in *Proc. IEEE Fault-Tolerant Computing Symp. (FTCS-25)*, pp. 22-31, June 1995.

Checkpointing and Its Applications

Yi-Min Wang, *Yennun Huang*, *Kiem-Phong Vo*, *Pi-Yu Chung* and *Chandra Kintala*

Abstract

This paper describes our experience with the implementation and applications of the Unix checkpointing library `libckp`, and identifies two concepts that have proven to be the key to making checkpointing a powerful tool. First, including all persistent state, i.e., user files, as part of the process state that can be checkpointed and recovered provides a truly transparent and consistent rollback. Second, excluding part of the persistent state from the process state allows user programs to process future inputs from a desirable state, which leads to interesting new applications of checkpointing. We use real-life examples to demonstrate the use of `libckp` for bypassing premature software exits, for fast initialization and for memory rejuvenation.

1 Introduction

Checkpointing and recovery is a technique for saving process state during normal execution and restoring the saved state after a failure to reduce the amount of lost work. Since it is often not possible to checkpoint everything that can affect the program behavior, it is essential to identify what is included in a checkpoint in order to guarantee a successful recovery. Figure 1(a) shows the three components which together determine the program behavior. **Volatile state** consists of the program stack and the static and dynamic data segments[2]. **Persistent state** includes all the user files that are related to the current program execution. **OS environment** refers to the resources that the user processes must access through the operating systems, such as swap space, file systems, communication channels, keyboard, monitors, process id assignments, time, etc. In this paper, we use the term **process state** to refer to everything that is included in a checkpoint, and the term **process environment** to refer to everything that is not included in a checkpoint but can affect program behavior. In other words, while the process state is restored to the checkpointed state at the time of recovery, process environment is not. Clearly, volatile state should be part of the process state and OS environment should be part of the process environment. The focus of this paper is on the following issue: "*should the persistent state belong to the process state or the process environment?*" Based on our experience, the answer is application-dependent, and the flexibility of making such a decision on a per-application basis can often lead to interesting new applications of checkpointing.

To our knowledge, existing Unix checkpoint libraries handle only *active* files, i.e., opened and not yet closed, at the time when a checkpoint is taken [1, 2]. Therefore, only part of the persistent state is included in the process state, as shown in Figure 1(a). Moreover, what part of the persistent state is checkpointed depends on when the checkpoint is taken. We will give examples in Section 3 to demonstrate that the above approach may lead to inconsistent recovery. Since the persistent state is often an important part of most long-running applications, we have developed a technique to include all persistent state in the process state, as indicated in Figure 1(b), to guarantee truly consistent checkpointing and transparent recovery[3]. The key concept behind this technique is that checkpointing a single-process application can no longer be achieved with a single snapshot, and *lazy checkpoint coordination* [4] can be used to make globally consistent checkpointing feasible.

Figure 1 (b) also implies that when a program fails and restarts from a checkpoint, it can have a different behavior if the OS environment is different. This observation suggests a new approach to software fault tolerance: the **environment diversity** approach executes the same (failed) program in a different environment so that the program can follow a different execution path and bypass the original software bugs that caused the failure. Section 4 gives examples to illustrate how transient software failures can be recovered by automatic environment diversity, and how permanent software failures can also be recovered by *intro-*

[1]The authors are with AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. Contact author: Yi-Min Wang (ymwang@research.att.com).

[2]Volatile state also includes those operating system kernel structures that are essential to current program execution, for example, the program counter, stack pointer, open file descriptors, signal masks and handlers.

[3]Strom et al. described a disk checkpoint manager for checkpointing disk files in a self-recovering distributed operating system [3]. In contrast, our approach has focused on developing application-level techniques that can be incorporated into existing standard Unix applications.

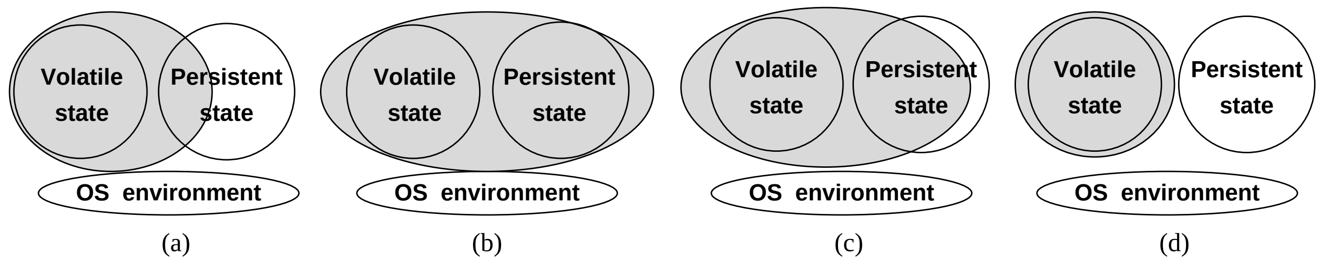


Figure 1: Process state (shaded area) vs. process environment (nonshaded area).

ducing environment diversity, for example, through *process migration* [5].

