LLNL-CONF-400662

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Compiler-Enhanced Incremental Checkpointing for OpenMP Applications

Greg Bronevetsky, Daniel Marques, Keshav Pingali, Radu Rugina, Sally A. McKee

January 23, 2008

International Conference on Supercomputing
Kos, Greece
June 7, 2008 through June 12, 2008

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Compiler-Enhanced Incremental Checkpointing for OpenMP Applications

Greg Bronevetsky
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA
greg@bronevetsky.com

Daniel Marques
Keshav Pingali
Institute for Computational Engineering and Sciences,
The University of Texas at Austin,
Austin, TX 78712, USA
daniel@ices.utexas.edu,
pingali@cs.utexas.edu

Radu Rugina
Department of Computer Science
Cornell University,
Ithaca, NY 14853, USA
rugina@cs.cornell.edu

Sally A. McKee
Computer Systems Lab
Cornell University
Ithaca, NY 14853, USA
sam@csl.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 HPC users, manual application-level checkpointing remains more popular due to its superior performance. This paper improves performance of automated checkpointing via a compiler analysis for incremental checkpointing. This analysis, which works with both sequential and OpenMP applications, reduces checkpoint sizes by as much as 80% and enables asynchronous checkpointing.

1. Introduction

Dramatic growth in supercomputing system capability from teraflops 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 [3] 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 [14] and soft errors [9].

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 BlueGene/L supercomputer to a parallel file system takes approximately 20 minutes [12]. Incremental checkpointing [13] 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 [8], page table dirty bits, and cryptographic encoding techniques [4].

When virtual memory fault handlers track application writes, checkpointing can be optimized via "copy-on-write checkpointing" or, more generally, "asynchronous checkpointing". At each checkpoint, all pages to be checkpointed are marked non-writable and placed on a write-out queue. The application continues executing, and a separate thread asynchronously saves pages on the write-out queue. When the checkpointing thread is finished saving a given page, the page is marked writable. If the application tries to write to a page that hasn't yet been saved, the segmentation fault handler is called, a copy of the page is placed in the write-out queue, and the application resumes execution. Checkpointing is thus spread over a longer period of time, reducing pressure on the I/O system, and allowing the application to continue executing.

In contrast to prior work, which uses runtime techniques for monitoring application writes, here we present a compile-time analysis for tracking such writes. Given an application that has been *manually annotated* with calls to a `checkpoint` function, for each array the analysis identifies points in the code such that either:

- there exist no writes to the array between the point in the code and the next checkpoint and/or
- there exist no writes to the array between the last checkpoint and the point in the code

When the analysis detects that a given array is unmodified between two checkpoints, this array is omitted from the second checkpoint. Also, the analysis enables the use of asynchronous checkpointing because each array can be asynchronously saved during the time period from the last pre-checkpoint write to the array until the first post-checkpoint write. In contrast to prior work, this technique enables asynchronous checkpointing to begin before the checkpoint itself. However, because it works at array granularity rather than the page- or word-granularity of runtime monitoring mechanisms, it may be more conservative in its decisions. Furthermore, the analysis makes the assumption that when the application reaches a one potential checkpoint location, it can determine whether it will take a checkpoint when it reaches the next location.

Prior work has looked at compiler analyses for checkpoint optimization [10] [15], and has focused on pure compiler solutions that reduce the amount of data checkpointed. Our work presents a hybrid compiler/runtime approach that uses the compiler to optimize certain portions of an otherwise runtime checkpointing solution. This allows us to both reduce the amount of data being checkpointed, as well as support purely runtime techniques such as asynchronous checkpointing.

2. Compiler/Runtime Interface

Our incremental checkpointing system is divided into run-time and compile-time components. The checkpointing runtime may either checkpoint application memory inside of `checkpoint` calls or include an extra thread that checkpoints asynchronously. Two checkpointing policies are offered. Memory regions that do not contain arrays (a small portion of the code in most scientific applications) are saved in a blocking fashion during calls to `checkpoint`. Arrays are dealt with in an incremental and possibly asynchronous fashion, as directed by the annotations placed by the compiler. The compiler annotates the source code with calls to the following functions:

- `add_array(ptr, size)` Called when an array comes into scope to identify the array's memory region.
- `remove_array(ptr)` Called when an array leaves scope. Memory regions that have been added but not removed are treated incrementally by the checkpointing runtime.
- `start_chkpt(ptr)` Called to indicate that the array that contains the address `ptr` will not be written until the next checkpoint. The runtime may place this array on the write-out queue and begin to asynchronously checkpoint this array.
- `end_chkpt(ptr)` Called to indicate that the array that contains the address `ptr` is about to be written. The `end_chkpt` call must block until the checkpointing thread finishes saving the array. It is guaranteed that there exist no writes to the array between any checkpoint and the call to `end_chkpt`.

Overall, the runtime is allowed to asynchronously checkpoint a given array between calls to `start_chkpt` and `end_chkpt` referring to this array. If `start_chkpt` is not called for a given array between two adjacent checkpoints, this array may be omitted from the subsequent checkpoint because it was not written to between the checkpoints.

For a more intuitive idea of how this API is used, consider the transformation in Figure 1. The original code contains two `checkpoint` calls, with assignments to arrays `A` and `B` in between. The code within the `...`'s does not contain any writes to `A` or `B`. It is transformed to include calls to `start_chkpt` and `end_chkpt` around the writes. Note that while `end_chkpt(B)` is placed immediately before the write to `B`, `start_chkpt(B)` must be placed at the end of `B`'s write loop. This is because a `start_chkpt(B)` call inside the loop may be followed by writes to `B` in subsequent iterations. Placing the call immediately after the loop ensures that this cannot happen.

Original Code	Transformed Code
`checkpoint();`	`checkpoint();`
`...`	`...`
`A[...]=...;`	`end_chkpt(A);`
`...`	`A[...]=...;`
`for(...) {`	`start_chkpt(A);`
`...`	`...`
`B[...]=...;`	`for(...) {`
`...`	`...`
`}`	`end_chkpt(B);`
`...`	`B[...]=...;`
`checkpoint();`	`...`
	`}`
	`start_chkpt(B);`
	`...`
	`checkpoint();`

Figure 1. Transformation example

3. Compiler Analysis

The incremental checkpointing analysis is a dataflow analysis consisting of forward and backward components. The forward component, called the *Dirty Analysis*, identifies the first write to each array after a checkpoint. The backward component, called the *Will-Write* analysis, identifies the last write to each array before a checkpoint. "Array" here refers to any block of memory. The analysis presented here is focused on the simple case of memory buffers that are lexically identified in the source code as arrays. Issues such as aliasing, heap arrays and pointer arithmetic are orthogonal to this analysis and are addressed by a large body of ongoing work.

