Memory Exclusion: Optimizing the Performance of Checkpointing Systems

James S. Plank †

Yuqun Chen ‡

Kai Li ‡

Micah Beck †

Gerry Kingsley †

† Department of Computer Science

University of Tennessee

Knoxville, TN 37996

[plank,beck,kingsley]@cs.utk.edu

‡ Department of Computer Science

Princeton University

35 Olden Street

Princeton, NJ 08544

[yuqun,li]@cs.princeton.edu

Technical Report UT-CS-96-335

University of Tennessee

August, 1996

Submitted for publication

See http://www.cs.utk.edu/~plank/plank/papers/CS-96-335.html for up to date information

Memory Exclusion: Optimizing the Performance of Checkpointing Systems

James S. Plank* Yuqun Chen Kai Li Micah Beck Gerry Kingsley

Technical Report UT-CS-96-335
University of Tennessee
August, 1996

This paper has been submitted for publication. Please see `http://www.cs.utk.edu/~plank/plank/papers/CS-96-335.html` for up to date information concerning the publication status.

Abstract

Checkpointing systems are a convenient way for users to make their programs fault-tolerant by intermittently saving program state to disk, and restoring that state following a failure. The main concern with checkpointing is the overhead *that it adds to running time of the program. This paper describes* memory exclusion*, an important class of optimizations that reduce the overhead of checkpointing. These optimizations have been implemented in two checkpointers:* **libckpt***, which works on Unix-based workstations, and* **libNXckpt***, which works on the Intel Paragon. Both checkpointers are publicly available at no cost. We have checkpointed various long-running applications with both checkpointers and have explored the performance improvements that may be gained through memory exclusion. Results from these experiments are presented and show that the improvements are significant. We conclude that all checkpointing systems should include primitives allowing programmers and users to gain the full benefits of memory exclusion.*

1 Introduction

Checkpointing has become an increasingly important tool for uniprocessors, multiprocessors and distributed systems. It provides the backbone for fault-tolerant systems, migration systems, load-balancing systems, playback debuggers and many other functionalities [WHV+95]. Currently, the major concern with checkpointing is

*`plank@cs.utk.edu`. This material is based upon work supported by the National Science Foundation under grants CCR-9409496 and MIP-9420653, by the ORAU Junior Faculty Enhancement Award, and by DARPA under grant N00014-95-1-1144 and contract DABT63-94-C-0049.

overhead [Vai95], defined as the amount of time added to a program due to checkpointing. Experimental research has shown that the main source of overhead in all checkpointing systems is the time required to save a checkpoint to stable storage, and larger programming platforms with more processing elements and more memory simply exacerbate the problem [EJZ92, PL94].

Many techniques have been studied to reduce the overhead of saving checkpoints. These can be divided into two classes. *Latency hiding techniques* attempt to reduce or hide the overhead of disk writes, and *size reduction techniques* attempt to minimize the amount of data that gets stored per checkpoint. An important concept in size reduction is *memory exclusion*. With memory exclusion, regions of a process's memory are excluded from a checkpoint because they are either *read-only*, meaning their values have not changed since the previous checkpoint, or *dead*, meaning their values are not necessary for the successful completion of the program.

The challenge with memory exclusion is to identify the dead and read-only regions of memory with low overhead. This paper describes various techniques, both old and new, for performing this identification. Also included are performance results of implementing these techniques on a uniprocessor checkpointer, **libckpt**, running on a Sparc-2 workstation, and on a multiprocessor checkpointer, **libNXckpt**, running on the Intel Paragon. The conclusion that we draw from these implementations is that memory exclusion can often provide drastic improvements in the performance of checkpointing. These improvements are significant enough that all implementations of checkpointing should include some facilities that enable the user to gain the full benefits of memory exclusion.

2 Overview of Checkpointing

The goal of checkpointing is to establish a *recovery point* in the execution of a program, and to save enough state to restore the program to this recovery point in the event of a failure. A checkpoint of a single process is usually composed of the process's address space and the state of its registers. To recover from a checkpoint, a new process is spawned which initializes its address space from the checkpoint file and then resets its registers. This resetting of the registers (the last being the program counter) effects a restarting of the failed program.

Checkpoints of multiprocessors and distributed systems require that the processors store a global system state to stable storage. This state is composed of checkpoints for each processor in the system plus information describing the causal relationship between the checkpoints.