When a checkpoint includes all volatile state and persistent state, the recovered process is expected to perform basically the same functions as was the failed process (except for the possibly different execution for bypassing software bugs). In other applications where checkpointing is used as a mechanism for saving intermediate process state, it may be desirable to explicitly exclude certain part of the persistent state from the process state, as shown in Figure 1(c), so that the saved intermediate state can also be used as a starting point for executing new tasks. Usually, an application-specific saving routine needs to be written which can be a time-consuming and error-prone task. We give an example in Section 5 to show how our checkpointing library can be easily incorporated into an existing application to provide such a facility by excluding the input data files from the process state.

In Section 6 we introduce the technique of *memory rejuvenation* based on the process state structure shown in Figure 1(d). In a long-running application, undesirable state related to memory management may gradually build up either because some allocated memory is not properly deallocated after its usage or because of the limitation and/or the weakness of the memory management algorithm. This kind of *virtual memory aging* process can gradually degrade system performance and eventually cause software failures. Memory rejuvenation is an on-line preventive rollback technique which checkpoints the memory of a process at a “clean” state and periodically rolls back the process to that state (from a point where all useful state has been saved as persistent state) in order to prevent software failures.

In the next section, we first give a brief description of the Unix checkpointing library `libckp`, and the overhead measurement for a set of long-running benchmark programs including commercial, industrial and research applications.

2 `Libckp`: A Checkpoint Library for Unix

`Libckp` is a library for checkpointing Unix processes. It saves and restores the data segments of user applications as well as dynamic shared libraries, stack segment and pointer, program counter, file descriptors, signal masks and handlers, etc. Compared with other existing Unix checkpoint libraries [1, 2], `libckp` has the following unique features which we have found crucial for making checkpointing and recovery an attractive tool to the users.

1. The library includes user files as part of the process state that is checkpointed and recovered. More specifically, when a process rolls back, all the modifications it has made to the files since the checkpoint are undone so that the states of the files are consistent with the volatile state.

2. For users who prefer *transparent checkpoints*, no changes to the source code or recompilation are necessary. Only object files are needed to link with the library to obtain the executables. This feature is essential when obtaining the source code is much more difficult than obtaining the object code, and is very desirable when recompilation takes a long time or may require special compilation environments to be successful. It also provides a uniform treatment for applications written in different programming languages such as C, C++ and Fortran.

3. For users who prefer *inserted checkpoints*, two basic function calls `chkpnt()` and `rollback(i)` are available. The function `chkpnt()` returns 0 when a checkpoint has been successfully saved. The function `rollback(i)` rolls back the process to a previous checkpoint, and the execution will return from `chkpnt()` with a return value `i`. These two function calls can be considered as a generalization of the two Unix system calls `setjmp()` and `longjmp()` to include the restoration of global variables and persistent state. They together provide powerful execution controls for many interesting applications.

4. To maximize the portability, we use a feature extraction tool `IFFE` (IF Features Exist) [6] at compilation time to determine which part of the code to activate, and a dynamic probing technique at run time to determine the boundaries of the stack and data segments.

Table 1 shows the overhead measurement for 14 long-running programs including CAD applications, simulation programs and signal processing applications. *TimberWolf* [7] is a complete timing-driven placement and global routing package applicable to row-based and building-block design styles. *Vdrop* [8] is a maximum voltage drop verification package. (The simulated annealing part of the package was used in the experiments.) *ACCORD* (Automatic Checking and CORrection of Design errors) [9] is a tool to verify a logic circuit implementation and correct logic design errors by formal methods. *Galant* [10] is a delay-area optimization package for ASIC design using a standard-cell library approach. Simulated annealing was used to implement the optimizer. *CADsyn* is a commercial CAD synthesis program. *TILOS* [11] is a commercial transistor sizing package for minimizing the sum of transistor sizes in synchronous CMOS circuits according to performance specifications. (The input circuit used for the experiment is a 15,498-transistor subcircuit of a commercial microprocessor.) *DBsim* is a program for simulating database creation, traversal and reorganization. *Qsim* is a simulation program for fixed-rate encoding of a second-order Gauss Markov source using an adaptive buffer-instrumented entropy-constrained trellis-coded quantizer. *SPRUN* is a simulation environment for experiments with real-time digital signal processing algorithms. *Csim* is a simulation program for coded channel in wireless communications. *LPC2TD* is a speech processing program for efficient coding of LPC (Linear Predictive Coding) parameters by temporal decomposition. *HERest* is a model training program for speech recognition. *VFSM* (Virtual Finite State Machine) [12] validator is a program that exhaustively generates possible execution sequences of a network of communicating processes, checking for errors in process interaction such as deadlock, livelock and unexpected inputs. (The example used in the experiment consisted of three VFSMs representing a protocol for signalling the digits of a telephone number over an interoffice trunk line.) *Winxe* mimics natural input speech through sophisticated models of speech production (i.e., for the glottis and for the vocal tract).

