Compiler-Assisted Memory Exclusion for Fast Checkpointing

James S. Plank

Micah Beck

Gerry Kingsley

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

Appearing in:

IEEE Technical Committee on Operating Systems and Application Environments
Special Issue on Fault-Tolerance

Winter, 1995

See: http://www.cs.purdue.edu/homes/helal/tcos/arch/winter95/winter95.html
or
http://www.cs.utk.edu/~plank/plank/papers/IEEETCOS95.html

Compiler-Assisted Memory Exclusion for Fast Checkpointing

James S. Plank* Micah Beck Gerry Kingsley

Abstract

Memory exclusion is a powerful tool for optimizing the performance of checkpointing, however it has not been automated completely with low enough overhead. In this paper we present *compiler-assisted memory exclusion (CAME)*, a technique that uses static program analysis to optimize the performance of checkpointing. With the assistance of user-placed directives, the compiler can perform data flow analyses for dead and read-only regions of memory that can be omitted from checkpoints. The result can be a significant reduction in the size of checkpoints, thereby reducing the overhead of checkpointing.

1 Introduction

Checkpointing has become an increasingly important tool for both uniprocessor and multiprocessor systems. It provides the backbone for fault-tolerant systems, migration systems, load-balancing systems, playback debuggers and many other functionalities [14]. A major concern with checkpointing is *overhead*, 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 [3, 12].

Many techniques have been studied to reduce the overhead of saving checkpoints. These can be divided into two classes. *Latency hiding techniques*, like main-memory [12], copy-on-write [3, 8] and diskless checkpointing [11] attempt to reduce or hide the overhead of disk writes, and *size reduction techniques*, like incremental checkpointing [4, 15], compression [12] and compiler-assisted full checkpointing [7], attempt to minimize the amount of data that gets stored per checkpoint.

An important concept in size reduction is that of *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. Memory exclusion can be a powerful tool for checkpoint optimization, however there are no transparent checkpointing techniques that perform optimal or near optimal memory exclusion without either penalizing performance drastically or relying on the user for correctness.

This paper gives an overview of *compiler-assisted memory exclusion (CAME)*, a technique for automating memory exclusion for both read-only and dead memory. CAME combines user directives with static data flow analysis to optimize memory exclusion in checkpointing systems. After providing more detail on memory exclusion as a checkpointing optimization, we describe the user directives and compiler analyses comprising the CAME technique. We then present results of performing CAME manually on three example applications. The purpose of this was to test the effectiveness of CAME by hand before attempting to implement it in a real compiler. As result of these tests, we are implementing CAME in a real compiler using the SUIF toolkit [16]. We briefly detail the progress of this implementation.

2 Memory Exclusion

Incremental checkpointing [4, 15] is an example of read-only memory exclusion. With incremental checkpointing, page-protection hardware is employed to keep track of non-read-only (i.e. dirty) pages between checkpoints. Each checkpoint is composed solely of these dirty pages. For programs that exhibit significant locality, incremental checkpointing is an effective checkpointing optimization. However, there are programs that write their entire data space at fairly fine-grained intervals. For these, incremental checkpointing increases overhead because of the extra time

*Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996. [plank,beck,kingsley]@cs.utk.edu, http://www.cs.utk.edu/~plank. This material is based upon work supported by the National Science Foundation under Grant CCR-9409496, and by the ORAU Junior Faculty Enhancement Award.

required to process the page faults [3, 4, 10].

Checkpointing the stack is a primitive example of dead memory exclusion. When taking a checkpoint, most checkpointing systems (e.g. [7, 10, 13]) do not save the memory addresses directly below the stack (this is assuming that the stack grows downward), because their current values will never be used. Sometimes the savings from this technique can be significant if the checkpoints are taken in the right places [7].

Incremental checkpointing is largely successful as an automatic technique for excluding read-only memory. However, there are significant savings available to a checkpointer by excluding dead memory apart from the stack [10], yet no automatic techniques exist for finding such memory at a low cost. Netzer and Weaver have used variable-level monitoring to find read-only and dead memory, but at a cost of 1.7 to 7 times the running time of the program [9]. Li, Stewart and Fuchs have used compiler analysis to checkpoint when the stack size is small, thereby maximizing dead memory exclusion in the stack [7]. Their techniques do not address finding dead variables elsewhere in the address space, but could be combined with the CAME technique to maximize dead memory exclusion throughout a process's address space.

Finally, the **libckpt** transparent checkpointing library [10] allows users to exclude and include bytes in a process's memory space with the procedures `exclude_bytes()` and `include_bytes()`, and then checkpoint at specific code locations using the procedure `checkpoint_here()`. This enables the user to force checkpoints at code locations where memory exclusion can be maximized[1]. **Libckpt** differentiates between read-only and dead memory exclusion, because they require differing semantics. **Libckpt** has been shown to be a powerful tool with a great capacity for improving the performance of checkpointing [10]. However, it depends on the user to specify the memory exclusion correctly. If the user errs, the program will not recover to a correct state, leading to arbitrary errors in the execution of the program. This is a serious problem.

