Static Analysis for Checkpoint Size Reduction in Array-Based Programs

Greg Bronevetsky\textsuperscript{1} and Radu Rugina\textsuperscript{2}

\textsuperscript{1} Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
\texttt{greg@bronevetsky.com},
\textsuperscript{2} Department of Computer Science, Cornell University, Ithaca, NY 14850, USA,
\texttt{rugina@cs.cornell.edu}

Abstract. As modern supercomputing systems reach the peta-flop performance range, they grow in both size and complexity. This makes them increasingly vulnerable to failures from a variety of causes. Checkpointing is a popular technique for tolerating such failures, enabling applications to periodically save their state and restart computation after a failure. Although a variety of automated system-level checkpointing solutions are currently available to High Performance Computing users, manual application-level checkpointing remains more popular due to its superior performance. This paper presents a compiler analysis that improves the performance of automated checkpointing by eliminating dead state, which will be overwritten before it is read by the application. In particular, the analysis focuses on 1-dimensional arrays that are accessed in loops with bounds that may be finite, linear or computed during the loop itself using linear expressions. As a concrete example we show how the analysis enables a 45\% reduction in checkpoint size of \texttt{mdcask}, one of the ASCI Purple benchmarks [1].

1 Introduction
Dramatic growth in supercomputing system capability from tera-flops to peta-flops has resulted in dramatically increased system complexity. Efforts to limit the complexity of Operating Systems for these machines have failed to reduce growing component complexity. Systems like BlueGene/L [2] and the upcoming RoadRunner have grown to more than 100k processors and tens of TBs of RAM; future designs promise to exceed these limits by large margins. Large supercomputers are made from high-quality components, but increasing component counts make them vulnerable to faults, including hardware breakdowns [12] and soft errors [7].

Checkpointing is a common technique for tolerating failures. Application state is periodically saved to reliable storage, and on failure, applications roll back to their prior states. However, automated checkpointing can be very expensive due to the size of saved data and amount of time the application loses
while blocked. For example, dumping all of RAM on a 128K-processor Blue-Gene/L supercomputer to a parallel file system takes approximately 20 minutes [10]. Incremental checkpointing [11] reduces this cost. A runtime monitor tracks application writes, and if it detects that a given memory region has not been modified between two adjacent checkpoints, that region is omitted from the subsequent checkpoint, thereby reducing the amount of data to be saved. Previously explored monitors include virtual memory fault handlers [6], page table dirty bits, and cryptographic encoding techniques [3].

One limitation is that it treats all state at the time of the checkpoint as useful for future application execution. Thus, all runtime techniques save all values generated by the application’s execution. The only optimization is in not saving a given value more than once. However, not all values are useful at all points in the application. In particular, many applications use large temporary buffers that are frequently overwritten with new values. Depending on when a checkpoint is taken relative to such overwrites, it may be possible to avoid saving such buffers because they contain dead data (dead = never read again). This paper presents a compile-time analysis that identifies points in the code when arrays contain dead data. Given an application that has been manually annotated with calls to a checkpoint function, the analysis tracks reads and writes to arrays relative to these locations. If an array will be overwritten between a given checkpoint and any subsequent read of an array, it is eliminated from the checkpoint.

Prior work has looked at compiler analyses for checkpoint optimization [8] [15] [4]. Plank et al [8] assume that the programmer manually annotates the application with directives about which memory regions should be excluded from subsequent checkpoints. It then propagates this dead-ness information and read-only information to determine the data to be saved at each checkpoint location identified by the programmer. The Zhang and Pande [15] analysis focuses on migration for embedded applications. In this context, it identifies code files that will not be used after migration and performs a variable-granularity dataflow analysis to identify dead variables. Our prior work [4] has focused on hybrid compiler/runtime approaches to variable-granularity incremental checkpointing, including extensions to asynchronous checkpointing and OpenMP applications.

Our analysis extends prior work by making it possible to optimize checkpoint sizes by removing arrays that are not themselves dead but contain dead data. This is in contrast to prior work, which treats arrays as plain variables and thus cannot represent read/write patterns to internal array indexes. The analysis handles 1-dimensional arrays that are accessed using loops with a variety of bounds, including finite, linear or computed during the loop itself using linear expressions.

2 Simple Loop Bounds

Consider the example in Figure 1. In this code a checkpoint is taken, followed by a loop that writes to \(A[0..n-1]\), followed by another loop that reads from \(A[0..n-1]\). Since all the values in \(A\) read by code after the checkpoint are overwritten before they are read, (i.e. \([0..n-1] \subseteq [0..n-1]\)) it is not necessary to save \(A\) inside the checkpoint. Indeed this simple inference can be applied to loops
of this form with arbitrarily complex bounds, as long as it can be shown that the bounds do not change between the write loop and the read loop.

```c
checkpoint();
// Write Loop
for(i=0; i<n; i++)
    A[i] = 0;
// Read Loop
for(j=0; j<n; j++)
    q = A[j];
```

Fig. 1. Simple array access pattern

```c
checkpoint();
// Write Loop
m=0;
while(???) {
    A[m] = 0;
    m++;
}
// Read Loop
for(j=0; j<m; j++)
    q = A[j];
```

