Libckpt: Transparent Checkpointing under Unix

James S. Plank

Micah Beck

Gerry Kingsley

Department of Computer Science
107 Ayres Hall
University of Tennessee
Knoxville, TN 37996
[plank,beck,kingsley]@cs.utk.edu

Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544
li@cs.princeton.edu

August 5, 1994

Technical Report CS-94-242
Department of Computer Science
University of Tennessee
Knoxville, TN 37996

Appearing in Usenix Winter '95

Libckpt: Transparent Checkpointing under Unix

James S. Plank, Micah Beck, Gerry Kingsley

[plank,beck,kingsley]@cs.utk.edu
Department of Computer Science, University of Tennessee, Knoxville, TN 37996

Kai Li

li@princeton.edu
Department of Computer Science, Princeton University, Princeton, NJ 08544

August 29, 1994

Abstract

Checkpointing is a simple technique for rollback recovery: the state of an executing program is periodically saved to a disk file from which it can be recovered after a failure. While recent research has developed a collection of powerful techniques for minimizing the overhead of writing checkpoint files, checkpointing remains unavailable to most application developers. In this paper we describe **libckpt**, a portable checkpointing tool for Unix[1] that implements all applicable performance optimizations which are reported in the literature. While **libckpt** can be used in a mode which is almost totally transparent to the programmer, it also supports the incorporation of user directives into the creation of checkpoints. This *user-directed* checkpointing is an innovation which is unique to our work.

1 Introduction

Consider a programmer who has developed an application which will take a long time to execute, say five days. Two days into the computation, the processor on which the application is running fails. If the programmer has not planned for this event, his only choice is to restart the program and lose two days' work. Upon restarting the program, he still needs five days of continuous failure-free processor time to complete the job.

Libckpt is a checkpointing library designed for such a programmer. To use **libckpt**, all he must do is change one line of his source code and recompile with the library `libckpt.a`. No other modifications need be made. Upon execution, the program will periodically save its execution state to disk. Upon a processor failure, the programmer need only restart the program with the command line flag `=recover`, and the program will roll back to the most recently checkpointed state. In the example above, at most ten minutes of work will be lost, and the programmer needs approximately three more days of *non*-continuous failure-free processor time to complete the job.

[1] Unix is a trademark of UNIX Systems Laboratories, Inc.

Libckpt is a tool for transparent checkpointing on uniprocessors running Unix. It implements incremental and copy-on-write checkpointing, two optimizations well-known in the literature [FB89, LNP90, EJZ92, LNP94]. **Libckpt** is a user-level library and uses only facilities which are commonly available under Unix. **Libckpt** has been ported to and tested on a variety of architectures and operating systems with no kernel modifications. Source code for **libckpt** can be obtained at no cost by anonymous FTP from `cs.utk.edu:pub/plank/libckpt`.

In this paper, we show the performance gains available in **libckpt** through transparent incremental and copy-on-write checkpointing. In addition, we introduce a new optimization technique, implemented in **libckpt**, called *user-directed checkpointing*. User-directed checkpointing works under the assumption that a little information from the user can yield large improvements in the performance of checkpointing. We demonstrate that this assumption is often valid.

2 Transparent Checkpointing

The goal of checkpointing is to establish a *recovery point* in the execution of a program, and save enough state to restore the program to this recovery point in the event of a failure. The main use of checkpointing is to provide a mechanism for performing *rollback recovery*, a general means of fault tolerance. The strength of rollback recovery is its ability to provide fault tolerance in the presence of unanticipated faults — faults which were not envisioned in the design of the program [AL81].

The most straightforward mechanism for establishing a recovery point under Unix is to suspend execution of the application while the entire contents of a process's memory and registers are written to a file. This is called *sequential* checkpointing because disk transfers are not interleaved with program execution. Recovery is effected by reloading the executable from its original file, and then reconstructing the memory and register state from the checkpoint file. This is akin to creating a `core` file, from which a user may recover using the `undump` utility and `execve()`.

We say that checkpointing is *transparent* when no changes need be made to the application program. While transparency is easy to obtain at the kernel level, it is harder to achieve in a user level checkpointing library. All current implementations of checkpointing share this limitation: They operate transparently and correctly so long as the application is *well-behaved* in a sense we will define in Section 4. **Libckpt** *and all other user-level checkpointers can cause a correct but ill-behaved application to fail or to produce incorrect output upon recovery.*

Checkpointing with **libckpt** is not completely transparent. The name of the initial procedure in C must be changed from `main()` to `ckpt_target()`. This enables **libckpt** to gain control of the program as it starts, check the command line for the `=recover` flag, read a file called `.ckptrc` to set checkpointing parameters, and begin checkpointing. In FORTRAN, **libckpt** is enabled by changing the main `PROGRAM` module to `SUBROUTINE ckpt_target()`. No other program modifications are needed.

By default, once **libckpt** gets control of a program, it generates a timer interrupt every ten minutes, and takes a sequential checkpoint at each interrupt. This and other defaults can be changed by placing appropriate lines in the `.ckptrc` file. In this section, we describe all options, where appropriate, as they would appear in the `.ckptrc` file.

