser` defines the primary user(s) of the machine. Typically, if a machine is physically located in a person's office, the `puser` attribute would include that person. In addition to these standard attributes, users can also add their own attributes for machine selections.

Dynamic status information is collected by a per-machine *status daemon*. Each status daemon periodically posts status to a per-machine status file visible to all machines in the local network. Status update frequency is configurable and the default value is 40 seconds plus a random fraction of 10 seconds in our installation. The random component tends to evenly distribute the update times of all status daemons.

A machine is considered idle if all of the following three conditions are satisfied: (1) the one-minute load average is less than a threshold, with a default value of 0.5; (2) the time elapsed since the last interactive usage of any primary users is longer than the time specified by he `idle` attribute; and (3) no nontrivial jobs are being run by any primary user. A nontrivial job is determined as follows. The process status is checked every minute by the UNIX command `ps`. If the status of a primary user's process is *running* in two

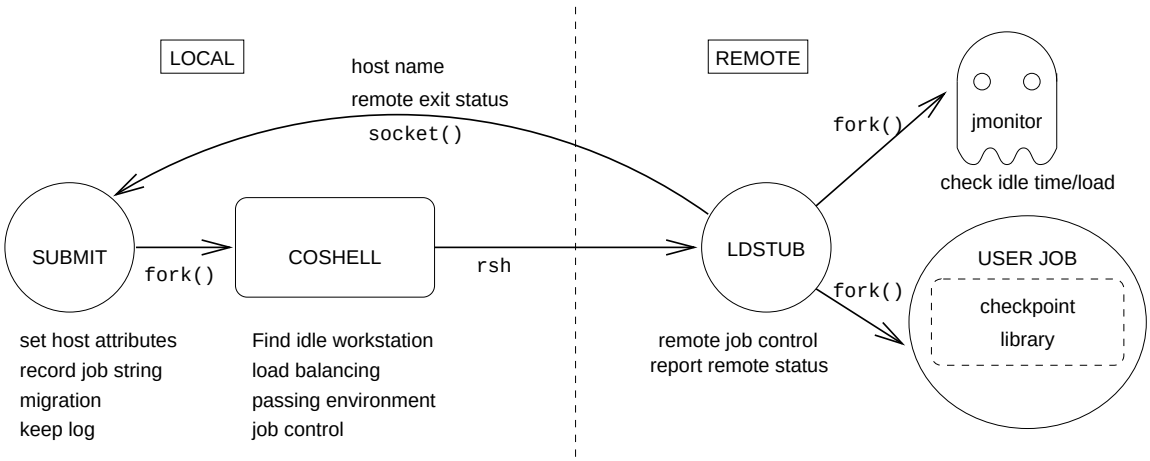

Figure 2. CosMiC architecture

```
banana      type=sgi.mips   rating=60  idle=15m mem=64m puser=emerald
orange      type=sun4       rating=9   idle=15m mem=32m puser=ruby
grape       type=sun4       rating=18  idle=15m mem=32m puser=gem
strawberry  type=sol.sun4   rating=4   idle=15m mem=64m puser=stone
kiwi        type=hp.pa      rating=40  idle=15m mem=64m puser=amber
pear        type=sun4       rating=21  idle=15m mem=96m puser=diamond
apple       type=att.i386   rating=5   idle=15m mem=96m puser=sapphire
```

Figure 3. An example host description file

consecutive polling, we consider that process as a nontrivial job.

2.2. Migration mechanism

Three programs, `submit`, `ldstub` and `jmonitor`, together handle job migration.

`Submit` is an interface between users and Coshell. User jobs can be commands, application programs, or shell scripts. Each job is specified by a shell command string. `Submit` passes the job string to Coshell, which then selects an idle machine and starts an `ldstub` on that machine to execute the job, as shown in Fig. 2.