Fig. 2. Simplified mdcask array access pattern

```c
checkpoint();
// Write Loop
m=0;
while(???) {
    A[m] = 0;
    A[m+1] = 0;
    A[m+2] = 0;
    m+=3;
}
// Read Loop
n=m/3;
l=0;
for(j=0; j<n; j++, l+=3) {
    q = A[l];
    q = A[l+1];
    q = A[l+2];
}
```

Fig. 3. Real mdcask array access pattern

While this pattern suffices for some applications, in this paper we focus on more general patterns, such as those found in mdcask, an ASCI Purple benchmark [1]. A simplified version of this pattern is shown in Figure 2 (the full-complexity version is shown in Figure 3). This pattern is more complex than Figure 1 because the write loop has no explicit bound that is computed before the loop. Instead, the bound is computed during the loop itself, which requires a more complex analysis to identify the final value of \( k \) as the correct loop bound and the overwritten region as \( \text{array}[0..k-1] \). The full complexity pattern in Figure 3 writes and reads array elements in blocks of \( c \) element (3 in this case, although different functions use different constants).

In mdcask this pattern appears in two places. First, it corresponds to message sends, where where the write loop collects data to be sent (data on atoms that migrate from one process to another) and the read loop is a call to MPI\_Send. Second, it corresponds to message receives, where the write loop is a call to MPI\_Recv, which is followed by a read loop that reads the atom data for all the atoms that may be in the message (this number is not known until MPI\_Recv returns). The two patterns in mdcask are common in other applications, and in particular may appear whenever application data is serialized before a data transfer (MPI, TCP/IP or disk storage) and de-serialized after a transfer. In all such cases, an analysis that can infer that the read loop only reads overwritten data can prevent any buffers used used in such communication from being checkpointed. In mdcask such buffers make up 45% of application state.

### 3 Outline of Analysis

The analysis presented in this paper is a two-phase forward analysis. In the first phase, for each checkpoint location and control flow graph (CFG) node it
determines the sub-range of each array that may be live at that point in the code, as seen at the checkpoint location, as well as constraints on the values of variables relative to other variables. In the second phase, this information is used to determine for each checkpoint location and each array read in the application, whether the read array index is live at that checkpoint location. If for a given checkpoint location there exists a read of a live array location, that array is saved as part of the checkpoint. Otherwise, the array is omitted from the checkpoint.

The full analysis is presented in two phases. Section 4 presents a simplified version of the analysis that can successfully show that $A$ can be eliminated from the checkpoint in the example in Figure 2. It presents constraint graphs, the representation used by our analysis to represent array ranges and inter-variable relationships in Section 4.1 and specifies the analysis itself in Section 4.2. Section 5 extends the basic analysis to handle the additional complexity of the example in Figure 3. Section 5.1 presents an auxiliary divisibility analysis that is required for the more complex problem. Section 5.2 extends the basic constraint graphs to enhance their information content. Finally, Section 5.3 shows how the simplified analysis of Section 5.3 is extended to handle Figure 3’s full complexity.

4 Simplified May-Live Range Analysis

4.1 Constraint Graphs

This analysis uses constraint graphs, which are suggested in [5] (4, Chapter 25.5, pp.539-543) and used in Shaham et al [13] in a similar fashion to the analysis presented in this paper. A constraint graph is an undirected graph of inequality relations between pairs of integer scalar variables of the form: $x \leq y + c$, (called “additive inequalities”). A given graph represents the logical conjunction of its constituent inequalities. Each graph edge $cg(x, y)$ is annotated with the corresponding constant $c$. Figure 4 shows an example constraint graph and its corresponding logical expression ($0$ and $\$A$ are explained below).

Individual Inequalities form a lattice, with $c = \infty$ being the top (least information) and $c = -\infty$ being the bottom (most information). The constraint graph itself also forms a lattice that is the product lattice of individual $cg(x, y)$ inequality lattices. In addition, the constraint graph may be $\perp$, which represents an inconsistent set of inequalities. Both lattices are shown in Figure 5. Since the lattices are infinite, we define a widening operator $\triangledown$ on constraint graphs, which speeds up convergence by forcing the inequalities between each $x, y$ pair to step through only a finite number of lattice levels.

Constraint graphs use two special variables. First, 0 is modeled as a variable, which makes it possible to represent relations such as $cg(x, 0) = 5 \equiv x <= 5$. Furthermore, following [13], array ranges are represented as a special class of scalar variables. For each array $A$, we create a separate scalar variable $\$A$, which represents the potentially live region of the array. Thus, if $cg(x, \$A) = 0$ and $cg(\$A, 0) = 7$, this implies that $A$’s may-live region is $[x..7]$. Constraint graphs may not have relations between different array range variables. Furthermore, if
there exists a negative cycle in a constraint graph that passes through an array range variable, the entire graph does not become $\bot$. Instead, the range becomes empty, which is represented as $cg(A) = \diamond$.