The size of the source code ranges from a few thousand to a hundred thousand lines of code; the execution time ranges from 2 to 17 hours; the checkpoint size ranges from 0.3 to 40 megabytes. The checkpoints are either sent to a remote file server or stored on a local disk, depending on the file system configuration of each organization. Local checkpoints can be taken with a much lower overhead and do not generate network traffic, but the checkpoints may not be available when the local machine needs rebooting or repair. The checkpoint interval is 30 minutes which is the default value in `libckp`. The result shows that checkpoint overhead is in general less than 7% for most applications. The only exception is the *DBsim* program which has the largest checkpoint size of 40 megabytes and checkpoint overhead of 11%. By directly transmitting the checkpoint data to the file server through Unix communication primitives in order to bypass the slow NFS, the checkpoint overhead can be reduced to 4.3%.

3 Checkpointing Persistent State

Existing Unix checkpointing libraries either do not support the rollback of user files or only provide the capability to a limited extent. Unlike the incorrect recovery of volatile state which usually cause obvious process failures, incorrect rollback of persistent state often leads to undetectable corrupted files and therefore has become the primary concern of many users. We have found that supporting file rollbacks is important in practice since most serious applications involve file operations, and requiring users to understand and deal with the limitations on file rollbacks often challenges the claim of transparency and ease of use. A straightforward but incomplete way of extending volatile state checkpointing to include persistent states is to record the file size and file pointer of each active file at the time of checkpoint. When a rollback is initiated, each of those files is truncated to the recorded size and its pointer is seeked to the recorded position. Figure 2 gives an example for which the above simple approach will result in an inconsistency between the volatile state and the persistent state. In Figure 2, the size of `fileapp` is not recorded in the checkpoint because it is not active at `chkpnt()`. As a result, `fileapp` is not truncated when a rollback occurs and so the character "4" will be incorrectly appended twice. Such an erroneous scenario can also exist if `chkpnt()` in Figure 2 is omitted, and the rollback is done by restarting the program from the very beginning. This shows that persistent state checkpointing is important even for non-long-running applications, and therefore has an even wider application than volatile state checkpointing.

A naive way to avoid the above incorrectness is to checkpoint all the user files when `chkpnt()` is called, but that would be prohibitively expensive. Even if the user can supply the information as to which files are involved in the current program execution, the checkpoint overhead may still be unacceptably high if the number of files is large or the files themselves are large. Our approach is to model the

Table 1: Checkpoint overhead measurement for long-running applications (checkpoint interval = 30 minutes).

Program name	*TimberWolf*	*Vdrop*	*ACCORD*	*Galant*	*CADsyn*
Language	C	C	C	C	C
Code size (lines)	100K	11K	6K	1.2K	14K
Machine type	Sparc 5	Sparc 1	Sparc server	Sparc 5	Sparc 1
OS type	SunOS 4.1.3	SunOS 4.1.1	SunOS 4.1.2	SunOS 4.1.3	SunOS 4.1.1
Execution time	8h 56m	12h 8m	2h 13m	7h 49m	2h 54m
Checkpoint size	9.1M	7.4M	33M	1.7M	3.1M
Checkpoint type	remote	remote	remote	remote	remote
Overhead (time)	22m 55s	20m 6s	9m 8s	≈ 0	≈ 0
Overhead (%)	4.3%	2.8%	6.8%	$\approx$ 0%	$\approx$ 0%

Program name	*TILOS*	*DBsim*	*Qsim*	*SPRUN*	*Csim*
Language	C	C++	C	C	C
Code size (lines)	9.9K	13K	1.4K	19K	1.1K
Machine type	Sparc 5	Sparc 2	Sgi Indy	Sgi Indy	Sgi Indy
OS type	SunOS 4.1.3	SunOS 4.1.1	IRIX 5.2	IRIX 5.2	IRIX 5.2
Execution time	9h 39m	17h 7m	6h 50m	5h 48m	7h 1m
Checkpoint size	5.1M	40M	11M	1.2M	0.3M
Checkpoint type	remote	remote / non-NFS	remote	local	local
Overhead (time)	29m	1h 53m / 44m	25m	≈ 0	≈ 0
Overhead (%)	5.0%	11.0% / 4.3%	6.1%	$\approx$ 0%	$\approx$ 0%

Program name	*LPC2TD*	*HERest*	*VFSM*	*Winxe*
Language	C	C	C	Fortran
Code size (lines)	4K	12K	3K	30K
Machine type	Sgi Indy	Sgi Indy	Sparc 1	Sgi Challenge
OS type	IRIX 5.2	IRIX 5.2	SunOS 4.1.1	IRIX 5.2
Execution time	5h 45m	7h 4m	4h 53m	6h
Checkpoint size	0.3M	2.8M	17M	9M
Checkpoint type	local	local	remote	remote
Overhead (time)	5m	≈ 0	9m	17m
Overhead (%)	1.45%	$\approx$ 0%	3.07%	4.7%

```
/*  fileapp contains three integers 1, 2 and 3  */
chkpnt();
fp = fopen("fileapp", "a");  /*  for append */
fprintf(fp, "%d", 4);
fclose(fp);
/*  failure occurs, roll back  */
unlink("fileapp");  /*  remove the file */
```

Figure 2: Example illustrating the need of correct rollback of persistent state.

volatile state and the persistent state as a *multiple-process system*, the file operations as *inter-process communications*, and the consistency problem as a *checkpoint coordination* [13, 14] problem. By means of dependency tracking for file operations, we use *lazy checkpoint coordination* [4] to make checkpointing persistent state feasible.