3.1 Basic Analysis

For each array at each node n in a function's control-flow graph(CFG) the analysis maintains two bits of information:

- $mustDirty[n](array)$: $True$ if there *must* exist a write to $array$ along *every* path from a `checkpoint` call to this point in the code; False otherwise. Corresponds to the dataflow information immediately *before* n.
- $mayWillWrite[n](array)$: $True$ if there *may* exist a write to $array$ along *some* path from a this point in the code to a `checkpoint` call; False otherwise. Corresponds to the dataflow information immediately *after* n.

This information is propagated through the CFG using the dataflow formulas in Figure 2. The Dirty and Will-Write analyses start at the top and bottom of each function's CFG, respectively, in a state where all arrays are considered to be clean (e.g., consistent with the previous and next checkpoint, respectively). They then propagate forward and backward, respectively, through the CFG, setting each array's write bit to $True$ at each write to this array. When each analysis reaches a `checkpoint` call, it resets the state of all the arrays to $False$. For the Dirty Analysis, all dirty arrays will become clean because they are checkpointed. For the Will-Write Analysis, at the point immediately before a checkpoint there exist no writes to any arrays until the next checkpoint, which is the checkpoint in question.

The application source code is annotated with calls to `start_chkpt` and `end_chkpt` using the algorithm in Figure 3. Such calls are added in three situations. First, `end_chkpt(array)` is inserted immediately before node n if n is a write to $array$ and is not preceded by any other write to $array$ along *some* path that starts at a call to `checkpoint` and ends with node n. Second, `start_chkpt(array)` is inserted immediately after node n if n is a write to $array$ and there do not exist any more writes to $array$ along *any* path that starts with n and ends at a `checkpoint` call. Third, a `start_chkpt(array)` is inserted on a CFG branching edge $m \rightarrow n$ if $mayWillWrite[n](array)$ is true at m, but false at n, due to merging of dataflow information at branching point m. This treatment is especially important when an array is written inside a loop. In this case, $mayWillWrite[n](array)$ is true at all points in the loop body, since the array may be written in subsequent loop iterations. The flag becomes false on the edge that branches out of the loop, and the compiler inserts the `start_chkpt(array)` call on this edge.

Because the Dirty analysis is based on *must-write* information, `end_chkpt` calls are conservatively placed as late as possible after a checkpoint. Furthermore, the Will-Write analysis' use of *may-write* information conservatively places `start_save` calls as early as possible before a checkpoint.

To provide an intuition of how the analysis works, consider the example in Figure 4. In particular, consider the points in the code

$$mustDirty[n](array) = \begin{cases} False & \text{if } n = \text{first node} \\ \bigcap_{m \in pred(n)} mustDirtyAfter[m](array) & \text{otherwise} \end{cases}$$

$$mustDirtyAfter[m](array) = [\![m]\!](mustDirty[m](array), array)$$

$$mayWillWrite[n](array) = \begin{cases} False & \text{if } n = \text{last node} \\ \bigcup_{m \in succ(n)} mayWillWriteBefore[m](array) & \text{otherwise} \end{cases}$$

$$mayWillWriteBefore[m](array) = [\![m]\!](mayWillWrite[m](array), array)$$

Statement m	$[\![m]\!](val, array)$
`array[expr] = expr`	$True$
`checkpoint()`	$False$
other	val

Figure 2. Dataflow formulas for Dirty and Will-Write analyses

```
foreach (array $array$), foreach (CFG node $n$) in application
  // if $n$ is the first write to $array$ since the last checkpoint call
  if($mustDirty[n](array) = False \wedge$
     $mustDirtyAfter[n](array) = True$)
       place end_chkpt($array$) immediately before $n$
  // if $n$ is the last write to $array$ until the next checkpoint call
  if($mayWillWriteBefore[n](array) = True \wedge$
     $mayWillWrite[n](array) = False$)
       place start_chkpt($array$) immediately after $n$
  // if $n$ follows the last write on a branch where $array$ is
  // no longer written
  if($mayWillWriteBefore[n](array) = False \wedge$
    $\exists m \in pred(n).\ mayWillWrite[m](array) = True$)
       place start_chkpt($array$) on edge $m \rightarrow n$
```

Figure 3. Transformation for inserting calls to `start_chkpt` and `end_chkpt`

where $mustDirty$ and $mayWillWrite$ change from $False$ to $True$. These are the points where `end_chkpt` and `start_chkpt` calls are inserted.

3.2 Loop-Sensitive Analysis

The basic analysis performs correct transformations, but it has performance problems when applied to loops. This can be seen in the transformed code in Figure 4. While `start_chkpt(B)` is placed immediately after the loop that writes `B`, `end_chkpt(B)` is placed inside the loop, immediately before the write to `B` itself. This happens because the placement of `end_chkpt` depends on *must-write* information, instead of the *may-write* information used in placing `start_chkpt`. This placement is conservative, and becomes problematic in the case where the first post-checkpoint write to an array happens in a small, deeply-nested loop, which are very common in scientific computing. In this case `end_chkpt` will be called during each iteration of the loop, causing potentially severe overheads.

To address this problem, the above analysis is augmented with a loop-detection heuristic, shown in Figure 5. This heuristic uses may-Dirty information, in addition to the must-Dirty and may-WillWrite information of Section 3, and identifies the patterns of dataflow facts that must hold at the top of the first loop that writes to an array after a checkpoint. Figure 5 contains the CFG of such a loop and identifies edges in the CFG where the various dataflow facts are $True$. The pattern at node $i < n$ is:

- $mustDirty[i < n](B) = False$
- $mayDirty[i < n](B) = True$
- $mayWillWrite[i < n](B) = True$
- $pred(i < n) > 1$

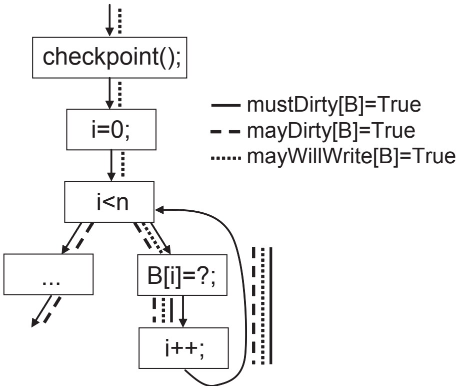

Figure 5. Dataflow pattern for writes inside loops

Furthermore, the CFG edge that points to $i < n$ from outside the loop is the one coming from the predecessor p where $mustDirtyAfter[p](B) = False$. Thus, by placing `end_chkpt(B)` on this incoming edge, we can ensure both that `end_chkpt(B)` is called before any write to `B`, and that it is not executed in every iteration of the loop.