A *checkpointer* is a piece of code that is compiled or linked with an application and directs its checkpointing and recovery. Checkpointers strive to be as *transparent* as possible. In other words, the programmer should not have to tailor his or her code to the checkpointer. A simple recompilation or relinking plus the setting of runtime variables is all that is necessary.

For fault-tolerance, the checkpointer periodically checkpoints the application program. Upon failure, the program is restored from its most recent checkpoint, thus reducing the amount of lost work.

For further details on checkpointing, the reader is referred to papers by Tannenbaum [TL95] (for uniprocessors) and Elnozahy [EJW96] (for multi-processor systems). Throughout this paper, we refer to **libckpt**, a public-domain checkpointer for checkpointing programs on Unix-based workstations, and **libNXckpt**, an extension of **libckpt** for the Intel Paragon. Both **libckpt** and **libNXckpt** are available to the public at no cost. For more detail on **libckpt**, the reader is referred the paper by Plank *et al* [PBKL95]. For more information on **libNXckpt**, the reader is referred to the URL `http://www.cs.princeton.edu/~yuqun/checkpoint/libNXckpt.html`.

3 Overview of Memory Exclusion

Memory exclusion can be motivated best by an example. Consider the C program in Figure 1. Suppose the arrays `a` and `b` are large. If a checkpoint is taken at the program location `C1`, then the contents of of array `b` do not need to be stored because they are never used. Instead, they are computed from array `a` in the loop. Since array `b` is large, excluding it from `C1` will constitute a significant savings.

If checkpoints are taken at locations `C1` and `C2`, then the contents of array `a` do not need to be stored in checkpoint `C2` because its values have not changed since checkpoint `C1`. They can instead be retrieved from checkpoint `C1`.

```
void double_a(double a[], double b[], int n)
{
  int i;

  /* C1 */
  for (i = 0; i < n; i++) b[i] = 2.0 * a[i];
  /* C2 */

  return;
}
```

Figure 1: Memory exclusion example program

To generalize the above example, there are two distinct ways in which memory can be excluded from checkpoints:

Dead memory: If a location in memory is *dead* at the time of a checkpoint, then it does not have to be included in the checkpoint. A dead memory location is one whose value is never read following the checkpoint. Array `b` at checkpoint `C1` above is an example of dead memory.

Read-only memory: If a location in memory has not been modified since the most recent checkpoint, then as long as its value is retained on disk, it does not have to be included in the current checkpoint. Array `a` at checkpoint `C2` above is an example of read-only memory.

As will be shown in the later sections, there are many times when these two types of memory exclusion can reduce the size of checkpoint files, and therefore the overhead of checkpointing, significantly.

4 Transparent Techniques for Memory Exclusion

There are two challenges for optimizing the performance of checkpointing with memory exclusion. These are identifying the locations of memory to exclude, and maximizing the amount of memory excluded per checkpoint. There are two transparent techniques that have used to implement memory exclusion in checkpointing systems. They are outlined below:

4.1 Excluding the Code Segment and Using the Stack Pointer

Checkpoints store the address space of a processor. Typically these address spaces have four segments: executable code, global data, heap and stack. In most computer systems, the code segment is initialized by loading the program into memory. Once loaded, it is never modified. Thus, it is read-only for the lifetime of the computation and does not have to be stored in any checkpoints. Upon recovery, it can be reloaded from the executable file.

The stack segment can also benefit from memory exclusion. When taking a checkpoint, most checkpointing systems do not save the memory addresses directly below the stack pointer (this is assuming that the stack grows downward), because their current values will never be used. This is a primitive example of dead memory exclusion. Sometimes the savings from this technique can be significant if the checkpoints are taken in the right places [LSF94].

Both of these examples of memory exclusion are standard. They are employed by all known implementations of transparent checkpointing.

4.2 Incremental Checkpointing

Incremental checkpointing [FB89, WM89] uses virtual memory protection hardware to perform page-based read-only memory exclusion. While the program is executing, the checkpointer maintains a list of all pages that have been modified since the most recent checkpoint. This list can be maintained with user-level virtual memory primitives if the operating system supports them. For example, on most Unix systems, one can use `mprotect()` to set all pages to be read-only following the checkpoint, and then add a page to the changed-page list upon catching the `SEGV` signal. The page is also set to be read-write at this time. When it is time to checkpoint, only the pages on this list are stored, since the remaining pages are composed solely of read-only variables.

