Compiler-Assisted Checkpointing[1]

Micah Beck, James S. Plank and Gerry Kingsley
Department of Computer Science
University of Tennessee, Knoxville, TN 37996
{beck, plank, kingsley}@cs.utk.edu

Abstract

In this paper we present *compiler-assisted checkpointing*, a new technique which uses static program analysis to optimize the performance of checkpointing. We achieve this performance gain using **libckpt**, a checkpointing library which implements *memory exclusion* in the context of *user-directed* checkpointing, The correctness of user-directed checkpointing is dependent on program analysis and insertion of memory exclusion calls by the programmer. With compiler-assisted checkpointing, this analysis is automated by a compiler or preprocessor. The resulting memory exclusion calls will optimize the performance of checkpointing, and are guaranteed to be correct. We provide a full description of our program analysis techniques and present detailed examples of analyzing three FORTRAN programs. The results of these analyses have been implemented in **libckpt**, and we present the performance improvements that they yield.

1 Introduction

Checkpointing is an important method for providing fault tolerance in general-purpose computing environments. Checkpointers have been implemented for uniprocessors [LF90, LS92, PBKL95] and all varieties of parallel computing environments [EZ92, LFS93, LNP94, PL94b, SVS94]. All experimental research on checkpointing has found that the main source of overhead is the time required to write a checkpoint to disk.

Many optimizations are found in the literature which attempt to reduce this source of overhead. They can be divided into two classes. *Latency hiding* optimizations attempt to hide or eliminate the latency of disk writes. *Memory exclusion* optimizations attempt to minimize the amount of memory that needs to be saved at each checkpoint.

Until recently, the only memory exclusion optimization that has been successful in improving performance has been *incremental checkpointing* [FB89, WM89], which eliminates the need to save memory that is *read-only* between checkpoints. However, another source of memory exclusion is memory that is *dead* at the time of checkpointing. There have been mechanisms implemented to exploit *both* read-only and dead memory exclusion in checkpointing, but they suffer either from too much overhead tracking memory usage [NW94] or from too much dependence on the programmer [PBKL95].

In this paper we present *compiler-assisted checkpointing*, a new technique which uses static program analysis to optimize the performance of checkpointing. This enables the user to gain the benefits of memory exclusion without memory-tracking overhead and without burdening the programmer. This optimization is orthogonal to the latency-hiding checkpointing optimizations, and can be combined with them to obtain checkpointing with very low overhead.

[1] James Plank is supported by the National Science Foundation under contract CCR-9409496.

We describe an implementation of compiler-assisted checkpointing that uses **libckpt**, a checkpointing library that implements user-directed memory exclusion [PBKL95]. We show how compiler-assisted checkpointing generates safe and efficient memory exclusion calls in three example FORTRAN programs, and present the results of checkpointing this programs. The end result is that compiler-assisted checkpointing can be very helpful at optimizing the performance of checkpointing. Moreover, it is the only mechanism known that can exclude dead memory from checkpoints both safely and efficiently.

2 Background and Related Work

A simple *sequential* checkpointer works by freezing the execution of a running process periodically and saving its entire execution state, including the contents of its data memory and registers, to disk. Recovering from such a checkpoint is straightforward: the values of memory and registers are restored from the disk file. The overhead of sequential checkpointing is the time it takes to write the checkpoint file to disk. This time is proportional to the size of the program's address space.

Sequential checkpointing of multiprocessors is similar, except that there are multiple register sets and often multiple memories to save. In message passing systems there is often a network state to be saved and synchronization issues to be considered. However, the major source of overhead in these systems is still the time it takes to write checkpoints to disk; the overhead of saving network state and synchronizing is secondary [EJZ92, EZ94, PL94b].

Current techniques for optimizing the performance of checkpointing work by trying to hide, minimize or eliminate the effect of disk writes. We discuss them below.

2.1 Latency-Hiding Optimizations

We divide checkpointing optimizations into two classes. The first of these are called *latency hiding*, and work by hiding or eliminating the latency of writing a checkpoint to disk. The following checkpointing optimizations are based on latency hiding.

Main memory/Copy-on-write Checkpointing [LNP94]: A copy of the checkpoint is made in memory, and this copy is written asynchronously to disk by another process. This allows disk writes to be overlapped with the execution of the application program and reduces the overhead imposed by checkpointing drastically. Copy-on-write allows the process writing the checkpoint to access the same memory as the target program: a page is copied only if it is modified before being written to disk. Main memory/copy-on-write checkpointing works extremely well at reducing checkpoint overhead unless memory is restricted [LNP90, EJZ92, PL94b, PBKL95].

Checkpoint Compression [LF90, PL94b]: Here the checkpoint is compressed before being written to disk. Any fast compression algorithm [Wel84, BJLM92] can be used. Compression only improves the overhead of checkpointing if the extra time taken to compress the checkpoint is less than the time saved in writing a smaller checkpoint file. In other cases compression increases checkpoint overhead. Checkpoint compression has only been shown to be beneficial in parallel environments with contention for secondary storage [PL94b].

Diskless Checkpointing [PL94a]: Instead of saving checkpoints to disk, one can use extra processors to save redundant information so that any one processor may fail and the system can still run. This eliminates disk writing as the major overhead in checkpointing, and places the burden on the network. Unless memory updates by the application program exhibit good locality, diskless checkpointing requires a large amount of extra memory to improve performance.

2.2 Memory Exclusion Optimizations

Checkpointing optimizations based on *memory exclusion* improve the performance of checkpointing by omitting read-only and dead portions of memory from each checkpoint. Read-only memory is defined to be memory that has not been modified since the previous checkpoint. It does not need to be saved as part of a checkpoint because it may be recovered from an earlier checkpoint. Dead memory is defined to be memory that will be written before it is next read. It does not need to be saved as part of a checkpoint because the values of such memory will never be needed. The following are checkpointing optimizations based on memory exclusion.

Incremental Checkpointing [FB89, WM89]: With incremental checkpointing, page protection hardware is used to identify *dirty* pages that have been modified since the last checkpoint. Only the dirty pages are saved in each checkpoint file, thereby omitting all pages of read-only memory. Upon recovery, the checkpointed state is rebuilt from all of the checkpoint files. Incremental checkpointing can reduce overhead dramatically if the target program exhibits good locality. For programs with bad locality, incremental checkpointing can degenerate to the sequential case, saving the entire data memory. In this case the cost of catching page faults actually increases the checkpointing overhead [FB89, EJZ92, PBKL95]. One extra cost of incremental checkpointing is increased use of disk space, since old checkpoint files cannot be discarded.