The basic concept of lazy coordination is that checkpoints for coordination purpose need not be taken at the time of checkpoint initiation by the initiating process; they can be delayed until the state inconsistency due to message dependency is about to occur. By considering each user file as a separate process and the main process as the checkpoint initiator, lazy coordination translates into the following: user files that are not active at the time of checkpoint do not have to be checkpointed when `chkpnt()` is invoked; it suffices to record the size of a file when the file becomes active and to make a shadow copy of the file when the portion that existed at `chkpnt()` is about to be modified. For the example shown in Figure 2, the size of `fileapp` is recorded (on disk) at `fopen()` so that at the time of rollback `fileapp` can be truncated to the correct size to undo the effect of `fprintf()`. In another case, suppose the failure does not occur; then a shadow copy of `fileapp` will be generated at `unlink()`. If a failure occurs later on, the shadow copy and the recorded size can be used to restore `fileapp` to have both correct contents and correct size. A natural optimization to further reduce both run-time and space overhead is to perform the shadowing on a page-by-page basis [3].

4 Bypassing Premature Software Exits

Design diversity [15, 16] and **data diversity** [17] are two well-known approaches to software fault tolerance. In order to recover from a software failure, the design diversity approach executes a different program (implementing the same function) on the same set of data, and the data diversity approach executes the same program on a different (but equivalent) set of data. Figure 1(b) suggests a third approach which we call the **environment diversity** approach. By restarting from a checkpoint that includes the entire volatile and persistent state, the same program running with the same set of data can still have different behavior if the OS environment is different. Therefore, the diversity in the OS environment provides an opportunity to bypass the software bugs that caused the failure. In this paper, we focus on the *virtual memory environment* which is part of the OS environment, and use real-life examples to demonstrate how environment diversity can bypass premature software exits.

Figure 3 shows a program segment that is commonly found in Unix applications which allocate dynamic memory through the `malloc()` function call. When a program fails to allocate any more memory, this segment is invoked to print out an error message and cause the software to exit prematurely. For long-running applications, this kind of premature software exits can be as undesirable as software failures because a lot of useful work can be wasted. We will show that the out-of-memory condition is in fact due to a problem in the virtual memory environment, and the resulting software exit can be bypassed when the environmental problem disappears by itself or is explicitly eliminated.

```
If  ((ptr = malloc(size)) == NULL) {
      print malloc error message;
      exit;
}
Use ptr;
```

Figure 3: Memory allocation failure.

Although the virtual address space of one process is supposed to be independent of that of any other process running on the same machine, processes do have to share the same swap space and, as a result, can potentially interfere with each other through memory allocation. More specifically, one process may run out of memory because other processes have exhausted the remaining swap space. The following experiment was conducted to illustrate the point. We started three programs: *TimberWolf*, *Vdrop* and *CADsyn* on the same machine at the same time. After 30 minutes, a malicious program was submitted to the same machine to constantly allocate more memory. The intent was to exhaust the swap space so that when any of the three programs requests any more memory, it would be forced to exit because of a memory allocation failure. The result is: *CADsyn* exited after 55 minutes; *Vdrop* exited after 3 hours and 30 minutes; only *TimberWolf* was able to finish the entire execution after 33 hours because it has a built-in memory management module which allocates all the required memory at the very beginning. The experiment suggests that, for applications requiring dynamic memory allocation and running on heavily loaded machines where swap space contention can be serious, checkpointing is highly valuable for preventing a total loss of useful work due to the out-of-memory problem. More specifically, if the two programs that exited had taken periodic checkpoints, then they could be restarted from their checkpoints and successfully finish their executions on the *same* machine after the malicious program was killed.

One primary difference between a premature software exit and a software failure is that, in the former, the program is still under control at the point just before it exits. This observation motivates our proposal of the program construct shown in Figure 4 for protecting applications against environmental problems. The program segment operates as follows. When `malloc()` function call fails, it retries for `MAX_RETRY_COUNT` times with each retry separated from the next one by `RETRY_WAIT_PERIOD` seconds. If the environmental problem is transient and any of the retries succeeds, the program can proceed; otherwise, a checkpoint is taken just before the program exits. (Note that `chkpnt()` returns 0 when it succeeds and return a negative values when it fails.) When the program is later restarted, the execution returns from `chkpnt()` with a default return value 1. The program then tries the memory allocation again and may succeed if the contention for swap space no longer exists. Compared with periodic checkpointing, the program structure shown in Figure 4 has two important advantages: first, since the program will continue from where it exited, no useful work is lost (except for the time for taking the checkpoint and restoring the state); second, when used alone, it does not incur run-time overhead during failure-free execution and yet provides the protection when the program needs it.

```
retry_count = 0;
while ((ptr = malloc(size)) == NULL) {
        retry_count = retry_count + 1;
        if (retry_count == MAX_RETRY_COUNT) {
            if (chkpnt() <= 0) {
                print malloc error message;
                exit;
            } else retry_count = 0;   /*  recovery */
        }
        sleep(RETRY_WAIT_PERIOD);
}
Use ptr;
```

Figure 4: Program construct for tolerating transient and permanent environmental problems such as the out-of-memory conditions.