Since this heuristic only applies to loops, it does not place `end_chkpt(A)` before the write to `A` in Figure 1. As such, we need to use both rules to ensure that `end_chkpt` is placed conservatively. However, if both rules are used then the example in Figure 1 will get two `end_chkpt(B)` calls: one before `B`'s write loop and one before the write itself, negating the purpose of the loop-sensitive placement strategy. To prevent this from happening we propose an extra *EndChkpt-Placed* analysis that prevents `end_chkpt(array)` from being placed at a given node if there already exists an `end_chkpt(array)` on every path from any `checkpoint` call to the node. *EndChkpt-Placed* is a forward analysis executed as a separate pass from the Dirty and Will-Write passes. It maintains a bit of information for every array at every CFG node. $mustEndChkptPlaced[n](array)$ is set to $True$ if `end_chkpt(array)` is to be placed immediately before node n. It is set to $False$ if `start_chkpt(array)` is to be inserted at n. The latter rule ensures that the "exclusion-zone" of a given insertion of `end_chkpt(array)` doesn't last past the next `checkpoint` call.

To implement this rule the loop-sensitive analysis maintains for each CFG node n the following additional dataflow information:

Original Code	Code with Dirty States	Code with Will-Write States	Transformed Code
`checkpoint();`	`checkpoint(); [A→F,B→F]`	`checkpoint(); [A→T,B→T]`	`checkpoint();`
`...`	`... [A→F,B→F]`	`... [A→T,B→T]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`A[...]=...; [A→F,B→T]`	`end_chkpt(A);`
`...`	`... [A→T,B→F]`	`[A→F,B→T]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→F]`	`for(...) { [A→F,B→T]`	`start_chkpt(A);`
`...`	`... [A→T,B→F]`	`... [A→F,B→T]`	`...`
`B[...]=...;`	`B[...]=...; [A→T,B→F]`	`B[...]=...; [A→F,B→T]`	`for(...) {`
`...`	`... [A→T,B→T]`	`... [A→F,B→T]`	`...`
`}`	`} [A→T,B→T]`	`} [A→F,B→T]`	`end_chkpt(B);`
`...`	`... [A→T,B→F]`	`... [A→F,B→F]`	`B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→F]`	`checkpoint(); [A→F,B→F]`	`...`
			`}`
			`start_chkpt(B);`
			`...`
			`checkpoint();`

Figure 4. Analysis example

```
foreach ($array$), foreach (CFG node $n$) in application
    if $placeEndChkptNode(n, array)$
        place end_chkpt($array$) immediately before $n$
    if $\exists m \in pred(n).\ placeEndChkptEdge(m, n, array)$
        place end_chkpt(array) on edge $m \rightarrow n$
```

Figure 7. Loop-sensitive transformation for inserting calls to `end_chkpt`

- $mayDirty[n](array)$: $True$ if there *may* exist a write to $array$ along *some* path from a `checkpoint` call to this point in the code; $False$ otherwise. Corresponds to the dataflow information immediately *before* n.
- $mustEndChkptPlaced[n](array)$: $True$ if *all* paths from any `checkpoint` call to this point in the code contain a point where a `end_chkpt(array)` call will be placed.

This information is computed as shown in Figure 6. The modified rules for placing `end_chkpt` calls are shown in Figure 7 and Figure 8 extends the example in Figure 1 with the new $mustEndChkptPlaced$ information and the new placement of `end_chkpt` calls.

3.3 Inter-Procedural Analysis

We extend the above analysis with a context-insensitive, flow-sensitive inter-procedural analysis. The inter-procedural analysis applies the data-flow analysis from Section 3.2 to the CFG that contains all of the application's functions. When the analysis reaches a function call node for the first time, it computes a summary for that function by applying the dataflow analysis using the formulas in Figure 6, but with a modified lattice.

In addition to the standard $True$ and $False$, we introduce an additional $Unset$ state that appears below $True$ and $False$ in the lattice. All the dataflow facts for all arrays are initialized to $Unset$ at the start or end of the function (start for the forward analyses and end for the backward analysis). The standard analysis is then executed on the function using the extended lattice, with $Unset$ being treated as $False$ for the purposes of the EndChkpt-Placed analysis. If the state of a given array remains $Unset$ at the end of a given pass, this means that it was not modified by the pass. In the case of the Dirty and Will-Write analyses this means that the array is not written to inside the function. In the case of the EndChkpt-Placed analysis, this means that no `end_chkpt` calls are placed for this array inside the function. The function summary then consists of the dataflow facts for each array at the opposite end of the function: end for the forward analyses and start for the backward analysis. Function calls are processed by applying the function summary as a mask on all dataflow state. If $dataFlow[array] = Unset$ in the function summary, $array$'s mapping is not changed in the caller. However, if $dataFlow[array] = True$ or $False$ in the summary, the corresponding dataflow fact for $array$ is changed to $True$ or $False$ in the caller.

4. OpenMP Support

The increasing popularity of shared memory platforms for HPC (ranging from clusters of symmetric multi-processors to large shared-memory machines like the SGI Altix) has led to the increased importance of the shared memory programming model. OpenMP is one of the most popular shared-memory APIs, with many applications written in either pure OpenMP or a combination of OpenMP and MPI. We have extended the above analysis to support multi-threaded OpenMP applications. OpenMP offers a structured fork-join parallelism model, with parallel regions of code identified using `#pragma omp parallel`. It also offers support for variable privatization, work-sharing, and synchronization. In light of prior work on shared memory checkpointing, our work has focused on *blocking* checkpointing tools such as C^3[5] [6] and BLCR[7], for these have proved to be the most portable. In particular, the analysis assumes that `checkpoint` calls are (i) global barriers across all threads, and (ii) every thread will execute the *same* `checkpoint` call as part of the same checkpoint. This is similar to OpenMP's semantics for placing `#pragma omp barrier`.

The intuition behind our analysis extension is that each thread is treated as a sequential application. The sequential analysis is applied to the application, ignoring any interactions among threads. This ensures that `start_chkpt` and `end_chkpt` calls are placed such that for any thread:

- There are no writes to `array` by the thread between a `start_chkpt(array)` call and a `checkpoint` call.
- There are no writes to `array` by the thread between a `checkpoint` call and a `end_chkpt(array)` call.