Word-level Memory Exclusion [NW94]: Here every read and write in the program is tracked so that both read-only and dead memory can be excluded from checkpoint files. This leads to optimally (or near optimally) small checkpoint files. Sophisticated techniques have been developed for performing this tracing with runtime overheads of 175 to 700 percent. Word-level memory exclusion is useful for playback debugging, where high runtime overheads are justified by the reduced storage requirements for checkpoint files. For fault-tolerance however the overhead of word-level memory exclusion is too large.

User-Directed Checkpointing [PBKL95]: All of the optimizations described above are automatic. They can be implemented as part of a checkpointing library, runtime system or operating system, and require no effort on the part of the user to employ. With user-directed checkpointing, procedure calls are provided for the user to improve the performance of checkpointing through memory exclusion. Specifically, there are procedure calls for excluding bytes of memory from checkpoints because they are dead or read-only. Moreover, the user can specify program locations at which to checkpoint. This is useful for two reasons. First, there are certain code locations (e.g.

the end of loops and subroutines) where the number of dead variables can be maximized. Thus by specifying to checkpoint at these locations, the user can maximize the effect of memory exclusion. Second, by checkpointing only at known code locations, the task of determining dead and read-only variables is simplified.

User-directed checkpointing has been implemented implicitly in checkpointers where the user is responsible for enacting recovery [LB87, SVS94]. It is implemented explicitly as part of the automatic uniprocessor checkpointer **libckpt** [PBKL95]. In **libckpt**, user-directed checkpointing has been shown to be quite effective in reducing the overhead of checkpointing for some programs with very little programmer effort.

The main drawbacks of user-directed checkpointing are that it can require a great deal of user effort and it is *unsafe*. For some programs, specifying checkpoint locations and memory exclusion is easy. For others, it requires patience and effort to identify the best points to checkpoint and the variables to exclude. Moreover, if the user makes a mistake and excludes memory that is not read-only or dead, then the resulting checkpoint will recover to an incorrect state. For this reason, user-directed checkpointing must be utilized with caution.

3 Compiler-Assisted Checkpointing

Compiler-assisted checkpointing is a technique first used by Li and Fuchs. In compiler-assisted checkpointing, static program analysis is used to assist in the optimization of checkpointing [LF90]. Li and Fuchs use compiler-assisted checkpointing to help the checkpointer choose at which points to checkpoint. In this paper, we use compiler-assisted checkpointing to help automate user-directed memory exclusion.

As we have discussed, an analysis of dead and read-only memory is required in order to employ user-directed checkpointing correctly. With the help of safe programmer directives, sets of dead and read-only locations can be can be automatically derived by a compiler or preprocessor. We will express these sets as the solution to a collection of data flow equations which can be solved efficiently using standard techniques [ASU86]. *Compiler-assisted checkpointing* uses this automatically generated information to generate memory exclusion calls that are safe: the checkpoints it generates are guaranteed to be correct.

3.1 Memory Exclusion in libckpt

We use the interface to user-directed memory exclusion provided by **libckpt** [PBKL95]. This interface consists of three procedure calls implemented by the **libckpt** runtime library. To implement memory exclusion, **libckpt** maintains three lists of memory regions: `I` (include), `E` (exclude) and `P` (pending exclusion). Each word of memory has an entry in exactly one of these lists. When **libckpt** takes a checkpoint, it writes all the memory in the `I` and `P` lists to disk, and then moves all elements of the `P` list to the `E` list. The user can manipulate these lists directly with two procedure calls:

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

`exclude_bytes()` tells **libckpt** to exclude the region of memory specified from subsequent checkpoints. It is called when the user 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: `CKPT_DEAD` or `CKPT_READONLY`. If the former, then the memory is moved to the `E` list — **libckpt** will exclude it from all subsequent checkpoints. If `CKPT_READONLY` is specified, then the destination list of the memory depends upon which list the memory currently resides. If the memory is on the `I` list, then it is moved to the `P` list. If it is on the `P` or `E` lists, then it remains there. Thus, unless the memory is already excluded from current checkpoints, it is included in the next checkpoint, but excluded from subsequent checkpoints.

`include_bytes()`tells **libckpt** to move the specified bytes to the `I` list so that they are included in the next and subsequent checkpoints.

In addition, **libckpt** provides a procedure `checkpoint_here()`that allows the user to specify program locations at which to checkpoint. There is a runtime variable `mintime` that can be set so that checkpoints are not taken if `checkpoint_here()` is called and `mintime` seconds have not past since the previous checkpoint.

3.2 Compiler Directives

The goal of compiler-assisted checkpointing is for the programmer to place *compiler directives* into their code, and have the compiler insert `exclude_bytes()`, `include_bytes()`, and `checkpoint_here()` calls into the code based on static program analysis. We introduce two such compiler directives: `EXCLUDE_HERE` and `CHECKPOINT_HERE`.

1. An `EXCLUDE_HERE` directive tells the compiler to compute memory exclusion information and insert memory exclusion calls at the that program location.
2. The `CHECKPOINT_HERE` directive indicates the program locations at which checkpoints can be taken. Our program analysis assumes that checkpoints are taken only at at these points. This simplifies the analysis. This assumption can be removed, however, by treating every statement as if it were followed by a `CHECKPOINT_HERE` directive.

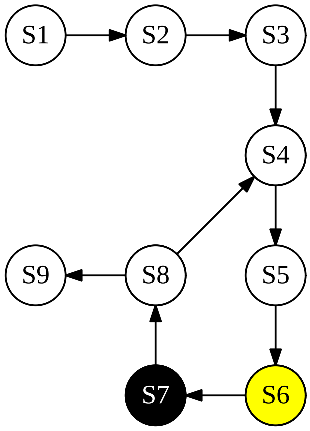
The user should place these directives in code locations that lead to the best use of memory exclusion. However, if the user errs, the result is *not* incorrect checkpointing, like it would if the user placed memory exclusion calls incorrectly. Instead, the result is simply unoptimized checkpointing.