Table 1 specifies the operations defined on additive inequalities and constraint graphs. Additive inequalities are ordered according to their information content, with looser constraints (less information) being larger than tighter constraints. As such, the union of two constraints is the constraint that contains no more information and either constraint and the intersection contains all the information of both constraints. The range reduction operation is used to narrow the may-live region of an array when an assignment overwrites an entry on the edge of the range. The normal form operation simplifies the constraint into a standard normalized representation. Although it is redundant here, it is used in the enhanced analysis presented in Section 5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Ordering</td>
<td>higher = less information</td>
</tr>
<tr>
<td>$[x \leq y + c] \preceq [x \leq y + c'] \equiv c \leq c'$</td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>$[x \leq y + c] \sqcup [x \leq y + c'] \equiv \max_{&lt;}([x \leq y + c], [x \leq y + c'])$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$[x \leq y + c] \sqcap [x \leq y + c'] \equiv \min_{&lt;}([x \leq y + c], [x \leq y + c'])$</td>
</tr>
<tr>
<td>Inference</td>
<td>$[x \leq y + c] \land [y \leq z + c'] \equiv [x \leq z + c + c']$</td>
</tr>
<tr>
<td>Normal Form</td>
<td>$[x \leq y + c] \equiv [x \leq y + c]$</td>
</tr>
<tr>
<td>Range Reduction</td>
<td>removes $x + c'$ from the range of $y$ in the given constraint</td>
</tr>
<tr>
<td>$[x \leq y + c] \ominus (y, x + c') \equiv \begin{cases} [x \leq y + c - 1] &amp; \text{if } c = -c' \ [x \leq y + c] &amp; \text{otherwise} \end{cases}$</td>
<td></td>
</tr>
<tr>
<td>$[y \leq x + c] \ominus (y, x + c') \equiv \begin{cases} [y \leq x + c - 1] &amp; \text{if } c = c' \ [y \leq x + c] &amp; \text{otherwise} \end{cases}$</td>
<td></td>
</tr>
<tr>
<td>Variable Update</td>
<td>this constraint to reflect that $i$ has become $i + c'$</td>
</tr>
<tr>
<td>$[x \leq y + c] \oplus (i \to i + c') \equiv \begin{cases} x \leq y + (c - c') &amp; \text{if } x = i \ x \leq y + (c + c') &amp; \text{if } y = i \end{cases}$</td>
<td></td>
</tr>
<tr>
<td>Consistency Check</td>
<td>true of the two constraints are mutually consistent and false otherwise</td>
</tr>
<tr>
<td>$[x \leq y + c] \sim [y \leq x + c'] \equiv c + c' \geq 0$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Additive Inequalities Definitions
Table 2 specifies the operations defined on constraint graphs. The union and intersection operations are trivial extensions of constraint union and intersection. The widening operator ensures that the constraint on any variable pair \((x, y)\) at a given node can change at most twice during a dataflow analysis: once to set the constraint’s initial value and possibly once more to \(\top\) if the pair has two possible constraints along two different code paths and the more recently identified constraint is looser than the original constraint. The transitive closure operation is defined as a the tightest inference that can be made along the path between every pair of scalar variables. Closure is also defined on array range variables and ensures that no constraints are established between different array ranges.

4.2 Simplified May-Live Region Analysis

The simplified may-live region analysis is a forward dataflow analysis that operates on the application’s CFG. For any checkpoint location \(L\), it starts the analysis at the top of the CFG and for each CFG node \(n\) it computes \(mayLive[n] = \langle L, cg \rangle\), where \(cg\) is the constraint graph that contains the relative constraints on all scalar variables and the live regions of all arrays, as seen at \(L\). \(mayLive[n]\) is defined as follows:

\[
mayLive[n] = \begin{cases} 
\langle L, \top \rangle & \text{if } n = \text{Start} \\
mayLive[n] \triangledown \bigcup_{m \in \text{pred}(n)} mayLive[m, n] & \text{Otherwise}
\end{cases}
\]

\(\triangledown\) is the analysis’ transfer function, defined in Table 3.

Figure 6 shows how the analysis operates on the example in Figure 2, providing the inequalities that would be computed by a sample pass of the analysis. The order of the nodes processed by the analysis is chosen for explanatory simplicity and should be read Round I, from top to bottom, followed by Round II from top to bottom and then Round III. Furthermore, we include only the important portions of the constraint graphs at each control flow node.

During the first pass through the writer loop the analysis determines that \(i = 0 \leq n\) and \(0 \leq A\). From this it infers \(i \leq A\). The assignment to \(A\) then causes the bottom element of the \(A\) range to be removed, which produces \(i < A\). When \(i\) is incremented, we return to \(i \leq A\), which is discovered to be a loop invariant when the analysis returns to the top of the writer loop. The second pass through the writer loop produces largely the same constraints as the first pass, except that \(i = 0\) is widened to \(0 \leq i\). The main result of this part of the analysis is that it discovers that \(i \leq A\) is a loop invariant. This means that in every loop iteration \(A[0..i]\) is a dead sub-range of the array.