The out-of-memory condition can be permanent when the machine simply does not have enough swap space to satisfy the memory requirement of an application. In other words, the environmental problem may not disappear automatically and some actions have to be taken to explicitly force environment diversity in order to bypass the premature exit. Figure 5 shows an example of using process migration for such a purpose. Program *faultsim* is a fast and accurate diagnosis fault simulator for digital circuits, consisting of approximately 7,000 lines of C++ code. The program was first submitted to a Sparc 1 with 32 megabytes of swap space. It ran out of memory after 2 hours and 50 minutes, took a checkpoint before it exited and was then migrated to a Sparc 2 with 80 megabytes of swap space. After another 27 minutes, it ran out of memory again and was migrated to a Sparc server with 140 megabytes of swap space, on which it ran for another 1 hour and 13 minutes and eventually finished the execution and produced correct results. The above technique is particularly useful for applications with unpredictable memory requirements which, for example, depend on the input cases. If a long-running program can usually finish its execution on a desk-top workstation but can potentially run out of memory in some cases, it is often submitted to a server machine with a large amount of memory in order to minimize the risk of premature exits. This can unnecessarily increase the demands on the more expensive CPU cycles of the server machines while leaving the desk-top workstations idle. By using the technique in Figure 4, such a program can always be started on a workstation and then migrated to a server only when it runs out of memory. This can effectively reduce the expenses for computing resources in an organization where the server machines are always heavily loaded.

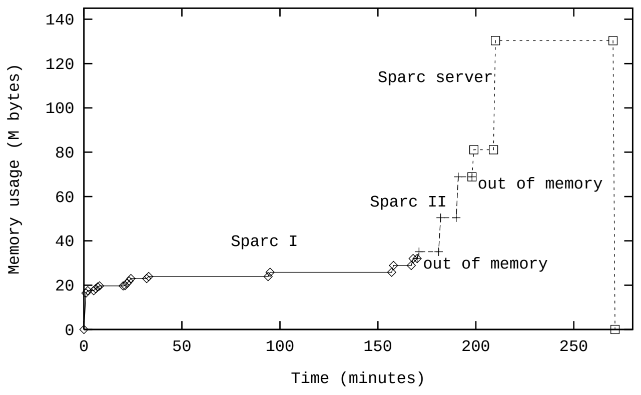


Figure 5: Migration when out of memory.

5 Bypassing Long Initialization

Many long-running applications have a program structure similar to Figure 6(a). The `input` which determines the type of processing the current execution is supposed to perform is obtained through `Read_Input()`. A long initialization routine is invoked to construct the initial volatile state as the basis for processing. The routine `Processing()` operates on the `init_state` according to `input` and generates a `new_state`, part of which will be written out as the main result. For example, in a speech processing program which evaluates techniques for verifying recognition hypotheses, `input` corresponds to the choice of a particular technique; `Long_Init()` reads in the hypotheses and the associated likelihoods from a 150-megabyte remote database; `Processing()` produces the list of hypotheses which should be rejected. As another example, in a database simulator, `input` contains the simulation parameters and the names of the trace files; `Long_Init()` creates a database in memory according to the first trace file consisting of creation events; `Processing()` performs traversal or reorganization simulations on the volatile database based on the other trace files containing simulation events.

```
input = Read_Input();
init_state = Long_Init();
new_state = Processing(input, init_state);
Write_Result(new_state);
                    (a)
input = Read_Input();
init_state = Long_Init();
exclude_input_files_from_chkpnt();
chkpnt();
input = Read_Input();
new_state = Processing(input, init_state);
Write_Result(new_state);
                    (b)
```

Figure 6: (a) Original program structure; (b) bypassing long initialization through checkpointing and restoration.

Very often, the `init_state` can be reused by many different executions with different `input`. It would be more efficient if `init_state` can be saved and then restored for use in future executions to avoid repeating the same time-consuming initialization. Usually, this is done in an application-specific way by identifying and saving only essential volatile state, which can be difficult to implement and error-prone. An alternative is to checkpoint the entire state by invoking `chkpnt()`, as shown in Figure 6(b), in order to guarantee correctness and consistency. The input files, which are part of the persistent state, are excluded from the checkpoint to allow new input files to be processed. For some applications, `Read_Input()` has to be invoked again after `chkpnt()` so that, when `init_state` is restored and the execution returns from `chkpnt()`, new parameters can be read in to overwrite the checkpointed old parameters. This feature has been used in a database simulator to save 40% of the total execution time.