It is the mission of the CAME technique to provide an automatic method of performing both read-only and dead memory exclusion of all program variables in a safe and efficient manner.

[1] There is a settable runtime parameter `mintime` that controls the minimum time between checkpoints. `Mintime` can be used to assure that too many checkpoints are not taken over a given period of time because of closely placed `checkpoint_here()` calls.

3 The CAME Technique

The CAME technique combines standard data flow analysis with user directives, enabling the compiler to insert memory exclusion procedure calls *that are guaranteed to be safe* into a user's program. Although the compiler cannot find all variables to exclude from checkpoints, it will exclude as many as it can find, and will never generate an incorrect checkpoint.

There are two directives that the user can place into the program: `CHECKPOINT_HERE` and `EXCLUDE_HERE`. The `CHECKPOINT_HERE` directive specifies the program locations at which to checkpoint. The compiler emits a `checkpoint_here()` procedure call at each of these locations, and uses them in its analysis. The `EXCLUDE_HERE` directive is more subtle. It tells the compiler where to insert memory exclusion procedure calls. In the simplest case, one would put an `EXCLUDE_HERE` directive immediately before each `CHECKPOINT_HERE` directive. However, the placement of `EXCLUDE_HERE` directives can affect the amount of memory exclusion and it can affect the performance of checkpointing. For example, if a `CHECKPOINT_HERE` directive is placed inside a nested loop, it would be wise to place the `EXCLUDE_HERE` directive before the loop so that memory exclusion calls are made only once and not during every iteration of the loop.

Note that a bad placement of these directives may lead to excess overhead in checkpointing or not enough memory exclusion. However, the program will still checkpoint and recover correctly. This is a major difference between CAME techniques and having the user insert memory exclusion calls directly.

Compiler Analysis

We first detail our program model. A *control flow graph (CFG)* is a directed graph $G = (N, E)$ where N is a set of nodes representing statements and $E \subseteq N \times N$ is a set of directed edges representing the possible flow of control between statements [1]. The compiler first divides G into subgraphs G', where each subgraph is rooted by an `EXCLUDE_HERE` directive and contains all paths reachable from that directive that do not pass through another `EXCLUDE_HERE` directive. Note that this does not necessarily partition the graph, but merely defines a collection of subgraphs.

For each subgraph G', we will compute two sets of memory locations:

- $\mathrm{DE}(G')$ is the set of memory locations that are dead at every `CHECKPOINT_HERE` directive in G'.

Set	Update Function	
DEAD	$F_S(X) = \begin{cases} L & \text{if } S \text{ is } \texttt{END} \\ X \cup \text{MUST\_DEF}(S) - \text{MAY\_REF}(S) & \text{otherwise} \end{cases}$	
DE	$F_S(X) = \begin{cases} X \cap \text{DEAD}(S) & \text{if } S \text{ is } \texttt{CHECKPOINT\_HERE} \\ L & \text{if } S \text{ is } \texttt{EXCLUDE\_HERE} \text{ or } \texttt{END} \\ X & \text{otherwise} \end{cases}$	
RO	$F_S(X) = \begin{cases} L & \text{if } S \text{ is } \texttt{EXCLUDE\_HERE} \text{ or } \texttt{END} \\ X - \text{MAY\_REF}(S) & \text{otherwise} \end{cases}$	

Table 1: Data Flow Equations for DEAD, DE and RO

- RO(G') is the set of memory locations that are read-only throughout G'.

At each `EXCLUDE_HERE` directive, memory exclusion calls are inserted to exclude variables in DE(G') and RO(G'), and to include all others.

Thus, our analysis focuses on finding the two sets DE(G') and RO(G'). We use data flow techniques to perform this analysis. The former (finding DE(G')) employs standard liveness analysis and the latter (finding RO(G')) is a form of dependence analysis.

In order to make use of data flow techniques, we characterize the memory accesses of each statement S in the program with *reference* and *definition* sets:

- Every location that may be referenced by some execution of S is in MAY_REF(S).
- Every location that may be defined by some execution of S is in MAY_DEF(S).
- Every location that must be defined by every execution of S is in MUST_DEF(S).

These sets can be determined by local syntactic analysis while making the CFG.

Given these basic sets, we can give a definition of DE(G') and RO(G'):

1. A memory location l is *live* at a statement S if there is a path from S to another statement S' such that $l \in$ MAY_REF(S') and for every S'' on that path $l \notin$ MUST_DEF(S''). A location l is an element of DE(G') if l is dead (i.e. not live) at all `CHECKPOINT_HERE` statements in G'.
2. A memory location l is *read-only* at a statement S if $l \notin$ MAY_DEF(S). Therefore $l \in$ RO(G') if and only if $l \notin$ MAY_DEF(S) for all S in G'.