This alone is sufficient for code outside of `#pragma omp parallel` constructs and code that deals with private variables. This is because in both cases the array access patterns are sequential. However, it presents problems for parallel code that deals with shared variables. Consider the case of `start_chkpt(array)`, where `array` is shared. Although each thread is guaranteed to call `start_chkpt(array)` after the last pre-checkpoint write to `array`, the fact that one thread has called `start_chkpt(array)` does not mean that all threads are finished writing to array. As such, in the multi-threaded setting the checkpointing runtime is not allowed to begin asynchronously checkpointing `array` until *all*

$$mayDirty[n](array) = \begin{cases} False & \text{if } n = \text{first node} \\ \bigcup_{m \in pred(n)} mayDirtyAfter[m](array) & \text{otherwise} \end{cases}$$

$$mayDirtyAfter[m](array) = [\![m]\!](mayDirty[m](array), array)$$

$$mustEndChkptPlaced[n](array) = \begin{cases} False & \text{if } n = \text{first node} \\ \bigcap_{m \in pred(n)} mustEndChkptPlacedAfter[m](array) & \text{otherwise} \end{cases}$$

```
$mustEndChkptPlacedAfter[m](array) =$
    if $\neg\, placeStartChkptNode(m, array) \ \wedge\ \neg\, \exists l \in pred(m).\ placeStartChkptEdge(l, m, array)$ then
        $False$
    else if $(placeEndChkptNode(m, array) \ \vee\ \exists l \in pred(m).\ placeEndChkptEdge(l, m, array))$ then
        $True$
    else $mustEndChkptPlaced[m](array)$
```

```
// end_chkpt(array) will be placed immediately before node $n$ if
$placeEndChkptNode(n, array) =$
    // node $n$ is the first write to $array$ since the last checkpoint
    $(mustDirty[n](array) = False \wedge mustDirtyAfter[n](array) = True)$
```

```
// end_chkpt(array) will be placed along the edge $m \rightarrow n$ if
$placeEndChkptEdge(m, n, array) =$
    // node $n$ is itself clean but predecessor $m$ is dirty, $n$ contains or is followed
    // by a write and predecessor $m$ is not itself preceded by $end\_chkpt(array)$
    $(mustDirty[n](array) = False \wedge mayDirty[n](array) = True \wedge$
     $mayWillWrite[n](B) = True \wedge mustDirtyAfter[m](array) = False \wedge$
     $mustEndChkptPlaced[m](array) = False)$
```

```
// start_chkpt(array) will be placed immediately after node $n$ if
$placeStartChkptNode(n, array) =$
    // node $n$ is the last write to $array$ until the next checkpoint
    $(mayWillWriteBefore[n](array) = True \ \wedge\ mayWillWrite[n](array) = False)$
```

```
// start_chkpt(array) will be placed along the edge $m \rightarrow n$ if
$placeStartChkptEdge(m, n, array) =$
    // node $n$ follows the last write to $array$ until the next checkpoint
    $(mayWillWriteBefore[n](array) = False \wedge mayWillWrite[m](array) = True)$
```

Figure 6. Dataflow formulas for the loop-sensitive extension

Original Code	Code with Must-EndChkptPlaced States	Transformed Code
`checkpoint();`	`checkpoint(); [A→F,B→F]`	`checkpoint();`
`...`	`... [A→F,B→F]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`end_chkpt(A);`
`...`	`[A→T,B→F]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→T]`	`start_chkpt(A);`
`...`	`... [A→T,B→T]`	`...`
`B[...]=...;`	`B[...]=...; [A→T,B→T]`	`end_chkpt(B);`
`...`	`... [A→T,B→T]`	`for(...) {`
`}`	`} [A→T,B→T]`	`...`
`...`	`... [A→T,B→T]`	`B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→T]`	`...`
		`}`
		`start_chkpt(B);`
		`...`
		`checkpoint();`

Figure 8. Transformation example with loop-sensitive optimizations

threads have called `start_chkpt(array)`. Similarly, the checkpointing runtime must finish checkpointing `array` when *any one* thread calls `end_chkpt(array)`.

While the `start_chkpt` rule it simple, it does not work unless the runtime knows the number of threads that will call `start_chkpt` between a pair of checkpoints. Unless this is known, it is not possible to determine whether a given thread's `start_chkpt` is the last one. OpenMP applications spawn new threads by executing code that is marked by `#pragma omp parallel`. Variables declared inside this block are private to each thread and variables declared outside this block may be either private or shared among the spawned threads, with heap data always being shared. A spawned thread may spawn additional threads of its own, making any variable it has access to either shared among all the newly spawned threads or private to each of them. We track the number of threads that a given array is shared among by adding calls to the start_thread and end_thread functions at the top and bottom of each #pragma omp parallel region, respectively:

- start_thread(parentThread, privateArrays) - Informs the checkpointing runtime that parentThread has spawned a new thread, giving it private copies of the arrays in the privateArrays list. All other arrays are assumed to be shared between parentThread and the new thread.
- end_thread() - Informs the checkpointing runtime that the given thread is about to terminate.

Note that since making a shared variable private to a thread is equivalent to creating a copy of the original variable, the start_thread and end_thread calls imply add_array and remove_array calls, respectively, for the privatized variables.

While these runtime monitoring mechanisms can identify the number of threads that *may* call start_chkpt(array), it is not clear that all of these threads will actually perform the call, since different threads may execute different code. Appendix A presents a proof (Theorem 2) that if one thread calls start_chkpt(array) between two checkpoints, all threads will do the same. Since it is possible for a thread to spawn new threads after calling start_chkpt(array), the runtime considers such threads as having also called start_chkpt(array), until the next checkpoint call. Similarly, if threads in a thread group call start_chkpt(array) before exiting their #pragma omp parallel region, the checkpointing runtime decrements the count of threads that have already called start_chkpt(array). Theorem 1 guarantees that if one such thread calls start_chkpt(array), they all do.

5. Experimental Evaluation

5.1 Experimental Setup

We have evaluated the effectiveness of the above compiler analysis by implementing it on top of the ROSE [11] source-to-source compiler framework and applying it to the OpenMP versions [1] of the NAS Parallel Benchmarks [2]. We have focused on the codes BT, CG, EP, FT, IS, LU, SP. MG was omitted from our analysis since it uses dynamic multi-dimensional arrays (arrays of pointers to lower-dimensional arrays), which requires additional complex pointer analyses to identify arrays in the code. In contrast, the other codes use contiguous arrays, which require no additional reasoning power. Each code was augmented with a checkpoint call at the top of its main compute loop and one immediately after the loop.

The target applications were executed on all problem classes (S, W A, B and C, where S is the smallest and C the largest), on 4-way 2.4Ghz dual-core Opteron SMPs, with 16GB of RAM per node(Peloton clusters at the Lawrence Livermore National Laboratory). Whenever data for all problem classes cannot be shown due to space constraints, we present data for class A, since it is representative of the others. Each run was performed on a dedicated node, regardless of how many of the node's cores were actually used by the NAS benchmark. All results are averages of 10 runs and each application was set to checkpoint 5 times, with the checkpoints spaced evenly throughout the application's execution. This number was chosen to allow us to sample the different checkpoint sizes that may exist in different parts of the application without forcing the application to take a checkpoint during every single iteration, which would have been unrealistically frequent. Data for BT, EP and FT is not available at input size C because it uses too much static memory to compile.