6 Memory Rejuvenation

Ordinary trees need selective pruning to make them grow better; overgrown and untidy old shrubs need *rejuvenation*, a drastic "take it to the ground" pruning technique [18]. Similarly, normal program executions need careful resource management to maintain performance; long-running applications with accumulated undesirable resource states need *rejuvenation*, a drastic "total recall" resource deallocation technique. We first give examples of how undesirable memory management state can built up, and then describe the use of `libckp` for performing on-line memory rejuvenation.

```
#define MG (1024*1024)
for (i=0; ; i++) {
     if ((ptr1 = malloc(MG)) == NULL) exit(1);
     if ((ptr2 = malloc(MG)) == NULL) exit(2);
     free(ptr1);
}                            (a)
for (i=0; ; i++) {
    if (i == 0) {
       for (k=0; k<MG/2; k++) ptr1[k] = malloc(32);
       for (k=0; k<MG/2; k++) free(ptr1[k]);
    } else {
            if ((ptr2 = malloc(20*MG)) == NULL) exit(1);
            free(ptr2);
    }
}                            (b)
for (i=0; ; i++) {
    if ((ptr = malloc(i*MG)) == NULL) exit(1);
    free(ptr);
}                            (c)
```

Figure 7: Examples of undesirable state accumulation due to memory allocation and deallocation. (a) Memory leakage; (b) memory caching; (c) weak memory reuse.

Figure 7(a) is a **memory leakage** example. The block of memory pointed by `ptr2` in the nth iteration is "leaked", i.e., no longer useful but cannot be reused, when `ptr2` is used to point to another block of memory in the $(n+1)$th iteration. Due to the leakage, the program will eventually run out of memory and exit. Memory leakage problem is likely to exist in those programs with complicated control flow, in which proper memory deallocation may be missing on certain execution paths. Figure 7(b) is a **memory caching** example. When executed on a machine with 30 megabytes of swap space and a standard version of `malloc()`, the code unexpectedly exits at $i = 1$ because, for efficiency, the deallocated small blocks of 32 bytes are not coalesced, and therefore a total of 16 megabytes are not available for the 20-megabyte request. This kind of caching mechanism is commonly found in both standard and customized memory management routines. Strictly speaking, caching is a performance feature and is not a memory leakage bug. But when an application runs out of memory and is not provided with any method to reclaim the cached memory blocks, memory caching can be as undesirable as memory leakage[4].

Figure 7(c) shows an example of a **weak memory reuse** problem which exists in another popular version of `malloc()`. With 30 megabytes of swap space, the program in Figure 7(c) exits at $i = 16$ (instead of $i = 30$) because, when receiving the 16-megabyte memory request, the `malloc()` implementation cannot reuse the previously deallocated 15 megabytes. As a result, the attempt to allocate additional 16 megabytes fails because only 15 megabytes are available outside the management of `malloc()`.

The three examples demonstrate that undesirable memory management state may build up as a long-running application continues to execute, and cause a program to exit prematurely even when a machine in fact has enough physical resource to satisfy the memory requirement of the program. In order to prevent that, *memory rejuvenation* can be performed by periodically rolling back the volatile state to a previously checkpointed "clean" state in order to discard the undesirable state. Clearly, one limitation is that memory rejuvenation can only be performed when the volatile state does not contain any useful information. For long-running applications consisting of a large number of *independent* iterations (as shown in Figure 8(a)), the boundary between two consecutive iterations are the ideal place for memory rejuvenation. Note that in Figure 8(a), `new_state` contains the read-only `init_state` and the accumulated undesirable state, and `Processing()` depends only on `init_state`. Figure 8(b) gives the program construct for performing memory rejuvenation. The variable `REJUV_PERIOD` specifies how often rejuvenation is to be performed in terms of number of iterations. The "clean" `init_state` is **checkpointed once at the very beginning of the first iteration** $i = 0$. Before processing each iteration with i being a multiple of `REJUV_PERIOD`, all useful state that still remains in memory is flushed to the disk and the volatile state is rolled back to `init_state` except that the loop index i retains its value to ensure correct progress.

```
init_state = Initialization();
for (i=0; i<NUM_CASES; i++) {
   new_state = Processing(init_state, i);
   Write_Result(new_state);
}
                (a)
init_state = Initialization();
for (i=0; i<NUM_CASES; i++) {
   if (i != 0) {
      if (i % REJUV_PERIOD == 0) {
         Flush_Output_Buffer();
         rollback(i);
      }
   } else i = chkpnt();
   new_state = Processing(init_state, i);
   Write_Result(new_state);
}
                (b)
```

Figure 8: Program construct for rejuvenation. (a) Original program; (b) with rejuvenation.