3.3 Analysis of Memory Exclusion

In order to define our data flow equations, we must give a more detailed account of our program model. A *control flow graph (CFG)* is a directed graph $G = \langle N, E \rangle$ where N is a set of nodes

```
          PROGRAM SIMPLE
          INTEGER I, X, Y, Z

S1:       Z = 3
S2:       X = 5
S3:       FOR 100, I = 1,1000
S4:            Y = X + Z
S5:            X = X * Y
S6: C          EXCLUDE_HERE
S7: C          CHECKPOINT_HERE
S8: 100   CONTINUE
S9:       END
```

(a) Example FORTRAN program



(b) Control Flow Graph

Statement	MAY_REF	MAY_DEF & MUST_DEF	Initial Values DEAD	Initial Values RO	Initial Values DE	Final Values DEAD	Final Values RO	Final Values DE
S1	$\emptyset$	$\{Z\}$	L	L	L	L	$\emptyset$	L
S2	$\emptyset$	$\{X\}$	L	L	L	$\{I, X, Y\}$	$\{Z\}$	L
S3	$\emptyset$	$\{I\}$	L	L	L	$\{I, Y\}$	$\{Z\}$	L
S4	$\{X, Z\}$	$\{Y\}$	L	L	L	$\{Y\}$	$\{I, Z\}$	L
S5	$\{X, Y\}$	$\{X\}$	L	L	L	$\emptyset$	$\{I, Y, Z\}$	L
S6	$\emptyset$	$\emptyset$	L	L	L	$\{Y\}$	L	L
S7	$\emptyset$	$\emptyset$	L	L	L	$\{Y\}$	$\{Z\}$	$\{Y\}$
S8	$\{I\}$	$\{I\}$	L	L	L	$\{Y\}$	$\{Z\}$	L
S9	$\emptyset$	$\emptyset$	L	L	L	L	L	L

(c) Initial values and final solutions to the data flow equations

Figure 1: An Example of Memory Exclusion Analysis

representing statements and $E \subseteq N \times N$ is a set of directed edges representing the possible flow of control between statements [ASU86]. Figures 1 (a) and (b) show an example FORTRAN program and its control flow graph. In this and all other CFG's, the **EXCLUDE_HERE** nodes are shaded in gray, and the **CHECKPOINT_HERE** nodes are colored black.

The program analysis works as follows. It divides the control flow graph 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. (This does not partition the graph. Instead it merely defines a collection of subgraphs). For each subgraph G', we compute two sets of memory locations: DE(G') and RO(G'). DE(G') is the set of memory locations that are dead at every **CHECKPOINT_HERE** directive in G'. RO(G') is the set of memory locations that are read-only throughout G'. At each **EXCLUDE_HERE** directive, the following procedure calls are inserted by the compiler:

- `exclude_bytes(`*lstart*`,` *lsize*`, CKPT_DEAD)` for all the memory locations $l \in \text{DE}(G')$, where $l = [lstart, lstart + lsize]$.
- `exclude_bytes(`*lstart*`,` *lsize*`, CKPT_READONLY)` for all the memory locations $l \in \text{RO}(G')$.
- `include_bytes(`*lstart*`,` *lsize*`)` for all the memory locations l that are in neither $\text{DE}(G')$ nor $\text{RO}(G')$.

Thus, our analysis focuses on finding the two sets $\text{DE}(G')$ and $\text{RO}(G')$. We use data flow techniques to perform this analysis. The former (finding $\text{DE}(G')$) is standard liveness analysis. The second is an analysis that is unique to compiler-assisted checkpointing.

In order to make use of data flow techniques, we characterize the memory accesses of each statement in the program with *reference* and *definition* sets. An execution of a statement operates by reading (or referencing) some set of memory locations, performing a computation, and then writing the results to (or defining) another set of locations. Associated with each statement $S \in N$ is a reference set $\text{MAY\_REF}(S)$ and two definition sets $\text{MUST\_DEF}(S)$ and $\text{MAY\_DEF}(S)$:

1. Every location that *may be* referenced by *some* execution of S is in $\text{MAY\_REF}(S)$.
2. Every location that *may be* defined by *some* execution of S is in $\text{MAY\_DEF}(S)$.
3. Every location that *must be* defined by *every* execution of S is in $\text{MUST\_DEF}(S)$.

All of these sets can be approximated conservatively: the set of all locations L is a conservative approximation to $\text{MAY\_REF}(S)$ and $\text{MAY\_DEF}(S)$, and the empty set is a conservative approximation of $\text{MUST\_DEF}(S)$. It is always true that $\text{MUST\_DEF}(S) \subseteq \text{MAY\_DEF}(S)$; in our examples, the two sets are always equal. Figure 1(c) shows the initial MAY_REF, MAY_DEF and MUST_DEF sets for our example program.

Given these basic sets, we can give a definition of $\text{DE}(G')$ and $\text{RO}(G')$:

1. A location l is *live* at a statement S if there is a path from S to another statement S' such that $l \in \text{MAY\_REF}(S')$ and for every S'' on that path $l \notin \text{MUST\_DEF}(S'')$. This characterization reflects the fact that the value of a live location may be read before it is written on some execution path. A location l is an element of $\text{DE}(G')$ if l is not live at all `CHECKPOINT_HERE` statements in G'. If there are no `CHECKPOINT_HERE` statements in G', then $\text{DE}(G') = \emptyset$.
2. A location l is *read-only* at a statement S if $l \notin \text{MAY\_REF}(S)$. Therefore $l \in \text{RO}(G')$ if and only if $l \notin \text{MAY\_REF}(S)$ for all S in G'.

These structural characterizations 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.

3.4 Data Flow Equations

The sets MAY_REF, MAY_DEF and MUST_DEF can be determined by local syntactic analysis. 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 $\text{DE}(G')$ and $\text{RO}(G')$ 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 locations to sets of locations, and characterizes the effect of executing statement S.

We will illustrate the framework using the analysis of liveness as an example. We will calculate a set $\text{DEAD}(S)$ 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

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

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

1. Initially, set $\text{DEAD}(S) = L$ for all S.
2. For every node S
   (a) compute $X = \bigcap_{S'} \text{DEAD}(S')$, the intersection of $\text{DEAD}(S')$ for all statements S' that are successors of S in the CFG, and
   (b) set $\text{DEAD}(S) = X \cup \text{MUST\_DEF}(S) - \text{MAY\_REF}(S)$.
3. Iterate until a fixed point is reached for all sets $\text{DEAD}(S)$.

We use three sets of data flow equations in our analysis: DEAD, DE and RO. $\text{DEAD}(S)$ represents dead data at statement S as described above. $\text{DE}(S)$ represents the intersection of $\text{DEAD}(S')$ for all statements S' that are `CHECKPOINT_HERE` statements reachable from S in the same subgraph as S. $\text{DE}(S)$ is used to propagate deadness information from the `CHECKPOINT_HERE` statements to the `EXCLUDE_HERE` statements. Finally, $\text{RO}(S)$ 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 Figure 2. The iterative algorithm defined for DEAD applies also to computing DE and RO. Figure 1(c) shows the initial values of these three sets for each statement of the example program, and the fixed point solution to these data flow equations at each statment.