> Placing the line "`checkpointing` <`on`|`off`>" in the `.ckptrc` file turns checkpointing on or off. If `off`, **libckpt** will take no checkpoints and will not affect the execution of the application. The default is `on`.
>
> `dir` <*directory*> specifies the directory in which checkpoint files are created and found. The default is the current directory.
>
> `maxtime` <*seconds*> defines the interval between checkpoints. At the beginning of the program, and after each checkpoint, **libckpt** calls `alarm(`*seconds*`)` and takes a checkpoint upon catching each `ALRM` signal. Setting the timer interval to zero turns off all timer-based checkpointing. The default value of `maxtime` is 600 (10 minutes).

Many optimizations to simple sequential checkpointing have been described in the literature. **Libckpt** implements all well-known optimizations that are applicable to general-purpose uniprocessor checkpointing, as well as the new user-directed optimization. In the remainder of this section, we consider each of them in turn.

2.1 Incremental Checkpointing

When a checkpoint is taken, only the portion of the checkpoint that has changed since the previous checkpoint need be saved. The unchanged portion can be restored from previous checkpoints. **Incremental checkpointing** [FB89, WM89, EJZ92] uses page protection hardware to identify the unchanged portion of the checkpoint. Saving only the changed portion reduces the size of each checkpoint, and thus the overhead of checkpointing.

> `incremental` <`on`|`off`> turns incremental checkpointing on or off. The default is `off`.

In general, the size of a non-incremental checkpoint grows very slowly over time if at all. Moreover, only the most recent checkpoint file need be retained for recovery — older ones may be deleted. In contrast, old checkpoint files cannot be deleted when incremental checkpointing is employed, because the program's data state is spread out over many checkpoint files. The cumulative size of incremental checkpoint files will increase at a steady rate over time, since many updated values may be saved for the same memory location. In order to place an upper bound on the cumulative size of incremental checkpoint files, it is necessary to coalesce all old checkpoint files into one new file, and then discard the old files. For this purpose, **libckpt** includes a utility program `ckpt_coa`, which coalesces a collection of incremental checkpoint files into a single checkpoint file.

`maxfiles` <n> sets the maximum number of incremental checkpoint files to n. After n checkpoint files have been created, **libckpt** calls `ckpt_coa` process to coalesce them down to one file. If $n = 1$, then no incremental checkpointing can occur. Values of n greater than one allow the user to strike a balance between incremental checkpointing and total disk storage. The default is $n = 1$.

Libckpt uses page protection to identify which pages should be included in incremental checkpoints. Specifically, after initialization and after each checkpoint, the `mprotect()` system call is invoked to set the protetion of all pages in the data space to *read-only*. When a write occurs to a memory location in a protected page, the `SEGV` signal is caught by a handler in **libckpt**. The faulting page has its access protection set to *read-write*, and the page is marked as dirty. When `libckpt` takes the next checkpoint, only the dirty pages are included.

2.2 Forked Checkpointing

Incremental checkpointing as described in the previous section is still sequential: Execution of the application program is suspended while the checkpoint file is written out. An alternative is to make a copy of the program's data space and to use an asynchronously executing thread of control to write the checkpoint file. This is called "main-memory checkpointing" [LNP94], and improves checkpoint overhead if there is enough physical memory to hold the checkpoint, as the saving of the checkpoint to disk is overlapped with the execution of the application.

The Unix `fork()` primitive provides exactly the mechanism needed to implement main-memory checkpointing [PL89, LFS93]. When forked checkpointing is specified, **libckpt** forks a child process, which creates and writes the checkpoint file while the parent process returns to executing the application. The `fork()` system call provides the child with a fixed snapshot of the parent's data space and a separate thread of control.

`fork` <`on`|`off`> in the `.ckptrc` file turns main forked checkpointing on or off. The default is `off`.

An important improvement to main-memory checkpointing is copy-on-write checkpointing [LNP90, EJZ92, LNP94]. Here the copy of main memory is taken using copy-on-write [TLC85, FR86]. Many implementations of `fork()` use a copy-on-write mechanism to optimize the copying of the parent's address space [Ste92]. Thus, forked checkpointing corresponds to either main-memory checkpointing or copy-on-write checkpointing, depending on the operating system's implementation of `fork()`.

2.3 Checkpoint Compression

With checkpoint compression, a standard compression algorithm like LZW [Wel84] is used to shrink the size of the checkpoint [LF90, PL94]. While this may be successful at reducing checkpoint size, it only improves the overhead of checkpointing if the speed of compression is faster than the speed of disk writes, and if the checkpoint is significantly compressed. For uniprocessor checkpointing, this is not the case, and compression has only been shown to be effective in parallel systems with disk contention [PL94]. For this reason, checkpoint compression is not implemented in **libckpt**.

3 User-Directed Checkpointing

All the optimizations presented so far maintain the transparency of checkpointing through techniques that are not visible to the application: signal handlers, page protection, and the creation of child processes. In this section, we consider a different approach that can improve on the performance of these transparent approaches and can also substitute for them when automatic mechanisms are not available. We call this approach "user-directed checkpointing." We consider two ways in which user-supplied directives can improve the performance of checkpointing: memory exclusion, and synchronous checkpointing.