On the selected machine, `ldstub` forks both the user job and a `jmonitor`. `Jmonitor` is a fine-grain polling device for detecting changes in machine status. It does the polling every 10 seconds. Migration is triggered by an occurrence of any of the following events: (1) an activity on an input device by a primary user, either directly or through remote login; (2) a nontrivial job being run by a primary user; (3) a one-minute load average being greater than or equal to a particular threshold, typically 3.5. Upon detecting any such event, `jmonitor` exits. Since `jmonitor` is a child process of `ldstub`, a `SIGCHLD` signal is sent to `ldstub`. Being interrupted by the signal, `ldstub` kills the user job, sends a job migration request to `submit`, and then exits. `Submit` simply resends the request to the Coshell daemon to restart the job on another machine. If the job is not linked with a checkpoint library, it is re-executed from the initial state.

If a job finishes without `jmonitor` exiting, `ldstub` sends its exit status back to `submit` and `submit` exits with the same status. The job will not be further submitted.

Occasionally, the selected machine crashes, and the job is lost. `Submit` detects such a failure when the communication channel, a UNIX socket, between `submit` and `ldstub` is closed. In such case, `submit` resends the job to `Coshell` to run on another machine. This mechanism provides fault tolerance to remote machine failures.

3. Checkpointing and Restart Techniques

In our design, the job control mechanism is responsible for starting, killing and restarting the *same* UNIX command until the job exits by itself. The checkpointing mechanism is orthogonal to the job control mechanism, and is under the full control of the user program with assistance from the checkpoint libraries. A user program linked with a check-

point library distinguishes a recovery from a fresh start by detecting the existence of an appropriate checkpoint file. That is, if the checkpoint file exists, then it is a recovery; otherwise, it is a fresh start. This design philosophy is different from Condor and LSF, which typically require extra command-line arguments when a migrated process is recovered from a checkpoint. Any application must be linked with a standard checkpoint library so that these extra arguments can be properly interpreted. In contrast, CosMiC recognizes the fact that different applications may need different kinds of checkpointing and restart techniques. Therefore, it provides an extensible architecture to allow each application to choose its own checkpointing mechanism. In this section, we describe four different types of checkpoints that are currently available for use with the CosMiC system.

3.1. User-transparent checkpoint library: `libckp`

`Libckp` is a user-transparent checkpoint library for Unix applications [20]. It can be linked with a user program to periodically save the program state on stable storage without requiring any modifications to the source code. The checkpointed program state includes (1) program counter, (2) program stack and stack pointer, (3) open file descriptors, (4) global and static variables of the user program, the statically linked libraries, and the dynamically linked libraries, (5) dynamically allocated memory, and (6) the `mmap` regions. `Libckp` takes over the `main` routine of the user program in order to perform necessary initializati on and to detect a recovery mode. Inside `main`, `libckp` uses a probing technique to identify the data areas of dynamically linked libraries, that need to be included in the checkpoint. If there exists an appropriate checkpoint file, `libckp` enters the recovery mode by restoring all checkpointed state, and the program eventually jumps to where the checkpoint was taken; otherwise, `libckp` invokes the original `main` routine supplied by the user program to do a fresh start.

The main advantage of using `libckp` is that it does not require any modifications to the user program source code. The main disadvantage is that, since the entire process state is checkpointed, the checkpoint file size can be large and the run-time overhead can be high. We have tested `libckp` on a dozen of signal processing, simulation and CAD applications, and measured the run-time overhead to range from zero to seven percent with a default 30-minute interval between checkpoints. If an application cannot tolerate this level of overhead, `libckp` can be modified to use a standard copy-on-write technique [15]: instead of blocking the process execution when taking a checkpoint, a child process is spawned to perform the actual checkpoint task while the parent process continues normal execution. It is noted, however, that the copy-on-write technique may not be effective if the memory usage of the user program approaches the limit of the available physical memory.