Incremental checkpointing can result in a significant reduction in checkpoint size and overhead if the program being checkpointed shows good locality of modification [FB89, EJZ92, PBKL95]. If the program modifies most of its pages between checkpoints, then incremental checkpointing increases the overhead of checkpointing because of the extra time it spends processing `SEGV` signals.

5 Non-transparent Techniques for Memory Exclusion

Though often effective, the above two techniques do not realize the full potential of memory exclusion. In particular, there are three weaknesses with the transparent techniques:

1. They do not exclude dead memory in the data and heap segments.
2. User-level virtual memory primitives are not available in all machines and operating systems.
3. A read-only memory location is only excluded if all the other bytes that share the same page are also read-only.

One way to attack these weaknesses is for the programmer to have some control over the memory exclusion. What follows is a description of the way programmer-directed memory exclusion has been implemented in **libckpt** and **libNXckpt**.

5.1 Excluding free memory

Many programming languages like C require the programmer to allocate and deallocate memory explicitly from the heap segment by employing procedures like `malloc()` and `free()`. Standard implementations of these procedures manage a list of free memory locations. When the program tries to allocate a region of memory, the free list is checked to see if it can provide the memory. If not, a request is made to the operating system to enlarge the heap segment, and if granted, memory is given to the program from this new area of the heap. The procedure `free()` is called to put memory back on the free list.

When these procedures are used correctly, all memory on the free list is dead. **Libckpt** and **libNXckpt** take advantage of this fact by instrumenting `malloc()`, `free()` and related procedure calls so that all memory on the free list is excluded. This is a simple technique that can enable the programmer to take advantage of dead memory exclusion in the heap.

5.2 Memory Exclusion Procedure Calls (MEPC's)

Although excluding free memory can help, there are opportunities for memory exclusion in non-free memory. For example, large portions of arrays can be read-only for long periods of time, or they can be dead at certain program locations. In order for the programmer to gain the full potential of memory exclusion, **libckpt** and **libNXckpt** allow the programmer to direct the explicit exclusion and inclusion of any regions of memory with two *memory exclusion procedure calls (MEPC's)*:

`exclude_bytes(`*char *addr, int size, int usage*`)`

`include_bytes(`*char *addr, int size*`)`

`Exclude_bytes()` tells the checkpointer to exclude the region of memory specified from subsequent checkpoints. It may be called when the programmer knows that these bytes are not necessary for the correct recovery of the program. *Usage* is an argument which may have one of two values: `READONLY` or `DEAD`. If `READONLY` is specified, then this memory is included in the next checkpoint, but excluded from subsequent checkpoints. If `DEAD` is specified, then the memory is dead — it will not be read before it is next written. Thus, it is excluded from the next and subsequent checkpoints.

`Include_bytes()` tells the checkpointer 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()`.

MEPC's allow the programmer to track memory usage as it affects checkpointing. When combined with *synchronous checkpointing*, described below, they have the potential to improve the performance of checkpointing drastically.

5.3 Synchronous Checkpointing

There are times during the execution of a program when the amount of dead memory may be very large. If MEPC's are being used, then it is most beneficial to checkpoint at these times. **Libckpt** and **libNXckpt** allow the programmer to specify such program locations with the procedure `checkpoint_here()`, which forces the checkpointer to take a checkpoint. Such checkpoints are called "synchronous" because they are not initiated by timer interrupts, and thus the programmer knows exactly when they occur. Synchronous checkpoints should be inserted by the programmer at points where memory exclusion can have the greatest effect.

Synchronous checkpoints may be placed in program locations that are reached very often or very rarely. Checkpointing too often, however, can lead to poor performance, and checkpointing too infrequently can negate the fault-tolerant benefits of checkpointing. Therefore, **libckpt** and **libNXckpt** contain two runtime parameters `maxtime` and `mintime`, that only allow synchronous checkpoints to occur within a window of time following the previous checkpoint. When that window expires, if a synchronous checkpoint has not been taken, the checkpointer forces an asynchronous checkpoint to be taken.

Specifically:

- `mintime` specifies the minimum period of time that must pass between checkpoints. If `mintime` seconds have not passed since the previous checkpoint, then `checkpoint_here()` calls are ignored.
- `maxtime` defines the maximum interval between checkpoints. At the beginning of the program, and after each checkpoint, **libckpt** calls `alarm(maxtime)` and takes an asynchronous checkpoint upon catching each `ALRM` signal. Setting the `maxtime` to zero turns off all asynchronous checkpointing.