The transformed codes were evaluated with two checkpointers. First, we used the Lazarus sequential incremental checkpointer, which implements page-protection-based incremental and asynchronous checkpointing. We extended this checkpointer with the API from Section 2 and used it to compare the performance of purely runtime and compiler-assisted incremental and asynchronous checkpointing. Since we did not have a multi-threaded checkpointer available to us, we developed a model checkpointing runtime to evaluate the performance of our compiler technique on multi-threaded applications.

The model runtime implements the API from Section 2 and simulates the behavior of a real checkpointer. It performs the same state tracking and synchronization as a real checkpointer but instead of actually saving application state, it sleeps for an appropriate period of time. One side-effect is that our checkpointer does not simulate the overheads due to saving variables other than arrays. However, since in the NAS benchmarks such variables make up a tiny fraction of overall state, the resulting measurement error is small. Furthermore, because the model checkpointer can sleep for any amount of time, it can simulate checkpointing performance for a wide variety of storage I/O bandwidths.

The checkpointers have two primary modes of operation: PR mode, where incremental checkpointing is done using a "Purely-Runtime mechanism", and CA mode, where "Compiler-Assistance" is used. Lazarus supports both modes, and the model checkpointer only supports the CA mode. Both checkpointers also support all four permutations of the following modes:
Checkpoint contents:

- Full: Lazarus saves the application's entire state and the model checkpointer simulates the time it would take to checkpoint all currently live arrays.
- Incr: Incremental checkpointing, where only the data that has been written to since the last checkpoint is saved. In PR mode the operating system's page protection mechanism is used to track whether a given page has been written to since the last checkpoint. In CA mode the checkpointer saves all arrays for which start_chkpt has been called since the last checkpoint.

Checkpoint timing:

- Block: The call to checkpoint is blocked while the appropriate portion of application state is saved.
- Asynch: Checkpointing is performed asynchronously using a separate checkpointing thread.

5.2 Pure-Runtime vs. Compiler-Assisted Checkpointing

This section compares the performance of compiler-assisted checkpointing with purely-runtime checkpointing, using Lazarus. [1]

5.2.1 Checkpoint Size

We begin by comparing the sizes of the checkpoints generated by PR and CA. In the context of the NAS codes, which have an initialization phase, followed by a main compute loop, the primary effect of the analysis is to eliminate write-once arrays from checkpointing. These are the arrays that are written to during the initialization phase and then only read from during the main compute loop. Since there do not exist any start_chkpt calls for these arrays during the main compute loop, they are only saved during the first checkpoint. In contrast, Lazarus' page-protection-based mechanism can track any changes to application state, at a page granularity.

Figure 9 shows the sizes of checkpoints created by the PR and CA on each NAS input size, as a percentage of each application's original checkpoint size. The amount of savings varies between different applications, ranging from 80% for CG to ¡10% for EP.

While for small inputs there is sometimes a difference between the effectiveness of the two approaches, for larger input sizes the difference becomes very small. This is because the major application arrays identified as as dead by the analysis grow as a fraction of the overall application state as the input size increases. This shows

[1] Data for Lazarus with IS was not available at the time of submission and will be included in the final version of the paper.

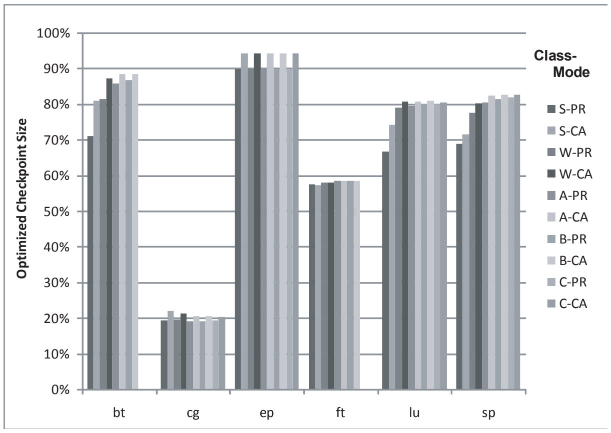

Figure 9. Relative Checkpoint Sizes Generated by Lazarus in PR and CA modes

that it is possible for a compiler-based mechanism to achieve the same reductions in checkpoint size as are available from a runtime-based technique, without any runtime monitoring. Since both the compiler technique presented in this paper and page-based incremental checkpointing operate at a fairly coarse grain of whole arrays and whole pages, in future work we plan to compare finer grain runtime techniques [4] to more accurate compiler-based techniques.

5.2.2 Running Time

The impact of the various checkpointing mechanisms on the application running time was evaluated by looking at (i) the cost of using the given mechanism, without actually writing anything to disk and (ii) the cost of writing to disk using the mechanism. Figure 10 shows the running time of each NAS application on input size A, both without Lazarus (Original column) and with each Lazarus checkpointing configuration. No checkpoint data was written to disk. All times are normalized to Original. In many cases the checkpointing mechanisms themselves add very little cost to original application. The exceptions are FT, CG and BT, where PR shows a notable overhead, with Asynch-PR showing the highest overhead. All three applications have a large amount of state relative to their running time. Since the checkpointing thread must acquire and release a lock that controls access to the checkpointing queue for each page that it needs to checkpoint, this becomes a notable cost because the main application thread is repeatedly forced to stall when it catches up to the slower checkpointing thread. This effect is most important for FT and CG, with 70,000 and 50,000 lock acquisitions per second, respectively and less so for BT, with only 6,000 lock ops/sec. CG shows little overhead with Incr-Asynch-PR because of the significant reductions in its state due to incremental checkpointing. Furthermore, BT shows a notable overhead for Asynch-CA. The reason is that BT's control flows cause the compiler analysis to place end_chkpt calls into frequently used utility functions. This results in many unnecessary calls to end_chkpt which itself releases and acquires a lock, thus raising the overhead for Asynch-CA.

Figure 11 shows the difference between the above times and the running times of these configurations on input size A, where Lazarus writes checkpoints to the parallel file system. This is the time spent writing checkpoints under each scheme. All times are normalized to the cost of Full-Block-PR.

The performance of the different checkpointing techniques is determined by three properties of each application:

- *Pre-Checkpoint Overlap*: the amount of time between start_chkpt calls and subsequent checkpoints. The larger the overlap, the better the performance of CA relative to PR since it allows CA to begin checkpointing much earlier than PR.
- *Post-Checkpoint Overlap*: the amount of time between checkpoints and the subsequent first write to each buffer. Larger overlap leads to better performance with Asynch relative to Block since it gives the checkpointing thread more time to save each memory region before the main thread first writes to it.
- *Post-Checkpoint Locality*: the amount of computation the algorithm performs on each page before moving to the next page during its first access to the page after a checkpoint. Larger amounts raise Asynch-PR performance relative to Asynch-CA since Asynch-PR can checkpoint .