3.1 Memory Exclusion

There are two situations where the values of memory locations can be excluded from a checkpoint file: when the locations are dead and when they are clean. In the case of dead locations, the values in memory will never be read or written, and thus do not need to be checkpointing. In the case of clean locations, the values in memory exist in a previous checkpoint and have not been changed. Thus they need not be saved in the current checkpoint. While the identification of excludable areas of memory can sometimes be automated (as in incremental checkpointing), **libckpt** also allows the programmer to declare them explicitly.

For example, suppose the user allocates a large temporary array `T` to make a calculation. When the *lifetime* of the data in array `T` is over, it will never be referenced again — the next use of array `T` will overwrite the old values. If a checkpoint is taken outside of the lifetime of array `T`, then it can be safely excluded from the checkpoint. Any recovered state from this point will not need to use the current values stored in array `T`.

The stack is a run-time mechanism that helps the checkpointer to determine the lifetime of local variables. This is one form of memory exclusion: only the live portion of the stack is saved. Unfortunately, this does not work for heap variables or for variables which reside in the statically allocated data segment.

The basis of incremental checkpointing is that clean data need not be repeatedly written to disk. In order to implement automatic incremental checkpointing, **libckpt** monitors page modifications using the `mprotect()` system call and a handler for the `SEGV` signal. This approach has a few weaknesses: it can only operate at the page granularity; system calls can fail rather than generating a `SEGV` signal when asked to write to a protected page; and on some systems `mprotect()` is not reliable.

In those cases where automatic mechanisms cannot infer all possible memory exclusions, the performance of checkpointing can suffer. For this reason, **libckpt** allows the programmer to manage memory exclusion explicitly through two procedure calls.

> `exclude_bytes`(*char *addr, int size, int start_now*): This procedure call tells **libckpt** to exclude the region of memory specified from subsequent checkpoints. It may be called when the user knows that these bytes are not necessary for the correct recovery from the program. *Start_now* is a boolean argument. If non-zero, then these bytes are excluded from the next and all subsequent checkpoints until `include_bytes()` is called. If *start_now* is zero, then these bytes are excluded from checkpoints starting *after* the next checkpoint. Memory locations which will not be written for some time are not actually clean, but will become clean after the next checkpoint is taken. Thus, by calling `exclude_bytes()` with a zero value for the *start_now* argument, the programmer can defer the exclusion of these memory locations until after they become clean.
>
> `include_bytes`(*char *addr, int size*): Because dead pages can become live and clean pages can become dirty, **libckpt** provides a mechanism for reversing the effects of memory exclusion. This procedure call tells **libckpt** to include the specified region of memory in the next and subsequent checkpoints. Thus, `include_bytes()` cancels the effect of calls to `exclude_bytes()`, although calls to `include_bytes()` do not have to match calls to `exclude_bytes()`. By default, **libckpt** includes all bytes in a process's active stack and data segments that have not been explicitly excluded.

User-directed memory exclusion can dramatically reduce the size of sequential and incremental checkpoint files, but it must be used very carefully. If a live region of memory is mistakenly excluded from a checkpoint, then a subsequent failure and recovery can cause an otherwise correct application to fail or to generate incorrect results.

3.2 Synchronous Checkpointing

In the previous section we discuss a mechanism for optimizing asynchronous checkpointing by excluding certain areas of memory. This allows the checkpointer to make use of data lifetime information which would not otherwise be available to it. However, the amount of data which can be excluded from the checkpoint is determined by the program's state *when the checkpoint is taken*. If the stack is large, or the size of excluded memory is small, then memory exclusion will have little effect.

Synchronous checkpointing is a user directive that allows the programmer to specify points in the program where it is most advantageous for checkpointing to occur. These are called "synchronous" checkpoints because they are not initiated by timer interrupts. Synchronous checkpoints should be inserted by the programmer at points where memory exclusion can have the greatest effect.

> `checkpoint_here()` is a procedure call specifying to take a synchronous checkpoint.

Synchronous checkpoints may be placed in program locations that are reached often. Checkpointing too often, however, can lead to poor performance, and in order to avoid this **libckpt** allows a minimum interval between checkpoints to be specified.

> `mintime` <*seconds*> specifies the minimum period of time that must pass between checkpoints. The default is zero. If `mintime` seconds have not passed since the previous checkpoint, then `checkpoint_here()` calls are ignored.

Synchronous and asynchronous checkpointing techniques can complement one another. If `maxtime` seconds have passed and no synchronous checkpoint has been taken since the last checkpoint, then an asynchronous checkpoint is still taken. However, the effect of memory exclusion is likely not to be as beneficial as in a synchronous checkpoint. If both the `mintime` and `maxtime` parameters are set, then the former specifies the minimum interval between synchronous checkpoints, and the latter specifies an interval after which an asynchronous checkpoint will be taken. Whenever a checkpoint is taken, both the minimum and maximum interval timers are reset.

If `maxtime` is zero, then asynchronous checkpoints are disabled. In this case, the specification of memory exclusion can be optimized for synchronous checkpoints, because the memory exclusions specified need only be valid at the calls to `checkpoint_here()`, and need not be kept constantly up-to-date to account for the possibility of a checkpoint being taken asynchronously.