At the top of the reader loop the analysis’ state contains \(i \leq A\) from the body of the write loop and \(n \leq i\) from write loop’s exit condition. From this we infer that \(n \leq A\), which means that \(A[0..n-1]\) is a dead sub-range of the array. Inside the reader loop the analysis determines that \(j < n\), from which it can infer \(j < A\). As such, the read \(q = A[j]\) only accesses dead values, meaning that array \(A\) can be eliminated from the checkpoint.
\[\bot \equiv \text{Graph contains a negative cycle that only includes scalar variables}\]
\[\top \equiv \forall x, y. \ (x \text{ is scalar or } y \text{ is scalar}) \rightarrow \top(x, y) = \top\]
\[\text{cg}(\$A) = \text{if cg contains a cycle that includes } \$A.\]

Union:
\[(\text{cg} \sqcup \text{cg}')(x, y) = \text{cg}(x, y) \sqcup \text{cg}'(x, y)\]

Intersection:
\[(\text{cg} \sqcap \text{cg}')(x, y) = \text{cg}(x, y) \sqcap \text{cg}'(x, y)\]

Widening:
\[(\text{cg} \triangledown \text{cg}')(x, y) = \begin{cases} \text{cg}(x, y) & \text{if } \text{cg}'(x, y) \not\leq \text{cg}(x, y) \\ \top & \text{otherwise} \end{cases}\]

Transitive Closure:
\[\text{TC}(\text{cg}) \equiv \text{TC}_{\text{range}}(\$A, \ldots, \text{TC}_{\text{range}}(\$A, \text{TC}_{\text{scalars}}(\text{cg})))\]
\[\forall \text{ array variables } A_1, \ldots, A_n.\]

Transitive closure of all scalar variables:
\[\text{TC}_{\text{scalars}}(\text{cg}) \equiv \text{fixed point of } \text{TC}_{\text{base}} \text{ applied to cg.}\]
\[\text{TC}_{\text{base}}(\text{cg})(x, y) = \begin{cases} \min_{\leq}(\text{cg}(x, y), \min_{\leq}(\text{cg}(x, z) \land \text{cg}(z, y))) & \text{if } x, y \text{ and } z \\ \forall z \neq x, y \land \text{are scalars.} \text{cg}(x, y) & \text{otherwise} \end{cases}\]

Transitive closure of an array range variable \$A:
\[(\text{no information passed from one scalar to another via } \$A)\]
\[\text{TC}_{\text{range}}(\$A, \text{cg}) \equiv \text{fixed point of } \text{TC}_{\text{base}}(\$A) \text{ applied to cg.}\]
\[\text{TC}_{\text{base}}(\$A, \text{cg})(x, y) = \begin{cases} \min_{\leq}(\text{cg}(x, y), \min_{\leq}(\text{cg}(x, z) \land \text{cg}(z, y))) & \text{if } x = \$A \text{ or } y = \$A \\ \forall \text{scalars } z \neq x, y & \text{otherwise} \end{cases}\]

Variable Erasure: removes all of \(i\)'s constraints in cg
\[(\text{cg} - i)(x, y) = \begin{cases} \top & \text{if } x = i \text{ or } y = i \\ \text{cg}(x, y) & \text{otherwise} \end{cases}\]

Range Reduction: \(\triangledown\) removes \(x + c\) from the range of array \(A\) in cg
\[(\text{cg} \triangledown (A, i + c))(x, y) = \begin{cases} \text{cg}(x, y) & \text{if } \text{cg}(\$A) = \varnothing \\ \text{cg}(x, y) \triangledown (i, j + c) & \text{if } (x = \$A \text{ and } y = i) \text{ or } (x = i \text{ and } y = \$A) \\ \text{cg}(x, y) & \text{otherwise} \end{cases}\]

Variable Update: modifies \(i\)'s constraints to reflect that \(i = i + c\) in cg
\[(\text{cg} \triangledown (i \rightarrow i + c))(x, y) = \text{cg}(x, y) \triangledown (i \rightarrow i + c)\]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{statement} & \| \text{statement} \| (< L, \text{cg} >) \\
\hline
\(i = i + c\) & update \(i\)'s connections in cg with its modified value \(< L, \text{TC}(\text{cg} \triangledown (i \rightarrow i + c)) >\) \\
\hline
\(i = j + c\) & update cg with the new value of \(i\) \(< L, \text{TC}((\text{cg} - i) \sqcap [i \leq j + c] \sqcap [j \leq i + (-c)]) >\) \\
\hline
\text{use } A[\text{expr}] & \text{< L, cg >} \\
\hline
\text{def } A[i+c] & \text{eliminate } i+c \text{ from the range of } \$A \text{ in cg} \text{< L, TC}(\text{cg} \triangledown (\$A, i + c)) > \\
\hline
\text{def } A[\text{expr}] & \text{< L, cg >} \\
\hline
\(i \leq j + c\) & (true arc) \text{TC}(< L, \text{cg} \sqcap [i \leq j + c]) > \\
& (false arc) \text{TC}(< L, \text{cg} \sqcap [j \leq i + (-1 - c)]) > \\
\hline
\text{checkpoint()} \neq L & \text{< L, cg >} \\
\hline
\end{tabular}
\caption{Simplified May-Live Region Analysis Transfer Function}
\end{table}
The remaining portion of the analysis passes over the reader loop again and determines that \( \leq j \) is a loop invariant. The state at the end of the graph is \( n \leq \$A \land n < i \land n < j \), which means that \( i \) and \( j \) have iterated above \( n \) and the live region of \( A \) is \([n \ldots \infty]\).