Each `EXCLUDE_HERE` directive defines a new subgraph G'. The sets $\text{DE}(G')$ and $\text{RO}(G')$ are defined to be $\text{DE}(S)$ and $\text{RO}(S)$, where S is the statement *directly following* the `EXCLUDE_HERE` directive. In the example program in Figure 1, this is the statement S7: $\text{DE}(S7) = \{Y\}$, $\text{RO}(S7) = \{Z\}$. Thus at

<table>
<tr><th>Set</th><th colspan="2">Update Function</th></tr>
<tr><td>DEAD</td><td>$F_S(X) =$</td><td>$\begin{cases} L & \text{if } S \text{ is } \texttt{END} \\ X \cup \text{MUST\_DEF}(S) - \text{MAY\_REF}(S) & \text{otherwise} \end{cases}$</td></tr>
<tr><td>DE</td><td>$F_S(X) =$</td><td>$\begin{cases} X \cap \text{DEAD}(S) & \text{if } S \text{ is } \texttt{CHECKPOINT\_HERE} \\ L & \text{if } S \text{ is } \texttt{EXCLUDE\_HERE} \\ L & \text{if } S \text{ is } \texttt{END} \\ X & \text{otherwise} \end{cases}$</td></tr>
<tr><td>RO</td><td>$F_S(X) =$</td><td>$\begin{cases} L & \text{if } S \text{ is } \texttt{EXCLUDE\_HERE} \\ L & \text{if } S \text{ is } \texttt{END} \\ X - \text{MAY\_DEF}(S) & \text{otherwise} \end{cases}$</td></tr>
</table>

Figure 2: Data Flow Equations for DEAD, DE and RO

the `EXCLUDE_HERE` statment `exclude_bytes()` will be called for Y (with *usage* set to `CKPT_DEAD`) and Z (with *usage* set to `CKPT_READONLY`), and `include_bytes()` will be called for the rest of the program locations.

3.5 Implementation Details

In our implementation of the data flow equations, simple sets of locations are replaced by rectangles whose corners may be defined in terms of loop indices. This generalization allows us to represent array regions that arise when analyzing loops. For example, a region of a two-dimensional array `A` extending from the corner `A(1,1)` to the element `A(I, J+1)` where `I` and `J` are loop indices is written `A(1:I,1:J+1)`. Such a parameterized rectangle represents array references in a representative iteration, and allows us to calculate the changes in memory exclusion between iterations.

We generalize primitive set operations, such as union and intersection, to operate on sets of parameterized rectangles. Our technique is both efficient and exact for the examples given in this paper, and for a larger class of similar problems. In some cases exact analysis is too expensive or impossible using this technique, and we settle for a conservative approximation. The problem of applying data flow techniques to array references with the greatest possible accuracy and efficiency is the subject of current research [DGS93, MAL93].

Our implementation is intended to prove the feasibility of our compiler-based techniques, and we have made several important restrictions:

- More general techniques for characterizing references to array regions have been reported in the literature [DGS93, MAL93]. The use of these more general techniques would improve the generality of our automatic analysis, but would not yield better results in any of our examples.

- Interprocedural data flow analysis is significantly more complex than intraprocedural analysis.

One approach is to make conservative assumptions about subroutine calls, letting MAY_REF = MAY_DEF = L and MUST_DEF = ϕ. This is the simplest approach which always yields a safe, albeit non-optimal result. More complex approachs include subroutine inlining [CHT91] and construction of a interprocedural summary graph [Cal88]. For the purposes of the analyses in this paper all subroutine calls have been inlined.

- Substantial research has been devoted to improving the efficiency of solving data flow equations [CCF91, JP93]. These approaches are complex to implement and do not improve the accuracy of the solution found. For these reasons, we have chosen to use the classic iterative technique.

We are currently implementing our approach in a FORTRAN preprocessor using the Stanford University Intermediate Form (SUIF) toolkit.

4 Examples of Exclusion Analysis

In this section we present three example programs and identify sets of dead and clean locations. These sets are obtained by solving the data flow equations described in Section 3.

4.1 CELL : Cellular Automata

CELL is a program that simulates the execution of a grid of cellular automata. The code has the structure shown in Figure 3(a), which maps to the control flow graph in Figure 3(b).

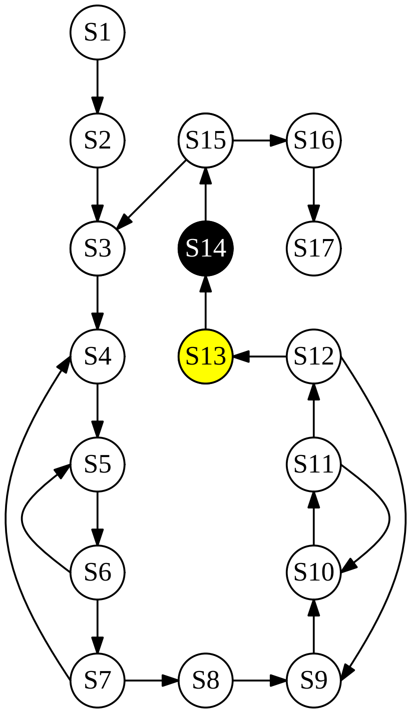
An initial pattern of cell values is read to array `G0`. Simulation proceeds in generations: in each generation an *update function* is evaluated at each grid location that determines the new value of that location. Two generations of updates are computed in every iteration of the outer loop. First `G1` is calculated as a function of `G0`, then `G0` is calculated as a function of `G1`. This is depicted graphically in Figure 4. At the end of each iteration, `G1` is dead.

The boundaries of these arrays are a special case. The update function at a given grid location is a function of its own current value and the values of its eight *neighbors* or adjacent cells. The update function is not defined on the boundary of the grid, so the boundary is never updated. For this reason, the boundaries of `G0` and `G1` are read-only after they have been initialized.

Memory exclusion is calculated at the bottom of the main loop, just before a checkpoint is taken. Figure 3(c) shows the values of the reference and definition sets, and Figure 3(d) shows the values of DE(S14) and RO(S14). The calculation of DE(S) and RO(S) for all statements S is included in the appendix; for brevity in Figures 3, 5, and 6, we just include the values that lead to memory exclusion. Figure 3(d) tells us that when checkpoints are taken, the interior of `G1` is dead. Moreover, during each iteration, the boundaries of `G0` and `G1` are read-only. Therefore, we can place the proper `exclude_bytes()` and `include_bytes()` calls at the `EXCLUDE_HERE` directive, and a `checkpoint_here()` call at the `CHECKPOINT_HERE` directive.

```
          PROGRAM CELL
          INTEGER I, ROW, COL
          INTEGER G0(1002, 1002)
          INTEGER G1(1002, 1002)

 S1:      CALL GRID_INIT(G0, G1)

 S2:      DO 200 I = 1, 60
 S3:         DO 100 R = 2, 1001
 S4:           DO 110 C = 2, 1001
 S5:             G1(R, C) = F(G0, R, C)
 S6: 100       CONTINUE
 S7: 110     CONTINUE

 S8:      DO 130 R = 2, 1001
 S9:        DO 120 C = 2, 1001
S10:          G0(R, C) = F(G1, R, C)
S11: 120      CONTINUE
S12: 130    CONTINUE
S13:      EXCLUDE_HERE
S14:      CHECKPOINT_HERE

S15: 200  CONTINUE

S16:      CALL GRID_WRITE(G0)
S17:      END
```