Code	Pre-Ckpt Overlap	Post-Ckpt Overlap	Post-Ckpt Locality
BT	high	low	low
CG	med	low	med
EP	high	med	med
FT	low	low	low
LU	med	low	low
SP	very high	none	none

Table 1. Properties of NAS benchmarks

Table 1 lists these properties for each NAS benchmark. As suggested by the able, BT shows best performance for Block-CA and shows poor performance with all the Asynch configurations. SP is similar to BT, except that it has less state, somewhat larger savings from incremental checkpointing, higher pre-checkpoint overlap and somewhat lower post-checkpoint overlap and locality. Accordingly, SP shows a larger improvement of CA relative to PR than does BT and shows similarly poor performance for all Asynch configurations.

CG is much cheaper with Incr than with Full because of the significant reductions in checkpoint size produced by incremental checkpointing. Both PR and CA show some improvement in running time with Asynch with un-optimized checkpointing but then show a slow-down for small checkpoint sizes. This is because while larger checkpointing costs hide the costs of Asynch, when the cost of checkpointing drops, the low amount of post-checkpoint overlap in CG is not enough to make up for this cost. Finally, the medium post-checkpoint locality in CG results in little difference between Asynch-PR and Asynch-CA.

EP is unusual in having much less computational work per byte of memory than any of the other benchmarks. As such, its checkpointing times are small and noisy, making it difficult to make performance conclusions about this benchmark.

FT's significant checkpoint size reductions from incremental checkpointing make Incr more efficient than Full. Furthermore, although FT's low pre-checkpoint overlap reduces the performance potential of CA (PR and CA show similar performance with Full-Block and Incr-Block), its low post-checkpoint locality makes Asynch-CA more efficient than Asynch-PR. LU has a similar profile and mostly similar checkpoint costs. It is unclear why its improved pre-checkpoint overlap does not result in better performance for Block-CA.

5.3 Compiler-Assisted OpenMP Checkpointing

This section looks at the cost of checkpointing for multi-threaded OpenMP applications, using the model checkpointing runtime.

5.3.1 Checkpoint Size

Figure 12 shows the checkpoint sizes optimized by the compiler analysis for input size A, normalized to the un-optimized checkpoint size. There is little change in the reduction as the number of threads increases. The only exceptions are EP and IS, where the savings increase with increasing thread counts.

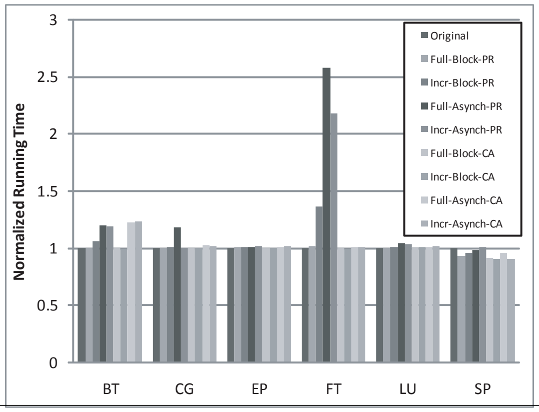

Figure 10. Execution times with all configurations of Lazarus, no data written (input size `A`)

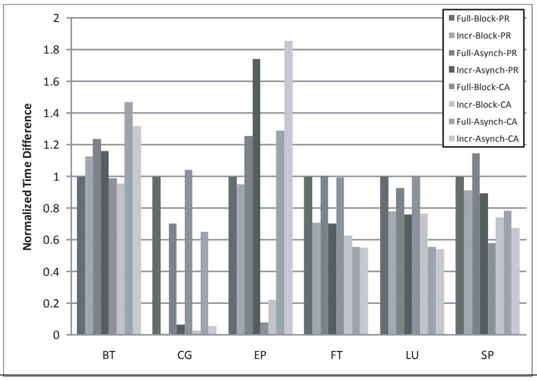

Figure 11. Checkpointing time for all configuration of Lazarus, data written to parallel file system (input size `A`)

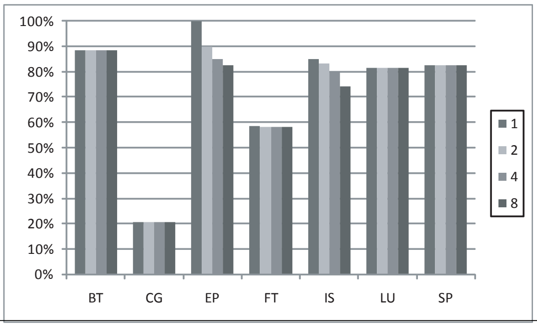

Figure 12. Checkpoint sizes generated by the model checkpointer (input size `A`)

5.3.2 Running Time - Incremental Checkpointing

Figure 13 shows the execution times of the NAS applications, running with the model checkpointer in configuration `Incr-Block` as a fraction of the execution time of `Full-Block`. We show only `CG` and `EP` on 4 threads, since the behavior of these codes is indicative of other codes that have either a small or a large checkpoint size reduction, respectively, and the number of threads has no effect on this behavior. The x-axis is the I/O bandwidth used in the experiments, ranging from 1 MB/s to 1 GB/s in multiples of 4, including a datapoint for infinite bandwidth. This range includes a variety of use-cases, including hard-drives (60MB/s write bandwidth) and 10 Gigabit Ethernet(1GB/s bandwidth). For `EP`, although there is some difference in performance between the two configurations, the effect is generally small and always $< 20\%$ at all bandwidths. However, for `CG`, the effect is quite dramatic, with the improvement from `IncrChkpt-Block` ranging from 95% for low bandwidths, when the cost of checkpointing is important, to $< 10\%$ high bandwidths.

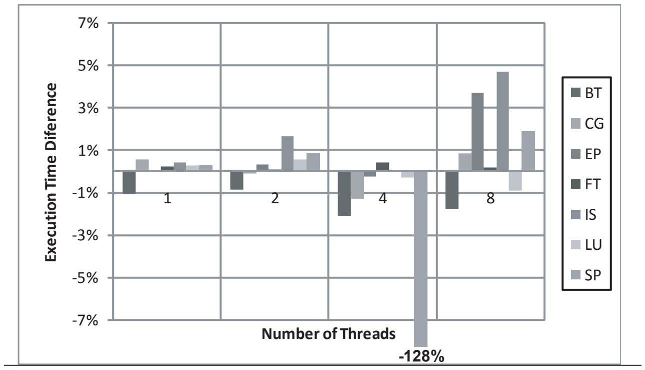

Figure 14. Relative speedup of `Incr-Asynch` relative to `Incr-Block` (BW=infinite, input size `A`)

5.4 Running Time - Asynchronous Checkpointing

We evaluated the performance of asynchronous checkpointing by comparing the execution time of applications that use `Incr-Asynch` to those that use `Incr-Block`. Figure 14 shows the % reduction in application running times from using with `Incr-Asynch` instead of `Incr-Block` for input size `A`. No data was saved to highlight the raw overhead of the runtime mechanisms required for blocking and asynchronous checkpointing. For most applications there is very little difference between the two configurations. The only exception is `SP` on 4 threads, where running with `Incr-Asynch` more than 2x slower than running with `Incr-Block`. The reason appears to be the addition of the extra thread. The nodes we were using have different memory banks associated with different pairs of processors. As such, an the extra checkpointing thread that is assigned to a processor that does not share a memory bank with the main computing processors will suffer from poor synchronization performance. In contrast, blocking checkpointing has no extra thread and requires no additional synchronization. Despite their similarities `BT` does not suffer from the same overhead as `SP` because `BT` uses a fewer number of larger arrays. The resulting 5x reduction in the number of `start_chkpt` calls removes synchronization as a performance bottleneck.