On-line memory rejuvenation as shown in Figure 8(b) is particularly valuable for applications requiring a long initialization procedure. Another important application domain is when the accumulation of undesirable state is caused by some imported library functions of which the source code is not available. For example, it would certainly be desirable if memory leakage problem can be identified and corrected, and there exist several commercial software tools such as `Purify` [20] and `Sentinel` [21] to help software developers to detect the problem. But when an application is using an imported software package and memory leakage is inside that package, the problem may be detectable but may not be correctable due to the unavailability of source code or limited knowledge of the program structure. *Garbage collectors* [22] can be used to perform on-line garbage collection to alleviate the memory leakage problem, but it has certain limitations[5] and also cannot solve the problem caused by memory caching. Memory rejuvenation is essentially a user-invoked garbage collector which exploits application-specific information to perform drastic and effective memory reclamation.

The logic correction program *ACCORD* as described in Section 2 consists of 6,000 lines of C code and also makes heavy use of another 10,000-line imported package for memory management and high-level structure allocation. The package has a memory leakage problem and a

[4]This is a reason why a new Virtual Memory ALLOCation package (or `vmalloc`) [19] allows specifying exception handlers that will be called when memory space is out so that garbage collection can be performed at the right time.

[5]For example, a memory block may still have a global variable pointer pointing to it, but the program has finished using that block and forgot to free it. This kind of *soft leakage* problem, in contrast with the *hard leakage* problem in Figure 7(a), in general cannot be detected without application-specific information.

structure caching feature, which caused *ACCORD* to exit prematurely when running out of memory. The upper curve in Figure 9(a) illustrates the scenario. While 100 cases need to be processed to evaluate algorithm performance, the program ran out of memory and exited at the 52nd iteration on machine A with 30 megabytes of swap space. By using the technique described in Section 4, the program took a checkpoint before it exited and then was migrated to machine B with a 70-megabyte swap space to finish the execution. By using the memory rejuvenation technique with `REJUV_PERIOD` set to 15 iterations, the memory usage never exceeded 30 megabytes and so the entire execution could be finished on machine A, as shown in the lower curve. For this 16,000-line application, only 7 lines of C code need to be added to the beginning of the main loop in the main routine to perform memory rejuvenation.

In order to measure the overhead of rejuvenation, we ran the program on machine B and compared the execution times of the rejuvenated and the unrejuvenated versions in Figure 9(b). For the various `REJUV_INTERVAL` values used in our experiment, rejuvenation actually made the program run 15%-27% *faster*. This was because the checkpointing and rollback overhead was offset by the following two factors. First, the software package linked with *ACCORD* has a built-in garbage collector; when a larger amount of memory needs to be managed, garbage collection would incur a larger overhead. Second, the memory locations in use were more scattered due to memory leakage and so the paging overhead would be higher.

7 Summary

By supporting the checkpointing of persistent state and providing `chkpnt()` and `rollback()` function calls, our checkpointing library `libckp` allows powerful execution controls to be easily incorporated into user applications. The capability to explicitly exclude part of the persistent state from the checkpoint provides further flexibility for various applications. The part of state that is checkpointed and recovered can be used to restore desirable state or to discard undesirable state. The part of state that is not checkpointed can be used to bypass premature software exits or to accept new input data. Several examples have been presented to demonstrate the usefulness of the above concepts.

Acknowledgement

The authors wish to express their sincere thanks to Oscar Agazzi, Jing-Yang Chou, Weitong Chuang, Bill Coughran, Jack Fishburn, Sunil Gupta, Ismed Hartanto, Artur Klauser, Harish Kriplani, C. C. Lee, Jin-Nan Liaw, Antoine N. Mourad, Rick Rose, J. Schroeter, Ram Sharma, Mark Staskauskas, Wern-Jieh Sun and William Zeng for their help with the experimental results, to David Berkley, Tsuhan Chen, Premkumar Devanbu, Fred Douglis, Larry Rabiner, Gaurav Suri, Kuansan Wang, Vijitha Weerackody, Alexander Wolf and Qiru Zhou for their comments, and to Chris Dingman and Sadik Kapadia for their challenges to persistent state checkpointing.

References

[1] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration ouside the Unix," in *Proc. Usenix Winter Conference*, 1992.

[2] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Proc. Usenix Technical Conference*, pp. 213–224, Jan. 1995.

[3] R. E. Strom, , S. A. Yemini, and D. F. Bacon, "A recoverable object store," in *Proc. Hawaii International Conference on System Sciences*, pp. II–215–II–221, Jan. 1988.

[4] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 78–85, Oct. 1993.

[5] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the Sprite implementation," *Software - Practice and Experience*, Vol. 21, No. 8, pp. 757–785, Aug. 1991.

[6] G. S. Fowler, D. G. Korn, J. J. Snyder, and K.-P. Vo, "Feature-based portability," in *Proc. VHLL Usenix Symposium on Very High Level Languages*, Oct. 1994.

[7] W.-J. Sun and C. Sechen, "Efficient and effective placement for very large circuits," in *Proc. IEEE International Conference on Computer-Aided-Design*, pp. 170–177, 1993.