These definitions are conservative, since they look at all possible paths through the control flow graph, when some of these can never be taken by an execution of the program. Exact analysis of liveness and dirtiness are undecidable problems.

Data Flow Equations

The analysis of liveness is usually expressed as a set of data flow equations, one for each statement in the program. We will give data flow equations which enable us to determine DE(G') and RO(G') for each subgraph G' of the program. Each of these equations can be solved by a general iterative technique.

For the purposes of this paper, a data flow equation is characterized by its *update function* F_S. This is a function associated with each statement S. The function maps sets of memory locations to sets of memory locations, and characterizes the effect of executing statement S.

We illustrate the framework using the analysis of liveness as an example. We will calculate a set DEAD(S) for each statement S, which represents the set of memory locations that are dead just before executing S.

The locations that are dead just before statement S are those that are dead after statement S plus those that must be written by statement S, minus any that may be read by statement S. Thus, the update function at node S is denoted:

$$F_S(X) = X \cup \text{MUST\_DEF}(S) - \text{MAY\_REF}(S)$$

The iterative algorithm for solving equations for DEAD proceeds as follows:

1. Initially, set DEAD$(S) = L$ for all S.
2. For every statement S, compute $X = \bigcap_{S'} \text{DEAD}(S')$, the intersection of DEAD(S') for all statements S' that are successors of S in G. Then set DEAD$(S) = F_S(X) = X \cup \text{MUST_DEF}(S) - \text{MAY_REF}(S)$.

3. Iterate until a fixed point is reached for all sets DEAD(S).

There are three sets of data flow equations in CAME analysis: DEAD, DE and RO. DEAD(S) represents dead data at statement S as described above. DE(S) represents the intersection of DEAD(S') for all statements S' that are `CHECKPOINT_HERE` statements reachable from S in the same subgraph as S. DE(S) is used to propagate deadness information from the `CHECKPOINT_HERE` statements to the `EXCLUDE_HERE` statements. Finally, RO(S) represents data that is read-only in all paths from S to the end of S's subgraph.

The data flow equations for DEAD, DE and RO are given in Table 1. The iterative algorithm defined for DEAD applies to computing DE and ROas well.

Each `EXCLUDE_HERE` directive defines a new subgraph G'. The sets DE(G') and RO(G') are defined to be DE(S) and RO(S), where S is the statement *directly following* the `EXCLUDE_HERE` directive.

4 Results of CAME By Hand

To test the feasibility of the CAME technique, we performed the technique by hand on three FORTRAN programs: **CELL**, a cellular automata program, **SEIVE**, a standard prime number generator, and **CONTOUR**, a program that finds map altitude contours. In each case, we judiciously inserted one `CHECKPOINT_HERE` and one to three `EXCLUDE_HERE` directives into the program. We then calculated the CFG and the sets MAY_REF, MAY_DEF and MUST_DEF, and determined the sets DE and RO for each `EXCLUDE_HERE` directive. Finally, we placed the proper memory exclusion calls at the `EXCLUDE_HERE` directives and tested the performance using **libckpt** [10].

Program	Running Time (sec)	Address Space Size (MB)	Ckpt. Interval (min)	# of ckpts
CELL	857	8.1	2.5	6
SIEVE	1445	0.2	2.0	12
CONTOUR	1072	6.3	3.0	5

Table 2: Application program information

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 speed of disk writes in this configuration is 160 Kbytes/second.

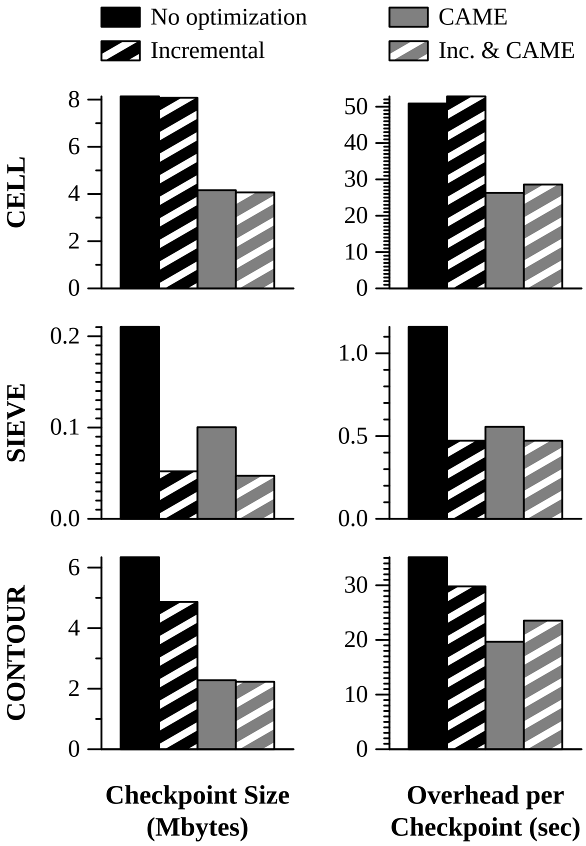

Figure 1: Results of CAME by hand

All results are averages of three or more runs of each program.