Figure 15 shows the same configurations but with the full range of bandwidths. Data for 2-thread and 4-thread runs is presented, since it is typical. Asynchronous checkpointing tends to perform better than blocking checkpointing for most bandwidths and applications, although the improvement does not hold in all cases. `CG` performs worse with asynchronous checkpointing for small bandwidths for all thread numbers and `SP` shows a large slowdown with asynchronous checkpointing for the reasons described above. As input sizes increase the performance gains of asynchronous checkpointing become more consistent but generally do not exceed 20%, while `SP` consistently shows very poor performance on 4 threads.

6. Summary

We have presented a novel compiler analysis for optimizing automated checkpointing. Given an application that has been augmented by the user with calls to a `checkpoint` function, the analysis identifies for each application array the regions in the code that do not have any writes to the array. This information is used to reduce the amount of data checkpointed and to asynchronously checkpoint this data in a separate thread. In our experiments with the NAS Parallel Benchmarks we have found that this analysis can reduce checkpoint sizes by as much as 80%. These checkpoint size reductions were found to have a notable effect on checkpointing performance. We compared the performance compiler-enabled checkpointing to pure-runtime checkpointing and identified several

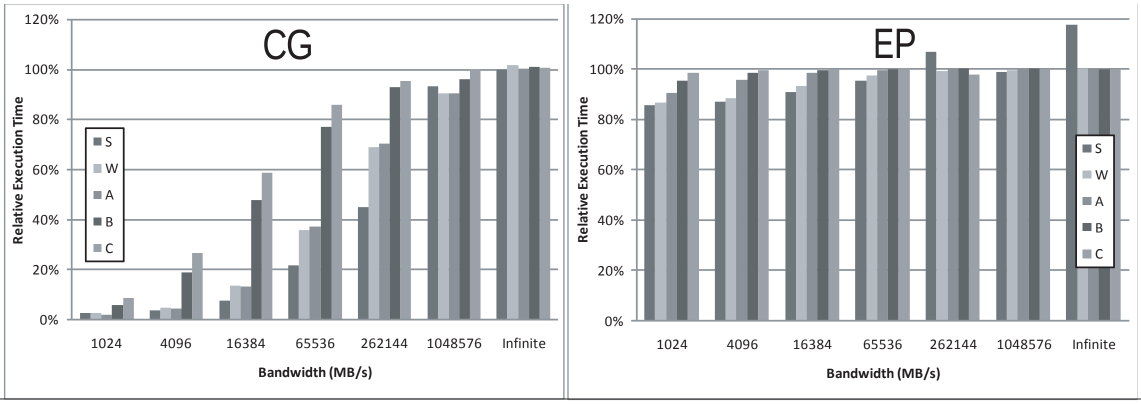

Figure 13. Execution time of `Incr-Block` relative to `Full-Block`

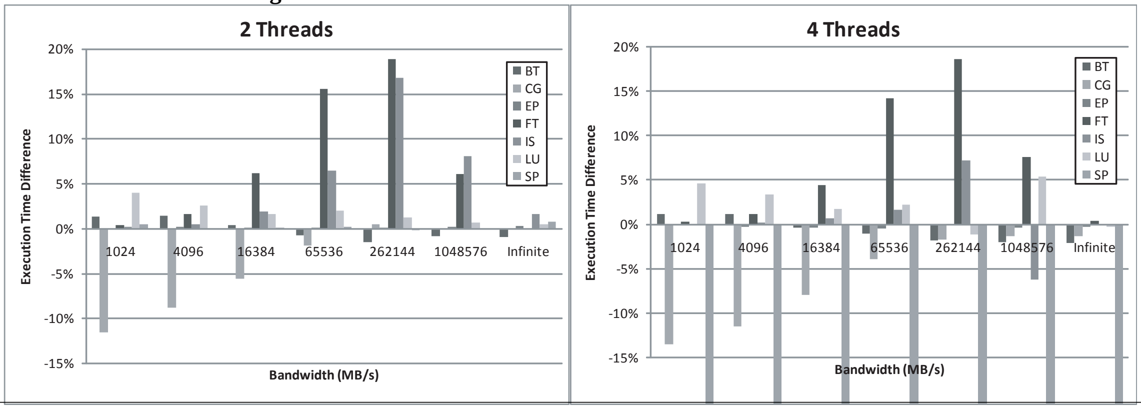

Figure 15. Execution time differences between `Incr-Asynch` and `Incr-Block` (input size `A`)

application characteristics that determine the optimal checkpointing technique for each type of application. Furthermore, our evaluation of compiler-enabled asynchronous checkpointing showed that asynchronous checkpointing is frequently better than blocking checkpointing, we discovered that this is oftentimes not the case, meaning that the choice of the optimal checkpointing technique closely depends on the application. These results also suggest that more work should be done to understand of the performance characteristics of asynchronous checkpointing runtime systems.

References

[1] http://phase.hpcc.jp/Omni/benchmarks/NPB.

[2] http://www.nas.nasa.gov/Software/NPB.

[3] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, and J. An overview of the bluegene/l supercomputer. In *IEEE/ACM Supercomputing Conference*, 2002.

[4] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th International Conference on Supercomputing (ICS)*, pages 277 – 286, 2004.

[5] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level checkpointing for shared memory programs. October 2004.

[6] Greg Bronevetsky, Paul Stodghill, and Keshav Pingali. Application-level checkpointing for openmp programs. In *International Conference on Supercomputing*, June 2006.

[7] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2006.

[8] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, November 2005.

[9] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratorys asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

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

[11] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.

[12] Kim Cupps Rob Ross, Jose Moreirra and Wayne Preiffer. Parallel i/o on the ibm blue gene /l system. Technical report, BlueGene Consortium, 2005.

[13] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 58, 2004.

[14] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[15] Kun Zhang and Santosh Pande. Efficient application migration under compiler guidance. In *Poceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 10–20, 2005.

This work performed under the auspices of the U.S. DOE by LLNL under Contract DE-AC52-07NA27344.