3.3 An Example

There are many times when user-directed, synchronous checkpointing can yield large performance gains. Consider the example program in the left side of Figure 1. This is a typical driver program for many kinds of programs that repeat calculations over numerous points in a data set. The right side of Figure 1 shows how one should checkpoint this program with synchronous, user-directed checkpointing in **libckpt**:

By specifying that the checkpoint must be taken at the `checkpoint_here()` call, we are able to omit all of the variable `D` from the synchronous checkpoint. This is because `D` is initialized anew at each iteration of the program. If `D` is large, then user-directed checkpointing will be responsible for a significant savings in checkpoint overhead. Note that this will be a vast improvement over incremental checkpointing because the memory locations in `D` will be dirty at the time of the checkpoint.

Section 6 shows other successful examples of user-directed checkpointing.

4 The Mechanics of Checkpointing and Recovery

The motivation for checkpointing is to reconstruct the recovery point. We therefore begin with an overview of the recovery process before describing the details of checkpointing.

```
main()
{
  struct data_set *D;
  FILE *fi, *fo;

  D = allocate_data_set();
  fi = fopen("input", "r");
  fo = fopen("output", "w");
  while(read_data(fi, D) != -1) {
    perform_calculation(D);
    output_results(fo, D);
  }
}
```

```
ckpt_target()
{
  struct data_set *D;
  FILE *fi, *fo;

  D = allocate_data_set();
  fi = fopen("input", "r");
  fo = fopen("output", "w");
  while(read_data(fi, D) != -1) {
    perform_calculation(D);
    output_results(fo, D);
    exclude_bytes(D, sizeof(struct data_set), 1);
    checkpoint_here();
    include_bytes(D, sizeof(struct data_set));
  }
}
```

Figure 1: A typical scientific driver program

Recovery has four parts: process creation, data state restoration, system state restoration, and processor state restoration.

1. Process creation is implemented by invoking the checkpointed program with a special command line argument for recovery. This automatically restores the text portion of the process's state and begins execution. **Libckpt** parses the command line, detects the argument for recovery, and calls the recovery routine.

2. The recovery routine performs the rest of the recovery. Data state restoration means reading the checkpoint file to recreate the contents of data memory: This consists of the process's stack and data segments.

3. System state restoration means restoring as much of the operating system state as possible to its state at the time of the checkpoint. Much of the operating system state, such as the process ID and parent process ID, is unrestorable. However, most applications that need checkpointing are what we call "well-behaved," and do not rely on such state. **Libckpt** determines the state of the open file table at each checkpoint, and saves it as part of each checkpoint. Upon recovery, **libckpt** restores the system so that the state of open files is the same as it was at the time of the checkpoint. All other system state is neither saved nor restored by **libckpt**.

4. Processor state restoration requires that processor registers, including the program counter and stack pointers be restored to their values when the checkpoint was taken. In **libckpt**, we use `setjmp()` to store the processor state in the memory, and processor state restoration is implemented using `longjmp()`. Thus, the recovery routine never returns, and instead execution continues as an apparent "second return" from the `setjmp()` of the checkpointing routine.

With this description of recovery, the mechanics of checkpointing are straightforward: When taking a checkpoint, **libckpt** saves the processor state using `setjmp()`, and records the state of the open file table. Then the data state, consisting of the program's stack and data segments, is written to disk.

5 Experiments

In this section, we present results of checkpointing five application programs using **libckpt**. The applications are long-running FORTRAN and C programs written by scientists to run under Unix. All are typical of programs that can benefit from checkpointing for fault-tolerance.

Application	Abbreviation	Language	Running Time (mm:ss)	Maximum Checkpoint Size (Mbytes)	Checkpoint Interval (min)
Matrix Multiplication	MAT	C	15:20	4.6	2
Linear Equation Solver	SOLVE	FORTRAN	13:42	4.6	2
Cellular Automata	CELL	C	17:39	8.4	2
Shallow Water Model	WATER	FORTRAN	25:54	13.1	3
Multicommodity Flow	MCNF	FORTRAN	18:38	24.3	6

Table 1: Description of application instances

The experiments were performed on a dedicated Sparcstation 2 running SunOS 4.1.3, and writing to a Hewlett Packard HP6000 disk via NFS. The specific instances of the applications are described in Table 1. We describe the applications below:

- **Matrix Multiplication (MAT)**: This is a straightforward matrix multiplication. Two 615×615 matrices are read from disk and multiplied, and the product matrix is written to an output file.
- **Linear Equation Solver (SOLVE)**: This is a testing program from LAPACK, a high-performance package of linear-algebra subroutines [DS86]. This program generates a system of 750 equations with 750 unknowns, uses LU decomposition to solve the system, and then writes the solution to disk. It repeats this process for seven separate systems of equations.
- **Cellular Automata (CELL)**: This program executes a 2048×2048 grid of cellular automata for fifteen generations.
- **Shallow Water Model (WATER)**: This is the program `STSWM` from the National Center for Atmospheric Research. The program is a shallow water model based on the spectral transform method [HJW93]. The instance used here is "Zonal Flow over a Mountain" from their test suite, modeled at 15-minute intervals for six hours.

- **Multicommodity Flow (MCNF):** This program solves the multicommodity network flow problem using the simplex method [Ken79]. The instance used here runs on a network of 100 vertices and 50 commodities.