5 Full Array Bounds Analysis

Although the simplified analysis is powerful enough to infer that array \( A \) in Figure 2 is dead at the time of the checkpoint, it cannot do the same for the example in Figure 3. The problem is that each iteration of the read loop reads multiple consecutive entries of \( A \). All that the simplified analysis can do is establish that \( j < n \) and \( m < \$A \) hold inside the read loop but it cannot connect \( j \) to \( \$A \) because it cannot represent the multiplicative relationship \( n \times 3 = m \). If it did, and was able to derive \( j + 3 = l \) and \( l \leq n \times 3 \leq m < \$A \), it would only be able to show that \( q = A[l] \) reads dead values. However, it would not identify \( q = A[l + 3] \) as a dead read because \( l + 3 \) may be \( \geq \$A \). The missing information is the fact that both \( l \) and \( m \) are integers divisible by 3. As such, if \( l < m \), then \( l + 3 < m \), which in turn implies that \( l + 3 < \$A \).

To deal with this problem we developed a simple dataflow analysis (presented in Section 5.1) to compute the divisibility information at each node. Section 5.2 details our extensions to the basic constraint graphs that uses affine inequalities instead of additive inequalities to represent the additional divisibility information. Finally, Section 5.3 shows the final extensions to the analysis’ transfer
function that enables it to prove that \( A \) does not need to be checkpointed in the complex example.

### 5.1 Divisibility Analysis

The divisibility analysis is a forward dataflow analysis that computes for each node and variable \( \text{must} \)-information about the variable’s value at the node. This information is represented as triples \(< val, div, rem >\), which form the following lattice:

\[
\begin{align*}
< val : \top, div : \{1\}, rem : \{0\}> : & \text{ more than one value is possible} \\
< val : \top, div : \mathbb{N}^+ - \{1\}, rem : \mathbb{N}^+ > : & \text{ all possible values are } = \text{rem} \mod div \\
< val : \mathbb{Z}, div : \bot, rem : \bot > : & \text{ only one value is possible} \\
< val : \bot, div : \bot, rem : \bot > : & \text{ the variable is unset}
\end{align*}
\]

The analysis proceeds forwards, starting at the CFG’s entry node and records the values that different variables are assigned to as \(< val, \bot, \bot >\). If a variable is assigned a complex expression, its record becomes \(< \top, 1, 0 >\). If two possible values are discovered at a meet point, the analysis computes their greatest common divisor and uses it as the \(\text{div}\) field of its record, leaving \(\text{rem} = 0\). In more complex situations the analysis uses a set of simple rules to pick the best matching divisor and remainder (listed in Table 5). The final result for each node \( n \) is \(\text{divS}[n]\), computed as follows:

\[
\text{divS}[n] = \begin{cases} \\
\forall \text{var. } \text{divS}[n][\text{var}] = < \bot, \bot, \bot > & \text{if } n = \text{Start} \\
\text{divS}[n] := \text{divS}[n] \parallel_{\text{pred}(n)} \parallel \text{st}(n) \parallel_{\text{div}} (\text{divS}[m]) > & \text{otherwise}
\end{cases}
\]

\(\parallel_{\text{div}}\) is the divisibility analysis’ transfer function, defined in Table 4. It uses operations defined in Table 5. Like the live region analysis, it depends on a widening operator to ensure convergence by limiting the total number of different lattice values that may be associated with a given variable at a given node. One difference is that unlike the may-live region analysis, the divisibility analysis uses the widening operator to also perform the function of the meet operator.

### 5.2 Affine Constraint Graphs

Since the divisibility analysis in Section 5.1 produces information that cannot be stored in constraint graphs with additive inequality constraints, this section extends constraint graphs with constraints of the form \( x \cdot a \leq y \cdot b + c \), where \( x \) and \( y \) are integers (called “affine inequalities”). The primary difference between additive and affine inequalities is that the same affine inequality may be represented by multiple triples \(< a, b, c >\). This creates the need for a non-trivial normal form operator, which as described in Section 5.3 becomes a critical part of the analysis’ reasoning power. Another important difference is that unlike
<table>
<thead>
<tr>
<th>statement</th>
<th>$\parallel$ statement $\parallel_{div}$ (vd)</th>
</tr>
</thead>
</table>
| $x = c$   | \[
edv[x] = < val = c, div = \bot, rem = \bot >\] |
| $x = y$   | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \\ < y.val + c, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \\ < \top, y.div, (y.rem + c) \% (y.div) > & \text{if } y.div \in \mathbb{N}^+ - 1 \\ < \top, 1, 0 > & \text{if } y =< \top, 1, 0 > \end{cases}\] |
| $x = y \pm z$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \text{ or } z.val = \bot \\ < y.val + z.val, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \text{ and } z.val \in \mathbb{Z} \\ < \top, z.div, z.rem > & \text{if } y.val \in \mathbb{Z} \text{ and } z.val \in \mathbb{Z} \\ < \top, y.div, y.rem > & \text{if } z.val \in \mathbb{Z} \text{ and } y.div \in \mathbb{N}^+ - 1 \text{ and } y.div \in \mathbb{N}^+ - 1 \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |
| $x = y * c$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \\ < y.val * c, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \\ < \bot, y.div * c, y.rem * c > & \text{if } y.div \in \mathbb{N}^+ \\ < \top, 1, 0 > & \text{if } y =< \top, 1, 0 > \end{cases}\] |
| $x = y * z$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \\ < y.val * z.val, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \text{ and } z.val \in \mathbb{Z} \\ < \top, z.div * y.val, z.rem * y.val > & \text{if } y.val \in \mathbb{Z} \text{ and } z.div \in \mathbb{N}^+ \\ < \top, y.div * z.val, y.rem * z.val > & \text{if } y.div \in \mathbb{N}^+ \text{ and } z.val \in \mathbb{Z} \\ < \top, y.div, (y.rem + z.rem) \% (y.div) > & \text{if } y.div \in \mathbb{N}^+ - 1 \text{ and } z.val \in \mathbb{N}^+ - 1 \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |
| $x = y / c$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \\ < y.val / c, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \\ < \bot, y.div / c, y.rem / c > & \text{if } y.div \in \mathbb{N}^+ \text{ and } c \text{ divides } y.div \text{ and } y.rem \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |
| $x = y / z$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \text{ or } z.val = \bot \\ < y.val / z.val, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \text{ and } z.val \in \mathbb{Z} \\ < y.val / z.div, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \text{ and } z.div \in \mathbb{N}^+ \text{ and } z.rem = 0 \text{ and } z.div \text{ divides } y \\ < \top, y.div / z.val, y.rem / z.val > & \text{if } y.div \in \mathbb{N}^+ \text{ and } z.val \text{ in } \mathbb{Z} \text{ and } z.val \text{ divides } y.div \text{ and } y.rem \\ < \top, y.div / z.div, y.rem / z.div > & \text{if } y.div \in \mathbb{N}^+ \text{ and } z.div \in \mathbb{N}^+ \text{ and } z.rem = 0 \text{ and } z.div \text{ divides } y.div \text{ and } y.rem \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |
| $x = y \odot z$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \text{ or } z.val = \bot \\ < y.val \odot z.val, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \text{ and } z.val \in \mathbb{Z} \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |
| $x = y \odot c$ | \[
edv[x] = \begin{cases} < \bot, \bot, \bot > & \text{if } y.val = \bot \\ < y.val \odot c, \bot, \bot > & \text{if } y.val \in \mathbb{Z} \\ < \top, 1, 0 > & \text{otherwise} \end{cases}\] |