In addition to the default totally transparent mode, we have observed that the following two semi-transparent modes of `libckp` are also useful in practice. First, `libckp` provides a `ckpcheckpoint()` function call that can be inserted into a user program to snapshot the entire process state at a user-controlled program location. This can be useful for debugging purpose, or to avoid losing all useful work when a program abnormally exits due to temporary resource unavailability or glitch [20]. Second, `libckp` allows users to hook in an application-specific recovery routine after data restoration but before the final program jump. This is useful for reestablishing whatever execution environment that is difficult to checkpoint, and for correcting state variables that should not have been checkpointed and restored in the first place. For example, if the program uses a variable to hold the current host name, the restored value upon a migration will be the old host name, which may cause subsequent execution to be incorrect. In this case, the user program can obtain the new host name and overwrite the restored value inside the application-specific routine to ensure correct recovery. As another example, whether a timing-related variable should be checkpointed or not is usually application-dependent. Variables holding "remaining time to timeout" information often need to be recalculated upon a recovery, instead of being restored to the checkpointed values.

3.2. File checkpoint library: `libfcp`

In addition to volatile memory state, a user program typically also modifies files during execution. To ensure consistency, all file updates since the previous checkpoint should also be rolled back when the memory state is rolled back. This functionality was first provided as part of the `libckp` implementation. Since checkpointing files is essentially orthogonal to checkpointing volatile memory and because of its practical importance, this file-rollback functionality is now taken out of `libckp` and provided by a separate checkpoint library `libfcp` [19]. This allows the flexibility of using `libfcp` with any other volatile memory checkpoint library. `Libfcp` can also be used alone in the following two application scenarios. First, if an application's critical data always all reside in files at checkpoint time, volatile memory would never need to be checkpointed. Second, if an application is not long-running and so volatile memory checkpoints are not necessary, the application may still need to use `libfcp` to ensure file consistency at restart time.

Since it is impractical to snapshot all the files associated with an application at checkpoint time, `libfcp` uses an *in-place update with undo logs* approach to checkpoint files. It intercepts all file operations except for read-only ones.

When a file is opened for modifications, its size is recorded and an undo log of file truncation is generated. When the portion of the file that existed at the previous checkpoint time is about to be modified, an undo log of restoring the pre-modification data is generated. When a rollback occurs, these undo logs are applied in a reversed order to restore the original files. It is noted that, when `libfcp` is used with an volatile memory checkpoint library, a simple linear two-phase commit protocol should be applied; otherwise, it is possible that a failure occurring during checkpoint time can cause inconsistency between the memory checkpoint and the file checkpoint.

3.3. Critical data checkpoint library: `libft`

Many long-running applications have the following program structure: they start with some initialization procedures and then enter a program loop executing a large number of iterations. Within each iteration, the stack may grow due to subroutine calls, and dynamic memory may be allocated to hold intermediate results. But at the end of each iteration, the stack often shrinks back to containing only the main stack frame, and most intermediate dynamic memory has been deallocated. Critical data that are needed for later iterations usually concentrate in a small number of well-defined regions or data structures such as arrays and matrices. Although `libckp` can still be used in this type of applications, saving the entire process state at arbitrary program locations often generates unnecessarily large checkpoint files. A more efficient approach is to use non-transparent checkpointing: checkpoint routine is always invoked at the loop boundary, and only user-specified critical data are saved.

`Libft` [10] provides a set of functions that users insert into their programs to specify, checkpoint, and restore critical volatile data. The critical data may include variables and dynamically allocated memory, but not data on the stack. In contrast with `libckp`-style recovery which does a `longjmp()` to a checkpointed program location to skip all executions before that, `libft`-style recovery runs through all original initialization procedures, and then invokes the `recover()` function to restore checkpointed data before entering the loop. An advantage is that variables that hold environment-dependent information such as host names can be excluded from the critical data. They are reinitialized to the correct current values in the restarted program, and are not overwritten by the `recover()` function.