Note that for the purposes of this experiment, input values have been chosen to give running times between thirteen and thirty minutes. Typically, the programs would be set up to run for much longer, thus making them ideal candidates for **libckpt**.

We present results pertaining to the three important metrics of checkpointing performance:

- **Checkpoint time:** This is the average duration of a checkpoint, from start to finish.
- **Checkpoint overhead**: This is amount of time added to the running time of the application as a result of checkpointing. Note that in sequential checkpointing, checkpoint overhead is equal to the total checkpoint time. In main-memory and copy-on-write checkpointing, the checkpoint overhead is smaller than the total checkpoint time because the disk writes are performed in parallel with the execution of the application.
- **Checkpoint size**: This is the average size of the checkpoint file.

The prime goal of checkpoint optimization is to minimize all three of these metrics. Minimizing checkpoint overhead is the most important, because users would rather take the risk of failure than use a checkpointer that increases their applications' performance significantly. Keeping the overhead of checkpointing under 10% of the program's total running time is a reasonable goal [LNP90, PL94]. Minimizing checkpoint size is also important, as disk space rarely comes for free. Checkpoint time is the least important of the three metrics. When checkpointing for fault-tolerance, the only concern is that the current checkpoint complete before the user desires the next checkpoint to begin. When checkpointing for job-swapping, checkpoint time is more important as the application cannot be swapped until the checkpoint is completed.

6 Results

All of the experimental results are contained in Table 2 in the appendix. All of the graphs and data in this section are drawn directly from Table 2.

6.1 Results of Sequential Checkpointing

With no optimizations, checkpoint time and overhead should be the same, and should be directly proportional to the checkpoint size. Figure 2 confirms this prediction, showing checkpoint overhead and time vs. size for the sequential checkpointing runs described in Table 2.

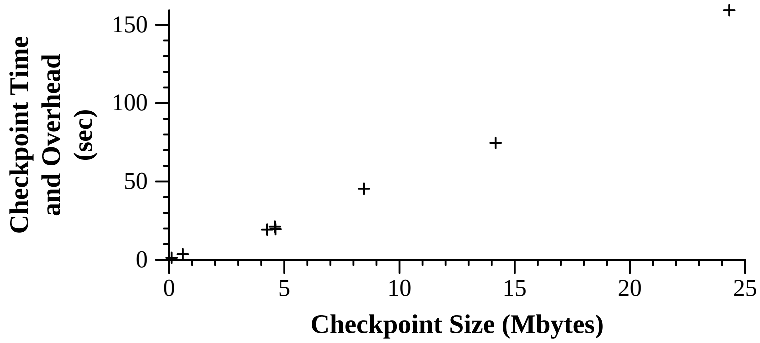

Figure 2: Checkpoint Time vs. Size for Sequential Checkpointing

6.2 Results of Checkpointing with `fork()`

When checkpointing with `fork()`, the application writes its checkpoints to disk asynchronously, meaning that it may run concurrently with the saving of the checkpoint. This reduces the overhead of checkpointing dramatically, as shown in Figure 3. This figure graphs the percentage reduction in the overhead of checkpointing by using `fork()`[2].

Because SunOS 4.1.3 implements `fork()` with copy-on-write, Figure 3 shows that copy-on-write improves the overhead of checkpointing by over 70 percent in almost all cases.

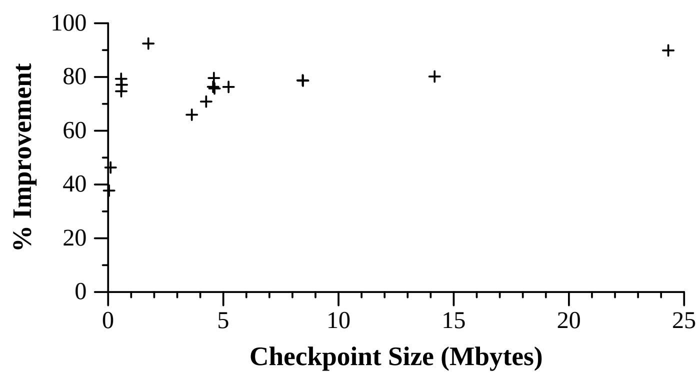

Figure 3: Percentage Reduction in Checkpointing Overhead by Using `fork()`

6.3 Results of Incremental Checkpointing

Figure 4 is a graph showing the percentage reduction of checkpoint size and checkpoint overhead when using incremental checkpointing instead of simple sequential checkpointing. In three of the applications (MAT, WATER, and MCNF), only a fraction of the applications' address spaces are modified between checkpoints, resulting in a significant reduction in the average checkpoint size. Correspondingly, the overhead of checkpointing is significantly reduced. In the other two programs, the entire address spaces of the programs are modified between checkpoints, yielding little to no reduction in the size of checkpoints. Therefore, in

[2] Ninety percent reduction in checkpoint overhead means that the overhead of checkpointing using `fork()` is ten percent of the overhead of sequential checkpointing

SOLVE and CELL, the overhead of checkpointing is *increased* due to the fact that the cost of handling page faults is not offset by a savings in the time to write the checkpoint to disk.