Table 4. Divisibility Analysis Transfer Function
with additive inequalities, the information-content order for affine inequalities is not a total order. For example, if you consider \([x \leq y \times 5]\) and \([x \leq y]\), neither is stronger than the other since there exist \([x, y]\) pairs that obey one but not the other (ex: \(<1, 1>\) and \(<-1, -1>\)). In general, only constraints where \(a/b\) (the line’s slope) is the same can be directly compared. This results in more complicated rules for inequality union and ordering.

Table 6 extends constraint graphs to affine inequalities. Since constraint graphs themselves do not depend on the internal details of their constraints, the graphs are defined exactly the same as in Section 4.1.

### 5.3 Full May-Live Region Analysis

The primary difference between the simplified and full may-live region analysis is the use of divisibility information, which is computed as shown in Section 5.1 and maintained as shown in Section 5.2. We now show how this divisibility information is incorporated into the may-live region analysis. At each CFG node \(n\), for all scalars \(x\) for which \(divS[n]\) contains divisibility or value information we create a special “divisibility” variable \(\hat{x}\) and add the appropriate affine relations between \(x\) and \(\hat{x}\) to \(n\)’s constraint graph. Thus, if \(divS[n][x] = \langle \top, 5, 1 \rangle\) \((x = 1 \mod 5)\), we add the constraints \([\hat{x} \times 5 \leq x - 1]\) and \([x \leq \hat{x} + 5 + 1]\) to the constraint graph. Since both \(x\) and \(\hat{x}\) are integers, these constraints capture this divisibility information in the constraint graph and allow the constraint graph to infer additional constraints based on this information. The enhanced dataflow
Information Ordering: higher = less information

\[ x \leq y \leq z \leq \cdots \] false
\[ ac \leq \top \equiv \bot \leq \equiv ac \equiv true \]

\[ [x \cdot a \leq y \cdot b + c] \equiv [x \cdot a' \leq y \cdot b' + c'] \equiv [x \cdot a' \leq y \cdot b + c + a'] \equiv [x \cdot a' + a \leq y \cdot b' + a + c'] \equiv [x \cdot a' + a < y \cdot b' + a + c'] \equiv [x \cdot a' + a < y \cdot b + a + c'] \equiv [x < a, a < that.b \iff x = Zero and c = c'] \equiv [a < that.a \iff y = Zero and c = c'] \equiv [c + a' < c' \iff b' + a = a' + b] \equiv [false \iff \text{otherwise}] \\

Explanation: \[ x \cdot a \leq y \cdot b + c \leq [x \cdot a' \leq y \cdot b' + c'] \] if the slopes of the lines are the same and the y-intercept the latter is higher (i.e. less information). If \( x = Zero \) or \( y = Zero \) the tighter constraint is one that eliminates more integer values of \( x \) or \( y \).