Information about each application is displayed in Table 2, and the results of checkpointing are displayed in Figure 1. A more complete analysis of this experiment is contained in [2].

The most interesting of these results is for **CELL**, which writes its entire address space between checkpoints, rendering incremental checkpointing ineffective. However, at certain points in the program, half of its memory is dead. This memory is discovered by CAME, and is excluded from the checkpoint files for a savings of almost 50% in checkpoint overhead.

In the **SEIVE** program, the main program array is dead at first, and once an element is initialized it becomes read-only. The CAME technique recognizes this, excluding an average of 110 KB per checkpoint. Incremental checkpointing saves another 53 KB by excluding read-only pages allocated to system data structures such as buffers in the standard I/O library.

Like the **CELL** program, the **CONTOUR** program does not lend itself to large performance improvements due to incremental checkpointing. This is because the granularity of page updates is small. If just one grid point is updated in a page, then the whole page must be included in an incremental checkpoint. CAME traces read-only data at the variable level, which allows for a 64% improvement in checkpoint size over sequential checkpointing, as opposed to a 23% improvement due to incremental checkpointing.

Note that the overhead of checkpointing is only improved by 44%. This is because `exclude_bytes()` and `include_bytes()` are called a total of 237,741 times during the course of the program. Thus the calls themselves add a non-trivial amount of overhead to checkpointing. However, this is more than offset by the savings of writing a smaller checkpoint file to disk.

The conclusion that we draw from this by-hand implementation is that CAME has the potential to be a powerful technique for checkpoint optimization, and is worth the work required to implement in a real compiler.

5 Implementation in SUIF

We are completing an implementation of CAME using the Stanford University Intermediate Form (SUIF) toolkit [16]. SUIF is a preprocessor for FORTRAN that facilitates performing data flow analyses. We have successfully implemented CAME for scalar variables and are nearing completion of an implementation of CAME for rectangular array regions using interval analysis [1, 5, 6]. When this implementation is finished, we will be able to draw more complete conclusions on the effectiveness of the CAME technique, including the impact of judicious placement of `EXCLUDE_HERE` directives.

6 Conclusions

In this paper, we have presented a compiler-assisted technique, CAME, for the static analysis of safe memory exclusion in checkpointing. We have expressed exclusion analysis as the solution of a set of data flow equations, using a general iterative method to solve them. We have implemented our technique by hand and have demonstrated its effectiveness in reducing checkpoint size and overhead in three example programs. We are near completion of an implementation of this technique for general FORTRAN programs.

This technique is significant as the only automatic checkpointing technique that performs dead memory exclusion with low overhead. Moreover, it can track memory exclusion at the variable level, outperforming page-based checkpointing strategies like incremental checkpointing in some cases.

The CAME technique can be combined with other checkpointing optimizations to improve the performance of checkpointing further. For example, the CAME technique fits hand-in-hand with all latency-hiding optimizations, such as copy-on-write checkpointing, and also with incremental checkpointing, as shown by the results with **libckpt**. The CAME technique may also be combined with the compiler-assisted full checkpointing technique of Li, Stewart and Fuchs so that the selection of `checkpoint_here()` calls at which to checkpoint may also be optimized [7].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] M. Beck, J. S. Plank, and G. Kingsley. Compiler-Assisted Checkpointing. Univ. of Tenn. Tech. Rep. CS-94-269, 1994.

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

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

[5] E. D. Granston and A. V. Veidenbaum. Detecting redundant access to array data. In *Supercomputing '91*, pp. 854–865, Nov 1991.

[6] T. Gross and P. Skeenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software – Practice & Experience*, 20(2):133–155, Feb 1990.

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

[8] 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, Aug 1994.

[9] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN PLDI*, pp. 313–325, June 1994.

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

[11] J. S. Plank and K. Li. Faster checkpointing with $N+1$ parity. In *24th Int. Symp. on Fault-Tol. Comp.*, pp. 288–297, June 1994.

[12] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Par. & Dist. Tech.*, 2(2):62–67, 1994.

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

[14] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In *25th Int. Symp. on Fault-Tol. Comp.*, pp. 22–31, June 1995.

[15] P. R. Wilson and T. G Moher. Demonic memory for process histories. In *ACM SIGPLAN PLDI*, pp. 330–343, June 1989.

[16] R. P. Wilson *et al.* SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec 1994.