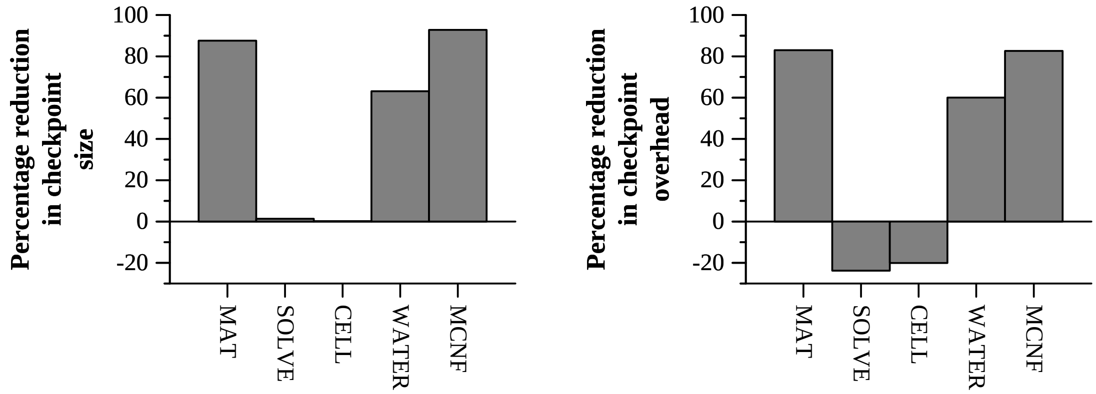

Figure 4: Percentage Reduction in Checkpoint Size and Overhead Through Incremental Checkpointing

6.4 Results of User-Directed Checkpointing

In the previous three sections, the results corroborate published research concerning checkpointing optimizations [FB89, LNP90, EJZ92, LNP94]. In this section, we evaluate the the new technique: user-directed checkpointing. In three of the applications, we analyzed the application programs and inserted directives in the code. In each case, we were able to add under ten lines of code, making checkpoints synchronous, and excluding memory from these checkpoints. We describe the details of each application below.

SOLVE: Adding directives to the Linear Equation Solver was straightforward. At the end of each iteration, all of the program's arrays are dead: The matrix of equations will be initialized anew for the next iteration, and the solution vector will be recalculated. Therefore at the end of each iteration, we insert `exclude_bytes()` calls for the equation matrix and solution vector, then a `checkpoint_here()` call, and finally `include_bytes()` calls to re-include the matrix and vector in case of an asynchronous checkpoint.

The results can be seen in Figure 5: The calls to `exclude_bytes()` and `checkpoint_here()` enable the checkpoint files to be almost empty, reducing the checkpoint size and overhead by over 90 percent. This is significant, because it is an application where incremental checkpointing fails to improve the performance of checkpointing.

CELL: At the end of each generation of the cellular automaton application, the previous value of the automaton grid becomes dead — its values are not used for the calculation of the subsequent generations of the computation. Therefore, we added user directives to checkpoint at the end of each generation, excluding the dead half of the grid from each checkpoint. In order to checkpoint at roughly the same interval as before, we also set `mintime` to 100, so that every second generation is checkpointed.

The results are in Figure 6: With user directives, the checkpoint size is halved. Accordingly, the overhead of checkpointing is also halved when user directives are employed. Thus, as

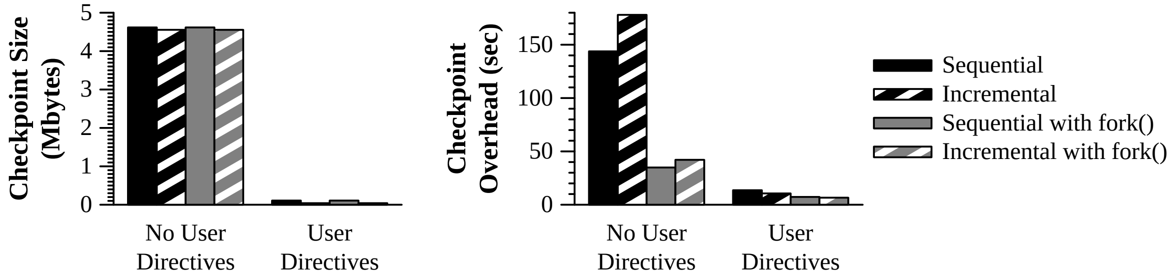

Figure 5: Results of User-Directed Checkpointing on the SOLVE Application

in SOLVE, the calls to `exclude_bytes()` and `checkpoint_here()` succeed in improving the overhead of checkpointing in an application where incremental checkpointing fails.