(a) Code



(b) Control Flow Graph

Statement	MAY_REF	MAY_DEF& MUST_DEF	Statement	MAY_REF	MAY_DEF& MUST_DEF	Statement	MAY_REF	MAY_DEF& MUST_DEF
S1	$\{G0, G1\}$	$\{G0, G1\}$	S7	$\{R\}$	$\{R\}$	S13	$\emptyset$	$\emptyset$
S2	$\emptyset$	$\{I\}$	S8	$\emptyset$	$\{R\}$	S14	$\emptyset$	$\emptyset$
S3	$\emptyset$	$\{R\}$	S9	$\emptyset$	$\{C\}$	S15	$\{I\}$	$\{I\}$
S4	$\emptyset$	$\{C\}$	S10	$\{G1\}$	$\{G0(R{:}C)\}$	S16	$\{G0\}$	$\emptyset$
S5	$\{G0\}$	$\{G1(R, C)\}$	S11	$\{C\}$	$\{C\}$	S17	$\emptyset$	$\emptyset$
S6	$\{C\}$	$\{C\}$	S12	$\{R\}$	$\{R\}$			

(c) Values of MAY_REF, MAY_DEF and MUST_DEF

Statement	DE	RO
S14	$\{G1[interior]\}$	$\{G0[border], G1[border]\}$

(d) Values of the exclusion sets

Figure 3: Cellular Automaton Example

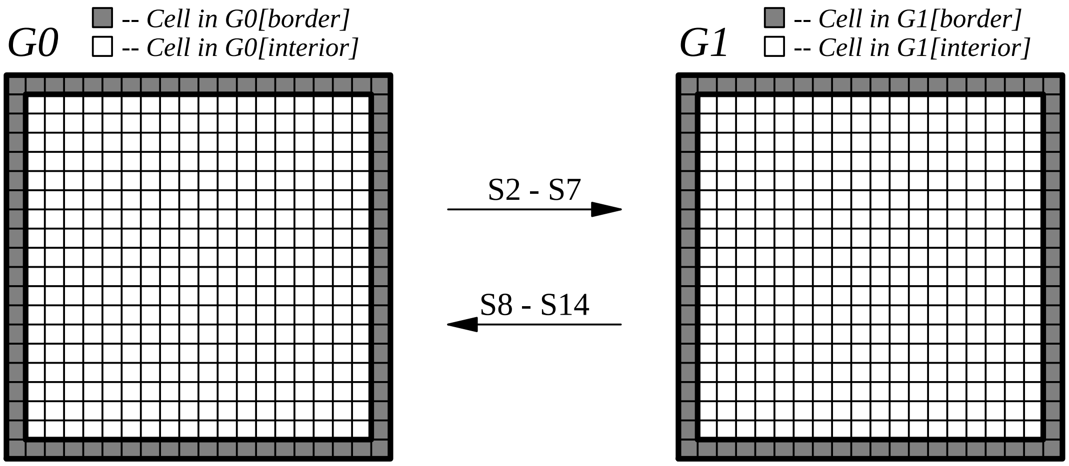
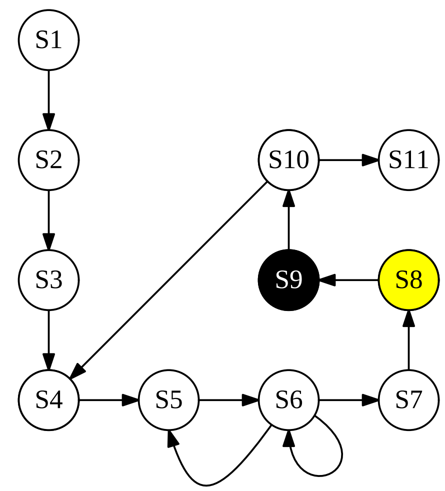


Figure 4: Cellular Automata Code Structure

```
              PROGRAM SIEVE
              INTEGER I, J, K, P(30000)

   S1:        I = 1
   S2:        P(1) = 2
   S3:        DO 300 J = 2, 30000
   S4:  100   I = I + 1
   S5:        DO 200 K = 1, J-1
   S6:  200   IF (DIVIDES(P(K), I)) GOTO 100
   S7:        P(J) = I

   S8:        EXCLUDE_HERE
   S9:        CHECKPOINT_HERE
  S10:  300   CONTINUE
  S11:        END
```

(a) Code



(b) Control Flow Graph

Statement	MAY_REF	MAY_DEF & MUST_DEF	Statement	MAY_REF	MAY_DEF & MUST_DEF	Statement	MAY_REF	MAY_DEF & MUST_DEF
S1	$\emptyset$	{I}	S5	{J}	{K}	S9	$\emptyset$	$\emptyset$
S2	$\emptyset$	{P(1)}	S6	{P(K), I}	$\emptyset$	S10	{J}	{J}
S3	$\emptyset$	{J}	S7	{I, J}	{P(J)}	S11	$\emptyset$	$\emptyset$
S4	{I}	{I}	S8	$\emptyset$	$\emptyset$			

(c) Values of MAY_REF, MAY_DEF and MUST_DEF

Statement	DE	RO
S9	{P(J + 1 : 30000)}	{P(1 : J)}

(d) Values of the exclusion sets

Figure 5: Analysis of Prive Sieve

Note that were we to move the `EXCLUDE_HERE` directive to be between statements S1 and S2, the exclusion sets would be the same, and the `exclude_bytes()` and `include_bytes()` calls would only be called once, instead of sixty times. This is an example of placing the `EXCLUDE_HERE` directive judiciously — although both placements of the directive yield the same checkpoint files, placing it between S1 and S2 will cause fewer procedure calls, and therefore less overhead.

4.2 SIEVE: Prime Sieve

SIEVE is a program that enumerates prime numbers using the classical sieve technique. The code has the structure shown in Figure 5(a), which leads to the control flow graph in Figure 5(b). 30,000 primes are calculated and stored in in array `P`, making use of previously computed primes to test for primality. In the `J`-th iteration of the outer loop, one prime is calculated and stored in location `P(J)`. Primes stored in locations `P(1:J-1)` are referenced in calculating `P(J)`.