6 Examples of Programs That Impact Memory Exclusion

In this section, we present five example programs that we checkpoint in the next section. Each program is a long-running application that can benefit from checkpointing. Moreover, each program exemplifies a different type of behavior that can affect the performance of checkpointing when memory exclusion is employed.

STSWM — A program with a large, contiguous, read-only data space

STSWM is a public-domain FORTRAN program from the National Center for Atmospheric Research that implements the "spectral transform shallow water model," an important technique in oceanic and atmospheric research. It models a complex system over several time steps and is a challenging computational problem. The main feature of **STSWM** that affects checkpointing is that it initializes a very large and contiguous data space at the beginning of the program that remains read-only for the lifetime of the program. This means that read-only memory exclusion, as implemented by incremental checkpointing, can provide great savings when this program is checkpointed.

NNET — A program with a large, non-contiguous read-only data space

This is a neural network simulation program that processes continuously generated input with a large neural network. **NNET** is typical of many graph-processing programs in its memory usage. In particular, its data structures for nodes and links contain both read-only and non-read-only components, and they are all allocated together at the beginning of the program. As such, the read-only and non-read-only variables are interleaved in memory at a relatively fine granularity. Thus, although there is great potential for read-only memory exclusion, it cannot be realized by incremental checkpointing, since most pages contain both read-only and non-read-only portions. To realize the benefits of memory exclusion, we traverse the network following its creation and insert MEPC's to mark the read-only part of nodes and links.

SOLVE — A problem that iterates over several large data sets

This is a testing program from LAPACK, a high-performance package of linear-algebra subroutines available on Netlib. The program reads input data to generate a linear system of equations represented by a square matrix of double precision floating point numbers. LU decomposition is used to solve the system, and the solution is then written to disk. This process is repeated for several sets of input data.

This program is typical of many driver programs for scientific applications — a complex procedure is executed several times on different sets of input data, and output is written at the end of each iteration. The important feature of these programs is that they share little to no information from iteration to iteration. In other words, between iterations, most of their data space is dead. For example, in **SOLVE**, the matrix, which composes the majority of the address space, is dead between iterations. Therefore, if a synchronous checkpoint is taken between iterations, then the bulk of the checkpoint may be excluded either through `malloc()/free()` or through MEPC's.

CELL — A problem with a large dead regions of memory

This is a simple program that executes a grid of cellular automata for numerous iterations. Like most cellular automaton programs, this one employs two automaton grids — a `current` grid and a `previous` grid. During a single iteration, the values of the `current` grid are calculated from the values of the `previous` grid. At the end of an iteration, the identities of the two grids are reversed so that the `current` grid becomes the `previous` grid, and vice versa.

The important feature of **CELL** as it impacts checkpointing is that between iterations, the `previous` grid becomes dead – its values are not read before they are next written. Thus, with synchronous checkpointing and MEPC's, almost half of the checkpoint may be eliminated due to dead memory exclusion.

EIGEN — A program that with little potential for improvement due to memory exclusion

EIGEN is a program that computes the eigenvalues of a general complex matrix using the `cgeev` subroutine from LAPACK. Like many subroutines from LAPACK, **cgeev** works in iterations (one per column of the matrix) and modifies almost all of the matrix during each iteration. As such, there is no significant amount of read-only or dead memory to exclude at any point in the program. **EIGEN** is included to show that there do exist programs that memory exclusion cannot help.

7 Experiments

In this section, we detail our experiments with memory exclusion on a uniprocessor checkpointer (**libckpt**) and a multicomputer checkpointer (**libNXckpt**). Each checkpointer implements incremental checkpointing and the MEPC's described above. In addition, **libckpt** implements the copy-on-write checkpointing optimization [LNP90, EJZ92], which enables the application to continue executing while the checkpoint is written to disk.

For each program, we selected input parameters that resulted in fairly long running times and the use of almost all of physical memory. Thus the programs represent a challenge to the checkpointing system. In particular, the large checkpointing sizes preclude simple buffering strategies such as taking an in-memory checkpoint, and writing that checkpoint asynchronously to disk [LNP90].