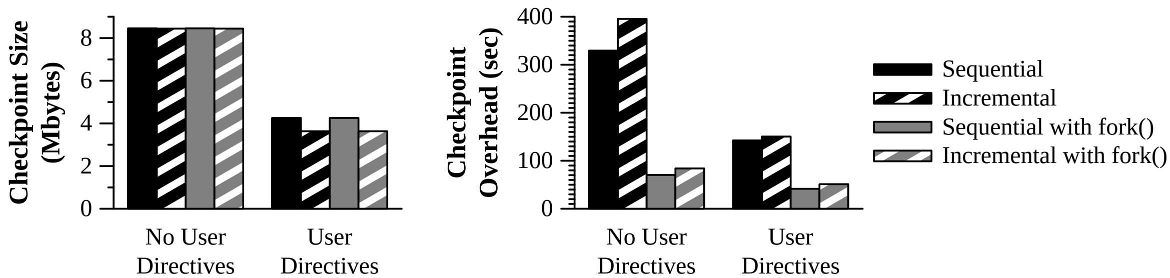

Figure 6: Results of User-Directed Checkpointing on the CELL Application

MAT: In the matrix multiplication, the two input matrices are read-only data. Moreover, once a product element is calculated, it too is read-only. This is the reason why incremental checkpointing works so well. In this application, we inserted `exclude_bytes()` calls (with *start_now* equal to 0) after reading the input matrices to mark them as read-only, and also after calculating a product row to mark that row as read-only. Thus, once a checkpoint contains these values, subsequent checkpoints omit them. The behavior of the application with these calls should approximate standard incremental checkpointing — after data becomes read-only, it is omitted from subsequent checkpoints.

The results of this experiment are in Figure 7. The important bars are the solid ones, showing that the checkpoints obtained with the user directives are approximately the same size as those obtained with incremental checkpointing. Moreover, they show slightly lower overhead, because they spend no extra time catching page faults.

7 Related Work

There has been much computer science research devoted to checkpointing. Checkpointing has been implemented on uniprocessors [LF90, LS92], multiprocessors [LNP90, LNP94], transputers [SVS94], multicomputers [PL94], and and distributed systems [EJZ92, LFS93]. Of these, only two (Condor [LS92] and Fail-Safe PVM [LFS93]) are publicly available code for Unix environments. Both implement sequential checkpointing with forking, and neither is designed for simple uniprocessor checkpointing: Condor is a system for batch programming

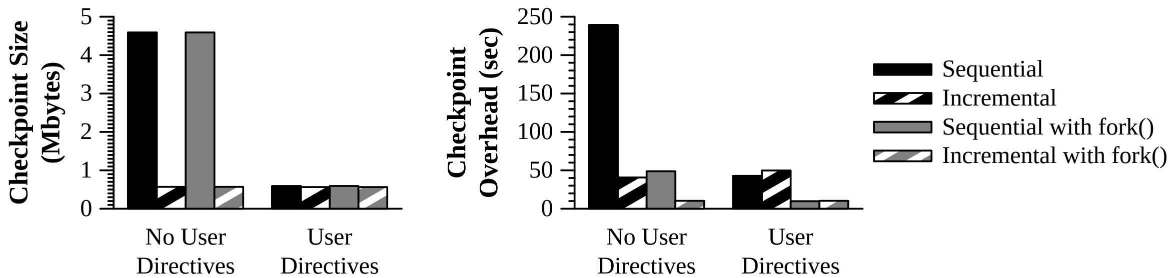

Figure 7: Results of User-Directed Checkpointing on the MAT Application

using process migration, and Fail-Safe PVM requires the programmer to have access to the PVM infrastructure. Moreover, neither package implements any optimizations beyond calling `fork()`.

User-directed checkpointing bears some similarity to checkpointers by Li and Fuchs [LF90], and Silva *et al* [SVS94]. In the former, they implement *static checkpointing*, which is the same as our synchronous checkpointing. The goal of their work is to have the user place `potential_checkpoint_here()` calls into his program, and have the compiler or runtime system decide which of those calls would be best for checkpointing. They call for no user assistance in figuring out what memory to exclude (they only exclude the stack and unallocated heap memory), and do not show the kind of dramatic performance improvements gained by user-directed checkpointing in the SOLVE, CELL, and MAT applications.

In the checkpointer of Silva *et al*, they implement a checkpointing package where the user specifies exactly what and where to checkpoint. However, the process state is not included in checkpoints, and therefore the user is also responsible for rebuilding the execution state (e.g. the call stack), although not the data, on recovery. They justify this approach by using it to implement a fault-tolerance library of grid subroutines on a transputer network. Our approach differs because the checkpointer is responsible for the entire process state, and not just the integrity of the data.

8 Conclusion

We have written a general-purpose checkpointing library, **libckpt**, that provides fault-tolerance for long-running programs under Unix. The strengths of this library are its ease of use and low overhead. **Libckpt** is currently available via anonymous FTP to `cs.utk.edu:pub/plank/libckpt`.

Our experiments with **libckpt** show first and foremost that it is general-purpose and easy to use. We were able to enable checkpointing in all five applications by changing one line of the applications' source code, and relinking with **libckpt**. Once enabled, these programs could save their state to disk periodically for fault-tolerance, using the `fork()` and incremental checkpointing optimizations if so desired. For all five applications, we were able to dramatically lower the overhead of checkpointing with copy-on-write, as implemented by **libckpt**'s `fork()` optimization. Moreover, in three of the five applications, checkpoint size and overhead were reduced by over 60 percent by incremental checkpointing. Thus, **libckpt** is able to take efficient checkpoints using standard techniques from the checkpointing literature.

Libckpt also implements user-directed checkpointing, a new technique for improving the performance of checkpointing based on the assumption that a little user input to the checkpointer can result in a large performance payoff. Memory exclusion and synchronous checkpointing are the two ways in which a user can direct the checkpointer to checkpoint more efficiently. In our experiments, a few directives were added to three of the applications, yielding performance improvements in all three cases.