APPENDIX

A. OpenMP Theorems

`Lemma 1`: Let $n_1 \rightarrow^* n_k$ be a path in the CFG from node n_1 to node n_k such that $mayWillWriteBefore[n_k](array) = False$ and no node $n_2, ..., n_{k-1}$ contains a `checkpoint` call.
$\forall 1 < l < k.$
$mayWillWriteBefore[n_l](array) = True \Leftrightarrow$
$\forall 1 < p < l.\ mayWillWriteBefore[n_p](array) = True.$
`Proof`:
`Case` $\Rightarrow$:

- Suppose that $mayWillWriteBefore[n_l](array) = True$ and
$\exists 1 < p < l.\ mayWillWriteBefore[n_p](array) = False.$
- Let p' be the largest such number. As such, while $mayWillWriteBefore[n_{p'}](array) = False,$ $mayWillWriteBefore[n_{p'+1}](array) = True.$
- $mayWillWrite[n_{p'}](array) = \bigcup_{m \in succ(n_{p'})} mayWillWriteBefore[m](array).$
- Since $mayWillWriteBefore[n_{p'+1}](array) = True,$ we know that $mayWillWrite[n_{p'}](array) = True.$
- $mayWillWriteBefore[n_{p'}](array) = \llbracket n_{p'} \rrbracket(mayWillWrite[n_{p'}](array), array).$
- However, the only statement that can cause a $True$ to $False$ transition is a `checkpoint` call and we have assumed that this cannot happen.

`Case` $\Leftarrow$:

- Assume that $\forall.\ 1 < p < l.\ mayWillWriteBefore[n_p](array) = True.$
- This means that $mayWillWriteBefore[n_{l-1}](array) = True.$
- $mayWillWrite[n_l](array) = \bigcup_{m \in succ(n_l)} mayWillWriteBefore[m](array).$
- As such, $mayWillWrite[n_l](array) = True.$
- $mayWillWriteBefore[n_l](array) = \llbracket n_l \rrbracket(mayWillWrite[n_l](array), array).$
- Since we know that node n_l's statement cannot be a `checkpoint` call, it must be that $mayWillWriteBefore[n_l](array) = True.$

`Lemma 2`: Let $n_1 \rightarrow^* n_k$ be a path as above.
$\forall 1 < l < k.$
$mayWillWriteBefore[n_l](array) = False \Rightarrow$
$\forall l \leq p \leq k.\ mayWillWriteBefore[n_p](array) = False.$
`Proof`:

- Assume that $mayWillWriteBefore[n_l](array) = False.$
- Suppose that $\exists l \leq p \leq k. mayWillWriteBefore[n_p](array) = True$
- If $l = p$ we have a contradiction, since $mayWillWriteBefore[n_l](array) = False.$
- Otherwise, by `Lemma 1`, $mayWillWriteBefore[n_l](array) = True$, since $l < p$.
- This contradicts our assumption.

`Lemma 3`: Let $n_1 \rightarrow^* n_k$ be a path as above.
$\exists$ a `start_chkpt(array)` call along this path ($\exists 1 \leq l < k$ such that `start_chkpt(array)` appears immediately after node n_l or on the edge $n_l \rightarrow n_{l+1} \Rightarrow \forall 1 \leq i < l.\ mayWillWrite[n_i](array) = True$ and $\forall l \leq j \leq k.\ mayWillWriteBefore[n_j](array) = False.$

`Proof`:

- Since $\exists$ a `start_chkpt(array)` call along the path, we know that either
(i) $mayWillWriteBefore[n_l](array) = True\ \wedge$ $mayWillWrite[n_l](array) = False$ or
(ii) $mayWillWrite[n_l](array) = True\ \wedge$ $mayWillWriteBefore[n_{l+1}](array) = False.$
- $i:\ 1 \leq i < l$:
  - If (ii) is true, we know that $mayWillWriteBefore[n_l](array) = True$ because $mayWillWriteBefore[n_l]](array) = \llbracket m \rrbracket(mayWillWrite[n_l]](array), array)$ and the node n_l is not a `checkpoint` call by assumption.
  - As such, in both cases: $mayWillWriteBefore[n_l](array) = True.$
  - From `Lemma 1` we know that $\forall 1 < p \leq l.\ mayWillWriteBefore[n_p](array) = True.$
  - $mayWillWrite[n](array) = \bigcup_{m \in succ(n)} mayWillWriteBefore[m](array).$
  - As such, $\forall 1 \leq i < l.\ mayWillWrite[n_i](array) = True.$
- $j:\ l \leq j < k$
  - If (i) is true, we know that $mayWillWriteBefore[n_{l+1}](array) = False$ because $mayWillWrite[n_{l+1}](array) = \bigcup_{m \in succ(n_{l+1})} mayWillWriteBefore[m](array).$
  - As such, in both cases: $mayWillWriteBefore[n_{l+1}](array) = False.$
  - According to `Lemma 2`, $\forall l \leq p \leq k.\ mayWillWriteBefore[n_p](array) = False$

`Theorem 1`: Let $n_1 \rightarrow^* n_k$ and $n'_1 \rightarrow^* n'_k$ be two paths as above, with $n_1 = n'_1$ and $n_k = n'_k$.
If $\exists$ a `start_chkpt(array)` call along one path then $\exists$ a `start_chkpt(array)` call along the other path.
`Proof`:

- Assume that $\exists$ a `start_chkpt(array)` call along path $n_1 \rightarrow n_2 \rightarrow^* n_{k-1} \rightarrow n_k$.
- From `Lemma 3` we know that $\forall 1 \leq i < l.\ mayWillWrite[n_i](array) = True$ and $\forall l \leq j \leq k.\ mayWillWriteBefore[n_j](array) = False.$
- As such, we know that $mayWillWrite[n_1](array) = True$ and $mayWillWriteBefore[n_k](array) = False.$
- Let l be the largest number s.t. $1 < l < k$ and $mayWillWrite[n_l](array) = True.$
- Thus, either (i) $l = k$ or (ii) $mayWillWrite[n_{l+1}](array) = False.$
- Suppose (i):
  - By the `start_chkpt` placement rules of Figure 3, `start_chkpt(array)` would be placed on edge $n_l \rightarrow n_k$.
- Now suppose (ii):
  - Either $mayWillWriteBefore[n_{l+1}](array) = True$ or $= False.$
  - If $= True$, the `start_chkpt` placement rules of Figure 3 would place `start_chkpt(array)` immediately after n_{l+1}.
  - If $= False$, the rules would place `start_chkpt(array)` on edge $n_l \rightarrow n_{l+1}$.

`Theorem 2`: Let n_1 be the start of the application or a `checkpoint` call and let n_k be a `checkpoint` call or the end of the application. For any two paths through the CFG from n_1 to n_k that do not contain `checkpoint` calls, if one contains a call to `start_chkpt(array)`, so does the other.
`Proof`: Direct application of `Theorem 1`.