7.1 Experiments with libckpt on the SPARC-2

Each of the applications described in section 6 was compiled with **libckpt** and executed on a Sparcstation-2 containing 16 MBytes of physical memory. Checkpoints were taken via NFS over a standard Ethernet to a central file server. The disk bandwidth in this configuration is poor (around 140 Kbytes/second) but is typical

Name	Language	Parameters	Running time (sec)	Running time (h:mm:ss)	Memory Usage (Mbytes)	Checkpoint Interval (min)	# of Check-points
STSWM	Fortran	"Test 5", MM=170, TAUE=9.0	13406	3:43:26	36.8	22	10
NNET	C	22,500 nodes, 900,000 links, 187 iterations	13077	3:37:57	13.8	22	10
SOLVE	C/Fortran	Nine different 1400 X 1400 matrices	13961	3:52:41	15.1	29	8
CELL	C	2850 X 2850 grid for 85 iterations	13407	3:43:27	15.6	20	11
EIGEN	Fortran	1000 X 1000 matrix of complex doubles	14794	4:06:34	16.0	21	12

Table 1: Basic parameters for the uniprocessor applications

for many workstation environments. In all cases, the checkpoint interval is between 20 and 30 minutes. The basic parameters for each application are in Table 1.[1]

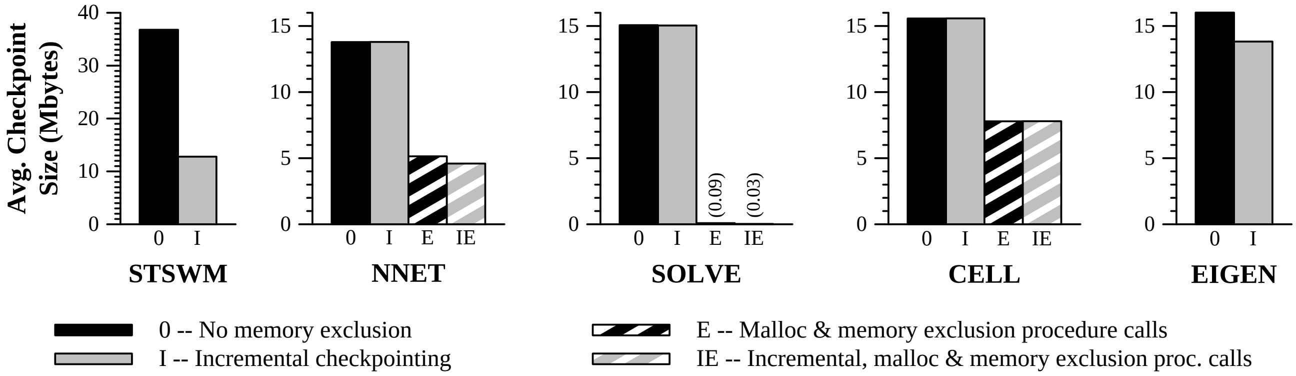

Figure 2: Checkpoint sizes of the uniprocessor applications

The impact of memory exclusion on the checkpoint size is shown in Figure 2. As expected, incremental checkpointing has a large impact on **STSWM**, but little on the other programs as they modify almost all of their pages between checkpoints. In **NNET** and **CELL**, significant portions (9.2 and 7.8 Mbytes respectively per checkpoint) of the checkpoints are excluded with MEPC's, while in **SOLVE**, almost all of the checkpoints (15.0 Mbytes) are excluded as free memory.

It should be noted that in **NNET**, incremental checkpointing with MEPC's excludes an average of 0.55 Mbytes more per checkpoint than using MEPC's without incremental checkpointing. This is because in **NNET**, many small regions of memory are excluded, requiring 0.55 Mbytes of data structures to keep track of the regions. These data structures are read-only once the MEPC's are made, and since they are all stored contiguously in memory, they can be tracked and excluded with incremental checkpointing.

The impact of memory exclusion on the checkpoint overhead is shown in Figure 3. Results for normal and copy-on-write checkpointing are given. The overheads are displayed as the average overhead per checkpoint. The

[1] Due to space constraints, we do not tabulate the data, and instead present it graphically. For the interested reader, the data is available at the URL `http://www.cs.utk.edu/~plank/plank/libckpt/raw.html`.