Thus, the entire array `P` is dead at the start of the program. At the end of iteration `J`, locations `P(1:J)` are read-only and locations `P(J+1:30000)` are dead.

Memory exclusion is calculated at the bottom of the main loop, just before taking a checkpoint. As shown in Figure 5(d), the read-only and dead portions of array `P` are correctly identified, and the proper memory exclusion procedure calls will be generated.

Unlike the previous **CELL** example, the exclusion calls are not loop invariant. Were we to move the `EXCLUDE_HERE` directive outside the loop (e.g. between statements S2 and S3), no memory exclusion calls could be made other than marking `P(1)` as read-only.

4.3 CONTOUR: Map Contours

CONTOUR is a program that finds altitude contours on a two-dimensional map. The code has the structure shown in Figure 6(a), which leads to the control flow graph in Figure 6(b). The map is represented as a grid of positive altitude values stored in array `MAP`. 100 contours are calculated and the array elements that lie on these contours are set to -1. One contour locus is calculated in each iteration of the outer loop. Each contour is calculated by one pass over the map, where each grid point calculation involves checking the value on that point with respect to its neighboring points, and with respect to the value of the contour locus.

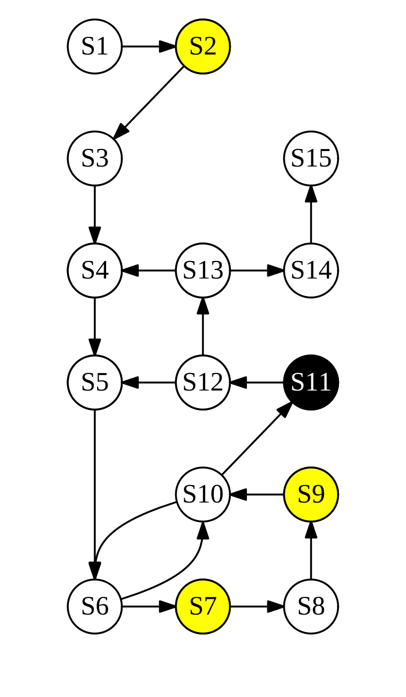
Therefore, the value of a grid point is read-only until it is determined to lie on a contour. It is then set to -1, and it once more becomes read-only.

Memory exclusion is calculated before the main loop, and also just before and after a grid location is modified. The memory exclusion sets for each `EXCLUDE_HERE` call is shown in Figure 6(d). These calls allow the checkpointer to make note of when grid points become read-only, and when they become modified. The resulting checkpoints are minimal, because they only include a grid point if that grid point is written in the previous checkpoint interval.

In this example, the placement of the `EXCLUDE_HERE` directives surrounding the update (statments S7 and S9) is very important. Were the first `EXCLUDE_HERE` to be placed before the conditional

```
              PROGRAM CONTOUR
              INTEGER I, J, MAP(1250,1250)

 S1:          CALL MAP_INIT(MAP)
 S2:          EXCLUDE_HERE

 S3:          DO 300 THRESHOLD = 10, 1000, 10
 S4:            DO 200 I = 1, 1249
 S5:              DO 100 J = 1, 1249
 S6:                IF (.NOT. ONCONTOUR(MAP, I, J))
            X            GOTO 100
 S7:                   EXCLUDE_HERE
 S8:                   MAP(I, J) = -1
 S9:                   EXCLUDE_HERE
S10:  100         CONTINUE
S11:              CHECKPOINT_HERE
S12:  200       CONTINUE
S13:  300   CONTINUE

S14:        CALL MAP_WRITE(MAP)
S15:        END
```

(a) Code



(b) Control Flow Graph

Statement	MAY_REF	MAY_DEF & MUST_DEF	Statement	MAY_REF	MAY_DEF & MUST_DEF
S1	{MAP}	{MAP}	S9	$\emptyset$	$\emptyset$
S2	$\emptyset$	$\emptyset$	S10	{J}	{J}
S3	$\emptyset$	{THRESHOLD}	S11	$\emptyset$	$\emptyset$
S4	$\emptyset$	{I}	S12	{I}	{I}
S5	$\emptyset$	{J}	S13	{THRESHOLD}	{THRESHOLD}
S6	{MAP, I, J}	$\emptyset$	S14	{MAP}	$\emptyset$
S7	$\emptyset$	$\emptyset$	S15	$\emptyset$	$\emptyset$
S8	{I, J}	{MAP(I, J)}			

(c) Values of MAY_REF, MAY_DEF and MUST_DEF

Statement	DE	RO
S3	$\emptyset$	{MAP}
S8	$\emptyset$	$L -$ MAP(I, J)
S10	$\emptyset$	{MAP}

(d) Values of the exclusion sets

Figure 6: Analysis of Map Contour Code

(statement S6) then the update region would include paths which do not include the update and would avoid the second directive altogether. This would result in all locations being included in the checkpoint after one pass through the main loop. The effect of the two directives is to isolate a region of the program with a deterministic and easily analyzed effect. While this is not always an obvious task, it is more straightforward than performing exclusion analysis by hand, and it is always safe.

5 Performance Results

For each of the three examples, we inserted the memory exclusion calls calculated above into the source code, and recompiled the code with **libckpt** [PBKL95]. We then tested the performance of checkpointing these programs with and without the procedure calls for memory exclusion. We also performed tests using standard page-based incremental checkpointing as implemented in **libckpt**.

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. The results presented below are averages of three or more runs of each program.

5.1 CELL : Cellular Automata

The cellular automaton code takes 857 seconds to run in the absence of checkpointing, and has a writable address space of approximately 8.1 megabytes. Checkpoints are taken approximately every 2.5 minutes, meaning that there are six checkpoints taken during the lifetime of the program. The performance of sequential and incremental checkpointing with and without the memory exclusion calls are presented in Figure 7.

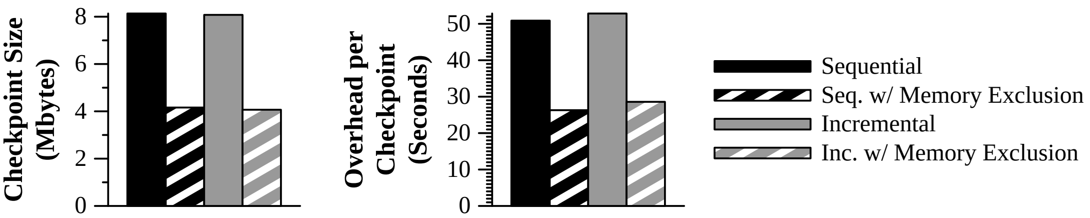


Figure 7: Checkpoint Size and Overhead of **CELL**

This program is interesting because standard incremental checkpointing actually degrades the performance of checkpointing. This is because the interior of each grid is completely updated at each iteration. Therefore the checkpoint sizes for incremental and non-incremental checkpointing are roughly the same, resulting in higher overhead for incremental checkpointing due to the extra cost of processing page faults. With memory exclusion, the major savings in checkpoint size and overhead come from the exclusion of dead memory at the `EXCLUDE_HERE` directive. This is a significant result because it is the first known performance gain through dead memory exclusion that is a result of a safe, automatic mechanism.