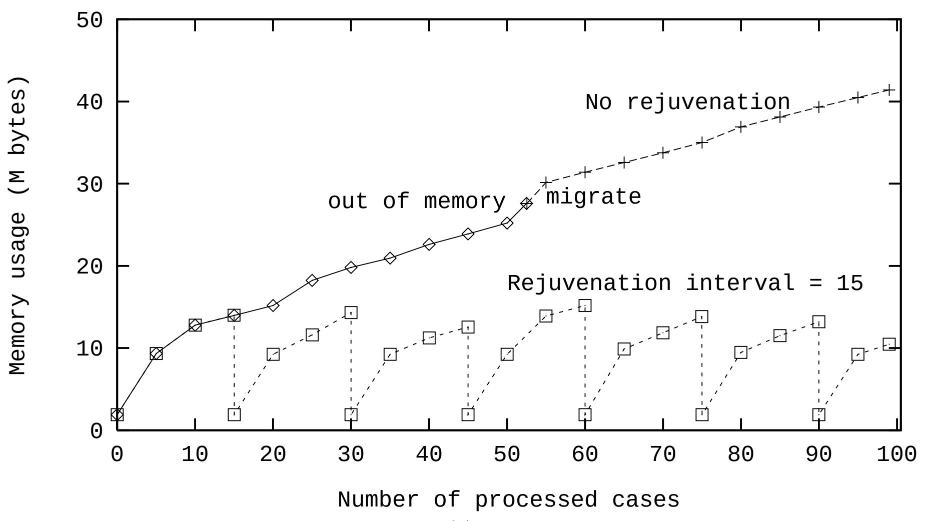
[8] H. Kriplani, F. Najm, and I. Hajj, "Pattern independent maximum current estimation in power and ground buses of CMOS VLSI circuits: Algorithms, signal correlations and their resolution." submitted to *IEEE Transactions on Computer-Aided Design*, Feb. 1993.

[9] P. Y. Chung, Y. M. Wang, and I. N. Hajj, "Diagnosis and correction of logic design errors in digital circuits," in *Proc. the 30th ACM/IEEE Design Automation Conference*, pp. 503–508, 1993.

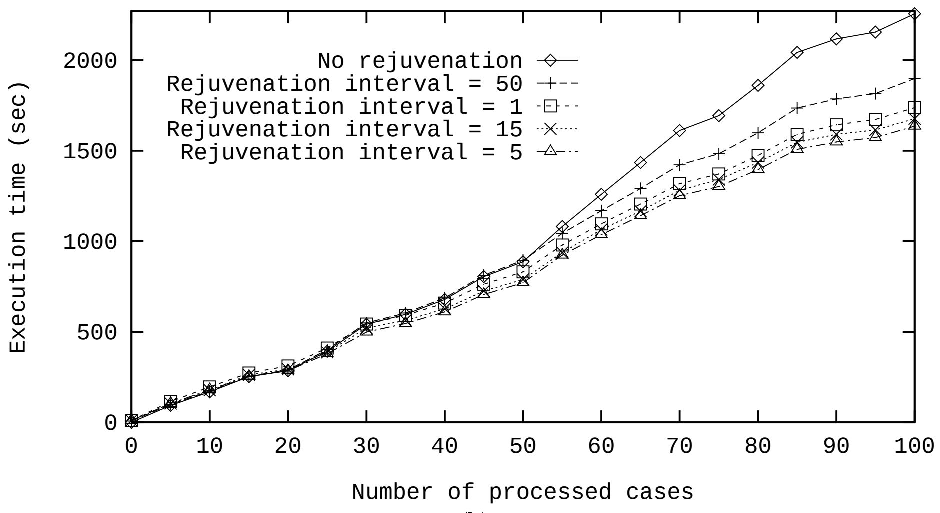
[10] W. Chuang, S. S. Sapatnekar, and I. N. Hajj, "Timing and area optimization for standard-cell VLSI circuit design," *IEEE Trans. Computer-Aided Design*, to appear.

[11] D. Hill, D. Shugard, J. Fishburn, and K. Keutzer, *Algorithms and Techniques for VLSI Layout Synthesis*. Kluwer, 1989.

[12] A. Flora-Holmquist and M. Staskauskas, "Software design technology for communication systems reliability," in *Proc. 1994 International Conference on Communication Technology*, June 1994.



(a)



(b)

Figure 9: (a) Memory usages and (b) execution times for program *ACCORD* with and without rejuvenation.

[13] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

[14] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, Vol. SE-13, No. 1, pp. 23–31, Jan. 1987.

[15] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 12, pp. 1491–1501, Dec. 1985.

[16] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 2, pp. 220–232, June 1975.

[17] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault-tolerance," *IEEE Trans. Comput.*, Vol. 37, No. 4, pp. 418–425, Apr. 1988.

[18] D. Fell, *The Essential Gardener*. Avenel, NJ: Crescent Books, 1993.

[19] K.-P. Vo, "Writing reusable libraries with disciplines and methods," in *submitted to ACM SIGSOFT Symposium on Software Reusability*, 1995.

[20] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. Winter Usenix Conference*, pp. 125–136, Jan. 1992.

[21] A. S. Corporation, "SENTINEL run-time analysis tool: User's guide." 1994.

[22] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software - Practice and Experience*, Vol. 18, No. 9, pp. 807–820, Sept. 1988.