Union:
\[ \top \cup ac \equiv ac \cup \top \equiv \top \]
\[ \bot \cup ac \equiv ac \cup \bot \equiv ac \]

\[ [x \cdot a \leq y \cdot b + c] \cup [x \cdot a' \leq y \cdot b' + c'] \equiv [x \cdot a \leq y \cdot b + \max(c, c')] \text{ if } b \cdot a' \equiv a' \cdot b \]
\[ \top \text{ otherwise} \]

Intersection:
\[ \top \cap ac \equiv ac \cap \top \equiv ac \]
\[ \bot \cap ac \equiv ac \cap \bot \equiv \bot \]

\[ [x \cdot a \leq y \cdot b + c] \cap [x \cdot a' \leq y \cdot b' + c'] \equiv [x \cdot a \leq y \cdot b + \min(c, c')] \text{ if } b \cdot a' \equiv a' \cdot b \]
\[ \top \text{ otherwise} \]

Inference:
\[ \top \equiv \top \]
\[ \bot \equiv \bot \]
\[ [x \cdot a \leq y \cdot b + c] \equiv \begin{cases} [x \cdot 1 \leq y \cdot (b/gcd(b, c)) + c/gcd(b, c)] & \text{if } x = Zero \\ [x \cdot (a/gcd(a, c)) \leq y + 1 + c/gcd(a, c)] & \text{if } y = Zero \\ [x \cdot (a/gcd(a, b)) \leq y \cdot b/gcd(a, b) + c] & \text{if } c = 0 \\ [x \cdot (a/gcd(a, b, c)) \leq y \cdot (b/gcd(a, b, c)) + c/gcd(a, b, c)] & \text{if } c \neq 0 \end{cases} \]

\( \gcd = \text{greatest common divisor function} \)

Range Reduction: removes \( x + c' \) from the range of \( y \) in the given constraint
\[ [x \cdot a \leq y \cdot b + c] \cap (y, x \cdot b' + c') \equiv \begin{cases} [x \cdot a \leq y \cdot b + c - 1] & c = -c' \text{ and } a = b' \text{ and } b = 1 \\ [x \cdot a \leq y \cdot b + c] & \text{otherwise} \end{cases} \]
\[ [y \cdot a \leq x \cdot b + c \cap (y, x + c') \equiv \begin{cases} [y \cdot a \leq x + c - 1] & c = c' \text{ and } a = 1 \text{ and } b = b' \\ [y \cdot a \leq x \cdot b + c] & \text{otherwise} \end{cases} \]

Variable Update: modify constraint to reflect that \( i \) has become \( i \cdot b' + c' \)
\[ [x \cdot a \leq y \cdot b + c] \oplus (i \mapsto i \cdot b' + c') \equiv \begin{cases} [x \cdot a \cdot b' \leq x' \cdot b' + (c \cdot b' - c' \cdot b)] & \text{if } x = i \\ [x' \cdot a \leq y \cdot b + b' + (c \cdot b' + c' \cdot a)] & \text{if } y = i \end{cases} \]

Consistency Check: true of the two constraints are mutually consistent and false otherwise
\[ [x \cdot a \leq y \cdot b + c] \equiv [y \cdot a' \leq x \cdot b' + c'] \equiv [x \cdot a' \cdot b' \leq x' \cdot b' + c' \cdot b] \equiv [x \cdot a' + a' \cdot b' \leq c' \cdot b + c \cdot a'] \equiv [x \cdot a' + a' \cdot b' \leq c' \cdot b + c \cdot a'] \equiv [x < a, a < that.b \iff x = Zero and c = c'] \equiv [a < that.a \iff y = Zero and c = c'] \equiv [c + a' < c' \iff b' + a = a' + b] \equiv [false \iff \text{otherwise}] \\

Explanation: the above implies that \( x \cdot a \cdot a' \leq x \cdot b' \cdot b + c' \cdot b + c \cdot a' \), which means that \( x \cdot (a \cdot a' - b' \cdot b) \leq c' \cdot b + c \cdot a' \)

As such, the inequalities are inconsistent if \( a \cdot a' - b' \cdot b = 0 \) and \( -c' \cdot b > c \cdot a' \)

| Table 6. Affine Inequalities Definitions | 12 |
The dataflow equations are:

\[
\text{mayLive}[n] = \begin{cases} < L, \top > & \text{if } n = \text{Start} \\ \text{mayLive}[n] \lor \bigcup_{m \in \text{pred}(n)} \text{mayLive}[m, n] & \text{otherwise} \end{cases}
\]

\[
\text{mayLive}[m, n] = TC(\text{addDivInfo}(\text{removeDivVars}(\text{addDivInfo}(\text{divS}[n], \| \text{st}(< m >) \| (\text{mayLive}[m])))\)))
\]

Adds to cg all the divisibility information present at the current node

\[
\text{addDivInfo}(vd, cg) = cg \bigcap_{\text{all scalars } x} \text{varDivInfo}(vd, cg, x)
\]

\[
\text{varDivInfo}(vd, cg, x) = \begin{cases} \{[\hat{x} + 1 \leq x + 0] \cap [x \leq \hat{x} + 1 + 0] \mid [vd[x].val \in \mathbb{Z} \]
\text{if } [vd[x].val \in \mathbb{Z}] \\
[x \leq \hat{x} \cdot vd[x].div + vd[x].rem] \cap \text{if } [vd[x].div \in \mathbb{N}^+] \\
\top & \text{otherwise} \end{cases}
\]

Removes from cg all the scalars i that have divisibility variables

\[
\text{removeDivVars}(cg, vd) = cg - \{i \mid \text{vd}[x].val \in \mathbb{Z} \text{ or } [vd[x].div \in \mathbb{N}^+]\}
\]