5.2 SIEVE: Prime Sieve

The prime sieve code takes 1,445 seconds to run in the absence of checkpointing, and consumes 210 kilobytes of writable memory. As such, it is an excellent program to checkpoint, because the maximum sequential checkpointing time is very small. We checkpoint this program once every two minutes, which results in a total of twelve checkpoints per run. Figure 8 displays the performance of checkpointing the **SIEVE** program.

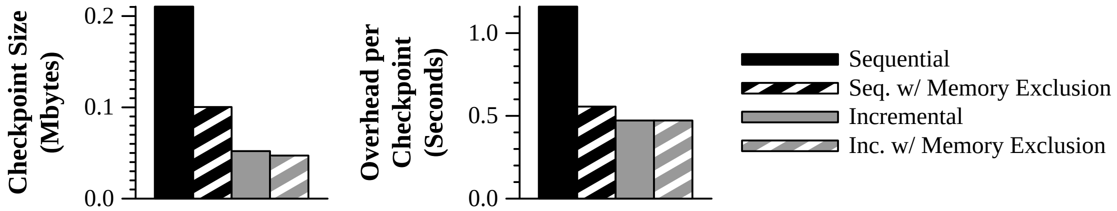


Figure 8: Checkpoint Size and Overhead of **SIEVE**

What the graphs show is that excluding dead and read-only portions of `P` saves roughly about 110 kilobytes per checkpoint. Incremental checkpointing saves another 53 kilobytes — this comes from excluding read-only pages from system data structures (like the standard I/O library).

5.3 CONTOUR: Map Contours

The map contour code was executed on an example map with a 1250×1250 grid of altitudes. It takes 1,072 seconds to run without checkpointing, and has a writable address space of 6.3 megabytes. We checkpoint this program once every three minutes, which results in five checkpoints per run. Figure 9 displays the performance of checkpointing this program.

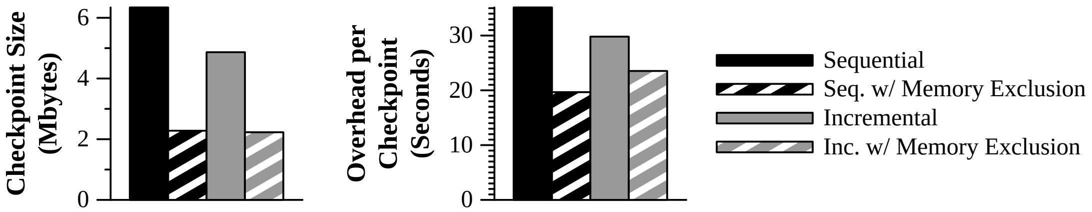


Figure 9: Checkpoint Size and Overhead of **CONTOUR**

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 — even if just one grid point is updated in a page, that page still must be included in an incremental checkpoint. With memory exclusion, read-only data is traced 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 overhead to checkpointing. However, this is more than offset by the savings of writing a smaller checkpoint file to disk.

When memory exclusion is combined with incremental checkpointing, the checkpoint files are smaller, but only marginally. Therefore the overhead of checkpointing is greater than without incremental checkpointing because of the overhead of processing page faults.

6 Conclusions

In this paper, we have presented a compiler-assisted technique for the static analysis of safe memory exclusion in checkpointing. We have expressed exclusion analysis as the solution to a set of data flow equations, allowing us to use 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. In two of these (**CELL** and **CONTOUR**) the static analyses outperform standard incremental checkpointing. In the third, the performance of our technique is comparable to that of incremental checkpointing. In all three cases, combining dynamic incremental checkpointing with static exclusion analysis achieve comparable or better performance than either method alone.

Static exclusion analysis does not replace dynamic incremental checkpointing, but complements it. In some systems page protection mechanisms are not accessible to the user; in such cases static analysis is the only recourse. In other cases (such as the **CONTOUR** program), the page granularity of the dynamic mechanism defeats its effectiveness. Finally, since dead memory locations cannot be detected efficiently at runtime, the exclusion of dead memory can only be implemented through static analysis. The techniques outlined in this paper provide a safe mechanism for identifying dead memory to exclude from checkpoints.

Future Work

We are approaching completion of a FORTRAN preprocessor based on the SUIF toolkit which inserts memory exclusion calls into programs with `EXCLUDE_HERE` and `CHECKPOINT_HERE` directives. We plan to release this tool as a supplementary program to **libckpt**.

In general, memory exclusion eliminates from a checkpoint locations whose values cannot affect the result of the computation. Further exclusions are possible if we consider the checkpoint file to be not simply an image of memory but a sequence of instructions for rebuilding the contents of memory.

In many cases, the instructions for rebuilding a data structure are considerably more compact than its binary image. One example is a binary search tree, that can be stored in a checkpoint as a linearization of its external node values. As long as the tree (or an equivalent tree) can be reconstructed from this linearization, the internal nodes of the tree can be excluded from the checkpoint, representing perhaps a significant savings in checkpointing cost.

This approach can be thought of as a form of compression that uses information specific to a given data structure, and that must be supplied by the implementer of that data structure. In effect, the checkpoint file is then viewed as a sequence of messages from the active process to the recovering process.

This approach lends itself to an object-oriented implementation (perhaps like [BSS94]). Every object has a checkpoint method for saving and restoring its state. The implementer of the object should have the option of implementing efficient save and restore operations, or simply copying the object's binary image to the checkpoint. As with any scheme that substitutes more complex operations for simple data movement, the challenge is to minimize processor overhead, take advantage of the structure of data, and maintain correctness.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BJLM92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9. ACM, October 1992.

[BSS94] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, May 1994.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *SIGPLAN Notices*, 23(7):47–56, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 21–23, 1991.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice & Experience*, 21(6):581–601, June 1991.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. *SIGPLAN Notices*, 28(6):68–77, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[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.

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers Special Issue on Fault-Tolerant Computing*, 41(5), May 1992.

[EZ94] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, Austin, TX, June 1994.

[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.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, New Mexico, June 23–25, 1993. Published as ACM SIGPLAN Notices 28(6).