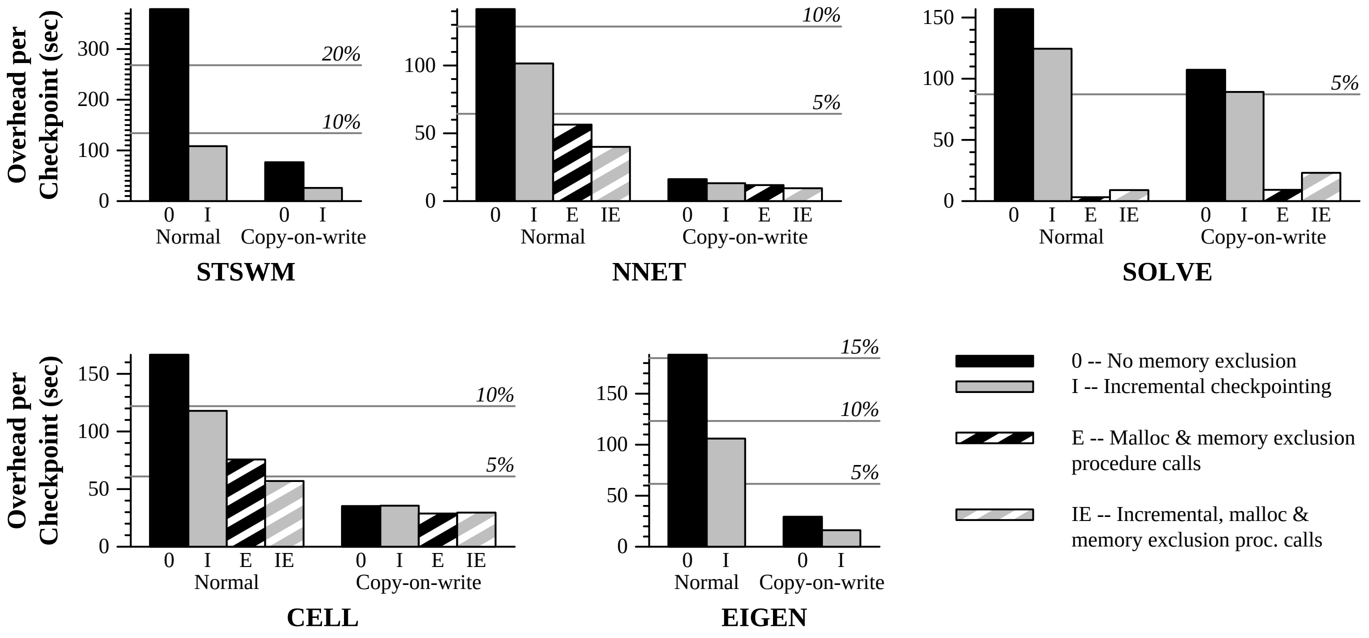

Figure 3: Checkpoint overheads of the uniprocessor applications

lines across the graphs represent the overhead as a percentage of the program's running time.[2]

As shown previously [LNP90, EJZ92, PBKL95], the copy-on-write optimization consistently improves the performance of checkpointing, though in varying degrees depending on the memory access patterns of the program. For example, although both programs modify all their pages between checkpoints, **SOLVE** has a very poor locality of modification, which penalizes the performance of copy-on-write checkpointing as compared to **CELL**.

SOLVE and **STSWM** show the most dramatic improvements in checkpoint overhead due to memory exclusion. In **SOLVE**, the per-checkpoint overhead is reduced from 107 seconds (using copy-on-write) to 3.3 (without copy-on-write) with memory exclusion, while in **STSWM**, incremental checkpointing improves the performance of copy-on-write checkpointing by 50.7 seconds per checkpoint. The other applications show more marked improvements when copy-on-write is not employed; however memory exclusion improves the performance of the copy-on-write cases too.

These results show that memory exclusion in combination with copy-on-write checkpointing can result in low-overhead checkpointing with very little programmer effort. To summarize, Table 2 displays the best combination of optimizations for each application. For the applications where extra code is inserted by the programmer, the

[2]There is an apparent anomaly in Figure 3 that should be explained. This is the fact that in **NNET**, **SOLVE** and **CELL**, incremental checkpointing *improves* the overhead even though the checkpoint sizes are the same. This should not be the case — incremental checkpointing should actually penalize performance here because since the same amount of data is being written to disk, and there is extra overhead processing `SEGV` signals. This anomaly arises because the operating system buffers user writes that do not fall on page boundaries, and **libckpt** does not take account of this fact in non-incremental checkpointing. This bug will be fixed shortly, and new data will be taken so that this anomaly is no longer present.

Name	Copy-on-write	Synchronous Checkpoints	Incremental Checkpointing	MEPC/Malloc	Lines of code added
STSWM	yes	no	yes	no	—
NNET	yes	no	yes	yes	6
SOLVE	no	yes	no	yes	1
CELL	yes	yes	no	yes	5
EIGEN	yes	no	yes	no	—

Table 2: Optimization used for the best checkpointing performance

number of additional lines of code is also noted.