One avenue of future research is to use compiler analysis to assist in user-directed checkpointing. If the user places the `checkpoint_here()` calls, the compiler can use data dependence analysis to make calls to `exclude_bytes()` and `include_bytes()`. The benefits may be twofold. First, the compiler may discover dead variables to exclude that the user may omit. Second, the compiler can guarantee that its memory exclusion will yield *correct* checkpoints. In other words, whereas the user might err in excluding too much memory from a checkpoint, resulting in a faulty recovery state, the compiler can guarantee correctness.

It is the authors' opinion that eventually checkpointing primitives such as those provided by **libckpt** should be implemented in the operating system. This will improve both the performance and the generality of checkpointing. Until such a time, users can make use of a tool such as **libckpt** to render their programs resilient to failure.

Acknowledgements

The authors thank Jian Xu, Jack Dongarra, Christian Halloy, and the National Center for Atmospheric Research for help in obtaining test programs. Kai Li is supported by ARPA and ONR under contracts N00014-91-J-4039, and Intel Supercomputer Systems Division.

References

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[DS86] J. Dongarra and D. Sorensen. Linear algebra on high performance computers. *Appl. Math. & Comp.*, 20:57–88, 1986.

[EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[FB89] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan 1989.

[FR86] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[HJW93] J. J. Hack, R. Jakob, and D. L. Williamson. Solutions to the shallow water test set using the spectral transform method. Technical Report TN-388-STR, National Center for Atmospheric Research, Boulder, CO, 1993.

[Ken79] J. Kennington. A primal partitioning code for solving multicommodity flow problems (version 1). Technical Report IEOR-79009, Southern Methodist University, 1979.

[LF90] C-C. J. Li and W. K. Fuchs. CATCH – Compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.

[LFS93] J. León, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[LNP90] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, WA, Mar 1990.

[LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[LS92] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Conference Proceedings, Usenix Winter 1992 Technical Conference*, pages 283–290, San Francisco, CA, January 1992.

[PL89] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.

[PL94] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technologies*, 2(2), 1994.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, Mass., 1992.

[SVS94] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High Performance Computing Conference*, pages 694–701, Knoxville, TN, May 1994.

[TLC85] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. In *Tenth ACM Symposium on Operating System Principles*, pages 2–11, Orchas Island Washington, December 1985.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17:8–19, June 1984.

[WM89] P. R. Wilson and T. G Moher. Demonic memory for process histories. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989.

Appendix

Application	User Directives	Incremental	Fork	Running Time (sec)	Overhead (sec)	% Overhead	Avg Ckpt Time (sec)	Avg Ckpt Size (Mbytes)	Num of Ckpts
MAT	no	no	no	1160.0	239.3	26.0	4.59	21.2	11
	no	yes	no	961.3	40.7	4.4	0.57	3.6	11
	no	no	yes	969.5	48.8	5.3	4.59	22.4	11
	no	yes	yes	931.0	10.3	1.1	0.57	3.4	11
	yes	no	no	963.5	42.8	4.7	0.59	3.6	11
	yes	yes	no	970.5	49.8	5.4	0.56	3.5	11
	yes	no	yes	930.5	9.8	1.1	0.59	5.0	11
	yes	yes	yes	931.0	10.3	1.1	0.56	3.3	11
SOLVE	no	no	no	966.5	143.8	17.5	4.62	19.6	7
	no	yes	no	1000.7	178.0	21.6	4.55	24.7	7
	no	no	yes	857.6	34.9	4.2	4.62	19.6	7
	no	yes	yes	864.8	42.1	5.1	4.55	24.7	7
	yes	no	no	836.3	13.6	1.7	0.11	1.3	7
	yes	yes	no	833.3	10.6	1.3	0.04	1.0	7
	yes	no	yes	830.0	7.3	0.9	0.11	1.4	7
	yes	yes	yes	829.3	6.6	0.8	0.04	1.1	7
CELL	no	no	no	1389.3	329.4	31.1	8.46	45.4	7
	no	yes	no	1455.4	395.5	37.3	8.44	53.0	7
	no	no	yes	1130.3	70.4	6.6	8.46	43.7	7
	no	yes	yes	1143.9	84.0	7.9	8.44	50.0	7
	yes	no	no	1202.3	142.4	13.4	4.26	19.3	7
	yes	yes	no	1210.3	150.4	14.2	3.63	20.3	7
	yes	no	yes	1101.4	41.5	3.9	4.26	19.3	7
	yes	yes	yes	1111.1	51.2	4.8	3.63	20.4	7
WATER	no	no	no	2170.4	616.5	39.7	14.17	74.6	8
	no	yes	no	1800.3	246.4	15.9	5.23	29.9	8
	no	no	yes	1676.1	122.2	7.9	14.17	74.6	8
	no	yes	yes	1612.3	58.4	3.8	5.15	28.5	8
MCNF	no	no	no	1681.8	563.6	50.4	24.31	159.3	3
	no	yes	no	1216.1	97.9	8.8	1.75	10.7	3
	no	no	yes	1175.2	57.0	5.1	24.31	131.3	3
	no	yes	yes	1125.6	7.4	0.7	1.75	10.3	3

Table 2: Results of all checkpointing experiments