[LB87] L. Lehmann and J. Brehm. Rollback recovery in multiprocessor ring configurations. In *Proceedings of the 3rd International GI/IGT/GMA Conference on Fault Tolerant Computing Systems*, pages 213–223. Springer-Verlag, 1987.

[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, March 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.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, January 1993.

[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.

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

[PL94a] J. S. Plank and K. Li. Faster checkpointing with $N+1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[PL94b] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[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.

[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: Solutions to the Data Flow Equations

Note that set subtraction ($-$) is used in these tables. For example, the boundary of grid $G0$ of the **CELL** program is denoted as $\{G0 - G0(2{:}1001, 2{:}1001)\}$.

Statement	DEAD	RO	DE
S1	$\{\mathsf{I,J,K,P(1:30000)}\}$	$\emptyset$	$\{\mathsf{P(2:30000)}\}$
S2	$\{\mathsf{J,P(1:30000)}\}$	$\emptyset$	$\{\mathsf{P(2:30000)}\}$
S3	$\{\mathsf{J,P(2:30000)}\}$	$\{\mathsf{P(1)}\}$	$\{\mathsf{P(2:30000)}\}$
S4	$\{\mathsf{P(J:30000)}\}$	$\{\mathsf{J,P-P(J:J)}\}$	$\{\mathsf{P(J:30000)}\}$
S5	$\{\mathsf{P(J:30000)}\}$	$\{\mathsf{J,P-P(J:J)}\}$	$\{\mathsf{P(J:30000)}\}$
S6	$\{\mathsf{P(J:30000)}\}$	$\{\mathsf{J,P-P(J:J)}\}$	$\{\mathsf{P(J:30000)}\}$
S7	$\{\mathsf{P(J:30000)}\}$	$\{\mathsf{J,P-P(J:J)}\}$	$\{\mathsf{P(J:30000)}\}$
S8	$\{\mathsf{P(J+1:30000)}\}$	L	L
S9	$\{\mathsf{P(J+1:30000)}\}$	$\{\mathsf{P(1:J)}\}$	$\{\mathsf{P(J+1:30000)}\}$
S10	$\{\mathsf{P(J+1:30000)}\}$	$\{\mathsf{P(1:J)}\}$	$\{\mathsf{P(J+1:30000)}\}$
S11	L	L	L

Solutions to the data flow equations for **SIEVE**

Statement	DEAD	RO	DE
S1	L	$\emptyset$	$G1$
S2	$\{I, R, C, G1(2{:}1001, 2{:}1001)\}$	$\{G0 - G0(2{:}1001, 2{:}1001), G1 - G1(2{:}1001, 2{:}1001)\}$	$\{I, R, C, G1(2{:}1001, 2{:}1001)\}$
S3	$\{R, C, G1(2{:}1001, 2{:}1001)\}$	$\{I, G0 - G0(2{:}1001, 2{:}1001), G1 - G1(2{:}1001, 2{:}1001)\}$	$\{R, C, G1(2{:}1001, 2{:}1001)\}$
S4	$\{C, G1(R, 2{:}1001)\}$	$\{I, G0 - G0(2{:}1001, 2{:}1001), G1 - G1(R{:}1001, 2{:}1001)\}$	$\{C, G1(R{:}1001, 2{:}1001)\}$
S5	$\{G1(R, C)\}$	$\{G0 - G0(2{:}1001, 2{:}1001), G1 - G1(R{:}1001, C{:}1001)\}$	$\{G1(R{:}1001, C{:}1001)\}$
S6	$\emptyset$	$\{I, G0 - G0(2{:}1001, 2{:}1001), G1 - G1(R{+}1{:}1001, C{+}1{:}1001)\}$	$\{G1(R{+}1{:}1001, 2{:}1001)\}$
S7	$\{C\}$	$\{I, G1 - G1(2{:}1001, 2{:}1001), G1 - G1(R{+}1{:}1001, 2{:}1001)\}$	$\{C\}$
S8	$\{R, C, G0(2{:}1001, 2{:}1001)\}$	$\{I, G0 - G0(2{:}1001, 2{:}1001), R, C, G1 - G1(2{:}1001, 2{:}1001)\}$	$\{R, C, G0(2{:}1001, 2{:}1001)\}$
S9	$\{C, G0(2{:}1001, C)\}$	$\{I, G0 - G0(2{:}1001, C), G1\}$	$\{C, G0(R{:}1001, 2{:}1001)\}$
S10	$\{G0(R, C)\}$	$\{I, G0 - G0(R{:}1001, C{:}1001), G1\}$	$\{G0(R{+}1{:}1001, C{+}1{:}1001)\}$
S11	$\emptyset$	$\{I, G0 - G0(R{+}1{:}1001, C{+}1{:}1001), G1\}$	$\{G0(R{+}1{:}1001, C{+}1{:}1001)\}$
S12	$\{C, G1\}$	$\{I, G0 - G0(R_1{:}1001, 2{:}1001), G1\}$	$\{C, G0(R{+}1{:}1001, 2{:}1001)\}$
S13	$\{I, R, C, G1(2{:}1001, 2{:}1001)\}$	L	L
S14	$\{R, C, G1(2{:}1001, 2{:}1001)\}$	$\{G0 - G0(2{:}1001, 2{:}1001), G1 - G1(2{:}1001, 2{:}1001)\}$	$\{G1(2{:}1001, 2{:}1001)\}$
S15	$\{R, C, G1(2{:}1001, 2{:}1001)\}$	$\{G0 - G0(2{:}1001, 2{:}1001), G1 - G1(2{:}1001, 2{:}1001)\}$	$\{G1(2{:}1001, 2{:}1001)\}$
S16	$L - \{G0\}$	L	$\emptyset$
S17	L	L	L

Solutions to the data flow equations for **CELL**

Statement	DEAD	RO	DE
S1	L	$\emptyset$	L
S2	$\emptyset$	{MAP}	$\emptyset$
S3	{I, J, THRESHOLD}	{MAP}	$\emptyset$
S4	{I, J}	{MAP}	$\emptyset$
S5	{J}	{MAP}	$\emptyset$
S6	$\emptyset$	{MAP}	$\emptyset$
S7	$\emptyset$	L	L
S8	$\emptyset$	L − MAP(I,J)	$\emptyset$
S9	$\emptyset$	L	L
S10	$\emptyset$	{MAP}	$\emptyset$
S11	{J}	{MAP}	$\emptyset$
S12	{J}	{MAP}	$\{J\}$
S13	{I, J}	{MAP}	{I, J}
S14	{I, J, THRESHOLD}	L	{I, J, THRESHOLD}
S15	L	L	L

Solutions to the data flow equations for **CONTOUR**