7.2 Experiments with libNXckpt on the Intel Paragon

We compiled three application programs with **libNXckpt** and executed them on a 32-node Paragon at CalTech. Each node of the Paragon is an Intel 80860 processor with around 22 Mbytes of physical memory available for user processes. Unlike the Sparc-2 environment, file I/O on the Paragon is extremely fast. It performs striping with 64 Kbyte blocks to six I/O nodes and can achieve a disk bandwidth of up to 29 Mbytes per second.

LibNXckpt periodically forces the system to take a *global checkpoint* using the "Sync-and-stop" algorithm [PL94]. All processors synchronize to eliminate message state, and then they checkpoint themselves before resuming the application. **LibNXckpt** implements all varieties of memory exclusion and synchronous checkpointing; however due to problems reconstructing the network state, **libNXckpt** does not implement copy-on-write checkpointing.

Name	Language	Parameters	Running time (sec)	Running time (h:mm:ss)	Memory Usage Per Node (Mbytes)	Checkpoint Interval (min)	# of Check-points
PRISM	C	Norder=7, Nsteps=2400	7656	2:07:36	17.8	11	12
PCELL	C	17408 X 17408 grid for 800 iterations	7099	1:58:19	19.5	12	10
PSOLVE	Fortran	10 6100 X 6100 matrices	4791	1:19:51	22.2	8	10

Table 3: Basic parameters for the parallel applications

Table 3 summarizes the basic characteristics of the three applications. **PCELL** and **PSOLVE** are parallelized versions of **CELL** and **SOLVE** respectively (**PSOLVE** coming from ScaLAPACK instead of LAPACK). **PRISM** is a fluid dynamics modeling code from the Aeronautics & Applied Mathematics at CalTech. Like their uniprocessor counterparts, **PCELL** and **PSOLVE** both benefit from dead memory exclusion and synchronous checkpointing. **PRISM** is a program with a very large and contiguous read-only portion, and thus can benefit from incremental checkpointing. Additionally, **PRISM** performs a good deal of memory allocation and deallocation during each iteration, meaning that dead memory exclusion via `malloc()/free()` should be effective.

Due to the high desirability of the CalTech Paragon, the program runs are shorter than the uniprocessor runs. However, they do attempt to use most of the available physical memory so that the checkpointer is again challenged.

The checkpoint size and overhead information is displayed in Figure 4. As expected, **PCELL** and **PSOLVE** show significant space savings (306 and 548 Mbytes per checkpoint respectively) due to dead memory exclusion. This results in a corresponding decrease in checkpointing overhead for both applications. In **PRISM**, 397 Mbytes are saved per checkpoint due to incremental checkpointing, and an additional 43 Mbytes are saved by excluding free memory. Again, these savings are reflected in lower checkpoint overheads.