Sometimes it may not be an easy task to correctly identify all critical data. For example, if a program imports software libraries written by others, it is often safer to consider all data used by these libraries as critical. To use `libft` to checkpoint such applications, the `all_critical()` function is first invoked to declare all global variables and dynamically allocated memory as critical. (Note that stack variables are not included.) Then, the `uncritical()` function is invoked to exclude from the checkpoint all data regions of the program, that are known to be uncritical.

`Libft` also provides a file checkpoint mechanism. It is different from `libfcp` in the following two aspects. First, it provides a set of APIs and so is not user-transparent. Second, it adopts a *deferred update* approach, as opposed to in-place update: if a file is opened with the `ftfopen()` call, all file updates are buffered in a log file, and are applied and committed only at `ftfclose()` or `ftcommit()`. A call to `ftabort()` or a migration rollback simply causes the log file to be removed and all updates to be aborted.

3.4. Strong-type checkpoint libary: `libst`

Coshell supports execution on a heterogeneous environment by defining a `HOSTTYPE` environment variable. A user's environment variable which contains any occurrence of */hosttype/* in it is changed to `/$HOSTTYPE/` in the remote job context. For instance, let the local host type be `sun4` and `PATH=/home/emerald/arch/sun4/bin:/bin`, if the remote $HOSTTYPE is `sgi.mips` then the remote shell would use `PATH=/home/emerald/arch/sgi.mips/bin:/bin` to find the proper executable.

However, both `libckp` and `libft` checkpoint and restore data in terms of memory addresses. Since machines of different architectures use different representation formats for various data types, checkpoints saved by these two libraries in general cannot be used to recover the same process on another machine of a different architecture.

To make CosMiC process migration in a heterogeneous environment possible, we developed another checkpoint library `libst` [21] to take checkpoints in a machine-independent format. A user of `libst` first specifies the critical data structures of the program in a specification file. A protocol compiler, `stgen`, then parses the specification file to generate specific checkpoint routines for the program. When a checkpoint is needed, the program calls `libst` checkpoint functions which in turn invoke the generated checkpoint routines. The checkpoint routines traverse the critical data structures and marshal the data in the XDR format. The marshaled data can then be saved into a file or to another process. Because the XDR format is machine independent, the checkpointed data can be used on another machine of a different architecture. During the recovery, the checkpointed data is read from the file and unmarshaled into relevant data structures.

Compared to `libckp` and `libft`, `libst` requires more efforts from the programmers to write the specification file and to generate checkpoint routines at compile time. It may also incur larger run-time overhead due to data marshaling and unmarshaling, if an application's critical data structures are complex and large. Therefore, `libst` is usually used in CosMiC only when process migration in heterogeneous environment is needed.

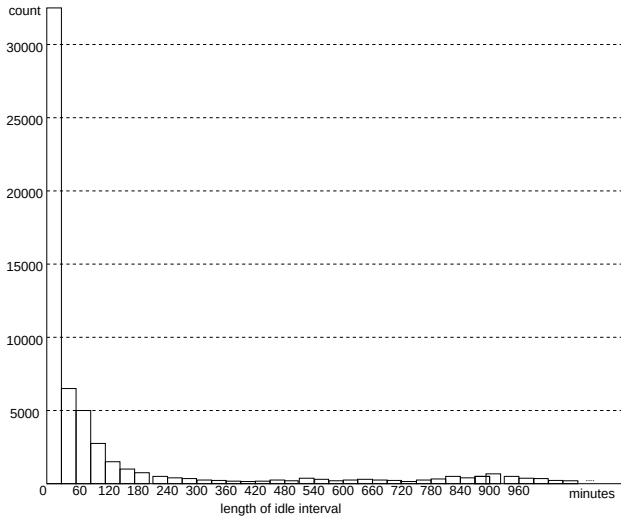

Figure 4. Histogram of idle intervals for 131 machines over a period of four months

3.5. Other checkpoint-related issues

In all of the above checkpoint libraries, the checkpoints are often stored as files, identified by UNIX filenames. In our environment, all the machines share a network file system and so any checkpoint file can be accessed from all the machines by a uniform name. We do not consider migration environments where files may not be accessible from a remote machine. The Condor system solves this problem by remote system calls, which always open a file from the home machine and send the file content to the selected machine.