The new equations are different in two ways. First, addDivInfo is used to add each node’s divisibility information available to its constraint graph. Second, after the resulting node is transitively closed, removeDivVars removes the variables for which divS[n] has divisibility information. addDivInfo then brings them back, followed by transitive closure. The idea behind this step is that x and \(\hat{x}\) represent information for the same variable. However, given the limitations of our representation and the information loss due to widening, it is possible for either x or \(\hat{x}\) to contain more useful information. For our analysis, it is generally the case that \(\hat{x}\) contains the most useful information, such as \(j = \hat{i}\) being an invariant of the write loop in Figure 3. We have thus chosen to focus on the data of the divisibility variables by dropping the constraints of each regular variable x for which we have \(\hat{x}\), and recomputing x’s constraints from \(\hat{x}\).

The new equations also use a modified transfer function that is identical to that in Table 3, except for the modified \(i = i + c\) case, shown in Table 7.

<table>
<thead>
<tr>
<th>Statement</th>
<th>((&lt; L, cg &gt;))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i = i * b + c)</td>
<td>update i’s connections in cg with its modified value</td>
</tr>
<tr>
<td>(&lt; L, TC(((cg - i) \oplus (i \rightarrow i * b + c)) \cap varDivInfo(divS[n], cg, i))&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Extended May-Live Region Analysis Transfer Function

The operation of the enhanced may-live region analysis is very similar to the simplified analysis. There are two primary differences. First is the inference that \(j \cdot 3 = l\) is an invariant of the read loop. Before the read loop the analysis determines that \(\hat{j} = j = l = \hat{l} = 0\). Inside the loop new values of j and l cause \(j = l\) to be lost due to widening, leaving \(\hat{j} = \hat{l}\) as a loop invariant. However, since \(\hat{j} = j\) and \(\hat{l} \cdot 3 = l\) inside the loop, the addDivInfo(removeDivVars()) step in the dataflow equations causes the relationship between j and l to be recomputed as \(j \cdot 3 = l\), which becomes a loop invariant.
The second difference is in how the enhanced analysis can show that the read loop only accesses dead data. Immediately before the read loop the analysis state includes \( n \cdot 3 = m \leq \$A \) \( \land \) \( m \cdot 3 = m \). Inside the reader loop the analysis also determines that \( 0 \leq j \leq n - 1 \land l = \hat{l} + 3 \). Using the invariant \( j \cdot 3 = l \) discussed above, the analysis infers \( l \leq \$A - 3 \) using the following inference:

\[
l \leq j \cdot 3 \leq n - 1 \land n \cdot 3 \leq m \leq \$A \rightarrow l \leq 3 \cdot n - 3 \leq m - 3 \leq \$A - 3
\]

Thus, the analysis proves that \( q=A[l] \), \( q=A[l+1] \) and \( q=A[l+2] \) are all dead reads, making it possible to eliminate \( A \) from the checkpoint.

6 Evaluation

We are currently implementing the above analysis using the ROSE [9] compiler infrastructure. Preliminary results show that it can analyze mdcask’s array access kernels, although it cannot yet process the application itself due to the limited support for Fortran currently available in ROSE. The analysis can show that arrays msgbuf, temp.msg.buf and ekin can be eliminated from checkpoints, if the checkpoint location is placed anywhere at the top level inside the main mdcask computation loop in file main.f. These arrays collectively make up \(~54\%\) of mdcask’s memory use, excluding any memory allocated by the MPI communication library, which would not be checkpointed in any case.

Arrays msgbuf and temp.msg.buf make up the bulk of the eliminated checkpoint. Their uses relate to MPI_Send and MPI_Recv calls in the application, which are used to send information about atoms that migrate from one process to another. The MPI_Send-related code corresponds to the pattern in Figure 2, where the write loop collects data on the migrating atoms and the read loop is the call to MPI_Send. The MPI_Recv-related code corresponds to the pattern in Figure 3, where the write loop is the call to MPI_Recv and the read loop processes the incoming atom data. Since our analysis needs loop bound divisibility information to handle code like in Figure 3, we assume that the application is annotated with information about which MPI_Send calls match which MPI_Recv calls. This can either be done manually or using a separate tool, such as MPI-Spin [14].

To put the performance of our analysis into perspective, mdcask has a handwritten checkpointer built into the code. This checkpointer eliminates \(~77\%\) of the application state. The additional data structures omitted by the handwritten checkpointer are not amenable to our analysis because they are accessed through an indirection array. As such, future progress on mdcask would require an array sub-range representation that includes the indirection vector itself.

7 Summary

This paper presents a novel compiler analysis that can identify state that is dead at the time of a given checkpoint and can thus be eliminated from the checkpoint. Given the relatively low speed of disk I/O relative to the computing power of modern High Performance Computing systems, this size reduction can lead to a significant speedup of automated checkpointing. This will enable users to avoid the manual effort of implementing checkpointing on their own without giving up on performance.
We have evaluated our analysis with the mdcask code from the ASCI Purple benchmark suite and have found that it reduces mdcask's checkpoint sizes by ∼45%. Although this is not as much as the ∼77% reduction that is achieved by mdcask's hand-written checkpointer, we expect that future work on compiler analyses and hybrid compiler-runtime solutions will close this gap.

References