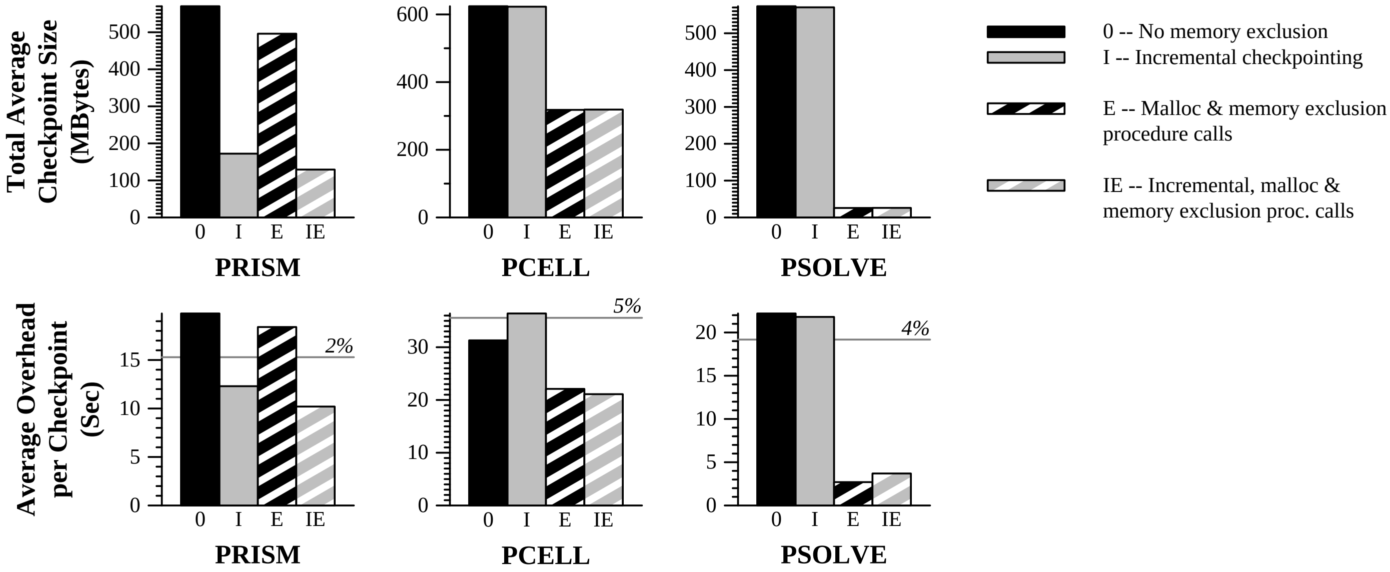

Figure 4: Checkpoint sizes and overheads of the parallel applications

In sum, the results from the Paragon mirror the uniprocessor results, although the checkpoint sizes, and therefore the savings due to memory exclusion, are much larger. The overheads, and (therefore the savings in overhead) are lower because of the faster file operations. However, percentage-wise, the savings in these applications due to memory exclusion are indeed significant.

8 More advanced memory exclusion techniques

There have been several research projects targetted at optimizing the performance of memory exclusion. Since each is too complex to describe fully, they are summarized below.

- **Compiler-assisted memory exclusion (CAME)** [PBK95]: One weakness of MEPC's is that the programmer can err. If the programmer excludes memory that is neither read-only nor dead, then the checkpoints may become invalid. With the CAME technique, the programmer inserts compiler directives into the program, telling the compiler when to take synchronous checkpoints, and when to exclude memory. The compiler uses these directives to determine what memory to exclude, and it inserts the proper MEPC's into the code. The CAME technique is advantageous because it will never err, and it can often discover more memory to exclude than the programmer.

- **Compiler-assisted full checkpointing** [LSF94]: This technique takes a collection of potential synchronous checkpoint locations and attempts to checkpoint only at the ones that maximize memory exclusion during the specified checkpointing interval. This can be done adaptively at runtime, or off-line using a "training" run. This technique may be combined with the techniques described in this paper to optimize the selection of synchronous checkpoint locations.
- **Variable-level tracking of memory exclusion** [NW94]: This is a technique for tracking read-only and dead memory at the variable level in order to minimize checkpoint sizes. To perform this tracking, memory reads and writes are monitored by executable rewriting. This enables the checkpointer to obtain optimally small checkpoint files. The monitoring incurs a very high overhead (a factor of 1.7 to 7 in their measurements), meaning that this technique is useful only in debugging applications where minimizing checkpoint size is more important than the time overhead.

9 Conclusion

Memory exclusion is an important concept in reducing the space and time overhead of checkpointing. We have presented several old and new techniques for excluding memory from checkpoints, and detailed results from two checkpointers that have implemented these techniques. The conclusions that can be drawn from these results is that memory exclusion can optimize the performance of checkpointing significantly for many long-running programs. The degree of optimization is, of course, dependent on the memory access patterns of the application. However, as shown by **NNET**, **SOLVE** and **CELL**, different kinds of access patterns may be converted into savings in checkpoint overhead with very little programmer effort.

It should be noted that while often beneficial (e.g. for **STSWM** and **PRISM**), incremental checkpointing is not sufficient for getting the most out of memory exclusion. Given the results of this paper, we believe that all checkpointing implementations should follow the lead of **libckpt** and **libNXckpt** by merging memory exclusion with memory allocation, and implementing memory exclusion procedure calls.

References

[EJW96] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. To appear, 1996.

[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, January 1989.

[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, March 1990.

[LSF94] C-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-assisted full checkpointing. *Software – Practice and Experience*, 24(10):871–886, October 1994.

[NW94] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 313–325, Orlando, FL, June 1994.

[PBK95] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under unix. In *Conference Proceedings, Usenix Winter 1995 Technical Conference*, pages 213–223, January 1995.

[PL94] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[TL95] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, February 1995.

[Vai95] N. H. Vaidya. On checkpoint latency. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, December 1995.

[WHV+95] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In *25th International Symposium on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

[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.