We have collected the status information from the 131 machines for a period of four months, including a total of 64,082 idle intervals. The distribution of these intervals is shown in Fig. 4. If we model the distribution as an exponential distribution, the estimated expected idle time θ, which is the average length of all idle intervals, is about 70 minutes. (Note that in our calculation, we did not include intervals longer than 8 hours. They occurred in evenings, weekends or holidays.) This information can be useful for calculating optimal checkpoint intervals. If the expected execution time for a job is known in advance and the checkpoint size can be estimated, several algorithms have been proposed to calculate the optimal checkpoint interval to minimize the execution time [8, 11, 18].

4. Summary and Future Work

We have presented the architecture of the CosMiC system, and described four different checkpoint-restart techniques that can be used with CosMiC. So far, most usage of CosMiC is for computing model parameters in speech or pattern recognition applications. Such applications are typically self-contained, single-process C or C++ programs. Future work includes checkpointing script applications, transparently migrating inter-process communications for message-passing applications, and integrating the migration techniques with emerging distributed object standards such as DCOM [4] and CORBA [1].

References

[1] *The Common Object Request Broker: Architecture and Specification*. July 1995. http://www.omg.org/corba/corbiiop.htm.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *the 1986 summer USENIX Conference*, 1986.

[3] A. Barak and A. Litman. Mos: a multicomputer distributed operating system. *Software-Practice and Experience*, 15(8), 1985.

[4] N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt, Nov 1996.

[5] D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3), 1988.

[6] G. S. Fowler. The shell as a service. In *Cincinnati USENIX Conference Proceedings*, June 1993.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Dec. 1994.

[8] E. Gelenbe. On the optimal checkpoint interval. *JACM*, 26(2):259–270, April 1979.

[9] D. Gelernter, M. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. Technical Report 983, Department of Computer Science, Yale University, Sept. 1993.

[10] Y. Huang and C. Kintala. A software fault tolerance platform. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*. John Wiley & Sons, 1995.

[11] C. M. Krishna, K. G. Shin, and Y. H. Lee. Optimization criteria for checkpoint placement. *CACM*, 27:1008–1012, 1984.

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

[13] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. http://www.osf.org/ dejan/papers/m8.6.fr.ps.Z, submitted for publication, 1996.

[14] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Transactions on Computers*, Feb. 1988.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *the 1995 Usenix Technical Conference*, pages 213–224, Jan. 1995.

[16] R. Rashid and G. Robertson. Accent: a communication oriented network operating system kernel. In *the 8th Symp.on Operating System Principles*, pages 64–75, 1981.

[17] E. Seligman and A. Beguelin. High-level fault tolerance in distributed programs. Technical Report 983, Department of Computer Science, Carnegie Mellon University, Dec. 1994.

[18] K. G. Shin, T. Lin, and Y. Lee. Optimal checkpointing of real-time tasks. *IEEE Trans. Comput.*, C-36(11):1328–1341, Nov. 1987.

[19] Y.-M. Wang, P. E. Chung, Y. Huang, and M. Elnozahy. Integrating checkpointing with transaction processing. In *Proceeding of 27rd Fault-Tolerant Symposium*, Seattle, Washington, June 1997.

[20] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *the 25th Intl. Symp.on Fault-Tolerant Computing*, pages 22–31, June 1995.

[21] S. Yajnik and Y. Huang. STL: A toolkit for on-line software update and rejuvenation. submitted for publication, 1996.

[22] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An osf/1 unix for massively parallel muticomputers. In *the 1993 winter USENIX Conference*, 1993.

[23] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software Practice and Experience*, Dec. 1994.