Man vs. Machine : Comparing Handwritten and Compiler-generated Application-Level Checkpointing*

James Ezick, Daniel Marques, Keshav Pingali, Paul Stodghill
Department of Computer Science
Cornell University
Ithaca, NY 14853
{ezick,marques,pingali,stodghil}@cs.cornell.edu

October 12, 2004

Abstract

The contributions of this paper are the following.

- We describe the implementation of the C^3 system for semi-automatic application-level checkpointing of C programs. The system has (i) a pre-compiler that instruments C programs so that they can save their states at program execution points specified by the user, and (ii) a novel memory allocator that manages the heap as a collection of pools.
- We describe two static analyses for reducing the overhead of saving and restoring the application state. The first one optimizes stack variables, while the second one optimizes heap data structures.
- To benchmark our system, we compare the overheads introduced by our semi-automatic approach with the overhead of handwritten application-level checkpointing in an n-body code written by Joshua Barnes. Except for very small problem sizes, these overheads are comparable.

*This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

- We highlight various algorithmic challenges in the optimization of application-level checkpointing that should provide grist for the mills of the PLDI community.

1 Introduction

The running times of many applications are now exceeding the mean-time-between failure (MTBF) of the underlying hardware. For example, protein-folding using *ab initio* methods on the IBM Blue Gene is expected to take a year for a single protein, but the machine is expected to lose a processor every day on the average. As a result, software needs to be resilient to hardware faults.

Checkpointing is the most commonly used technique for fault tolerance. The state of the running application is saved periodically to stable storage; on failure, the computation is restarted from the latest checkpoint. Checkpointing comes in two very different flavors: *system-level checkpointing (SLC)* and *application-level checkpointing (ALC)*.

SLC saves the bits of the machine state to stable storage [13, 1]. On large, parallel machines, this can be a lot of bits, and the overhead of saving them is large. In most applications however, there are some key data structures from which the entire computational state can be recovered. In ALC, these data structures are saved and restored directly by the application [14]. For example, the program on the IBM Blue Gene saves only the positions and velocities of all the bases in the protein since the entire computational state can be recovered from this information [11]. Instead of saving terabytes of data, it saves only a few megabytes.

Both approaches have advantages and disadvantages. SLC can be transparent to the applications programmer, while ALC requires the developer to write code to save and restore the application state. This code can be tedious to write and debug. On the other hand, SLC must save the entire process state, while ALC exploits domain knowledge in order to save only the core application state.

In this paper, we describe a system called C^3 for semi-automatic application-level checkpointing of C programs. The system has (i) a pre-compiler that instruments C programs so that they can save their states at program execution points specified by the user, and (ii) a novel memory allocator that manages the heap as a collection of pools.[1] We describe two static analyses for reducing the overhead of saving and restoring the application state. The first one optimizes stack variables, while the second one optimizes heap data structures.

To evaluate the effectiveness of our techniques, we studied one application in depth. This application, called `treecode`, is an n-body simulation written by Joshua Barnes. We chose it because it is a non-trivial code, and because it contains hand-written state saving code, which provides us with a benchmark. Our experiments show that for all but the smallest problem sizes, the overheads are comparable.

The rest of the paper is organized as follows. Section 2 describes the features of the `treecode` application that are relevant to this paper. In Section 3, we describe the C^3 pre-compiler and memory allocator. Section 4 describes the analysis that we use to reduce the number of lexical variables checkpointed. In Section 5, we describe our approach to reducing the amount of heap data that is saved. In Section 6, we describe related work, and in Section 7, we discuss future work.

2 Treecode

Figure 1 presents a skeletal overview of `treecode` [2], an application for conducting n-body simulations using a hierarchical force calculation algorithm written in ANSI C. It consists of 7 source files and 7 header files, comprising a total of approximately 3000 lines of source code.

Unlike $O(N^2)$ direct sum methods that completely calculate the force that each body asserts on the others, `treecode` partitions the bodies using an oct-tree, such that each node of the tree describes the bodies within a spatial volume, termed a *cell*. The use of such a structure allows for a "reasonably accurate" approximation of the forces exerted by the bodies in $O(NlogN)$ time.

The non-leaf nodes of the oct-tree are represented by instances of the `cell` structure; the leaves are instances of `body`. Both are subtypes (of a sort) of the `node` structure. A `cell` contains (up to) 8 pointers to its descendants, each one either a `body` or another `cell`. In addition to this tree structure, each of the nodes of the tree contains a `next` pointer, which is used to create

[1] A paper in PPoPP 2003 describes our system for *parallel* non-blocking application-level checkpointing; the C^3 system is a component of this bigger system.

```
void *pvec;
body *bodytab;
cell *root;
node *active;
cell *interact;


int main(){
     pvec = calloc(...);
     bodytab = calloc(...);

     while(...){
        maketree(bodytab);
        gravcalc();
        savestate(); // manual checkpointing
     }
}

void maketree(){
     for(...)
        if(...)
           free(cell)

     #pragma ccc PotentialCheckpoint

     for(...)
        if(...)
           cell = calloc(...);
}

void gravcalc(){
     active   = calloc(...);
     interact = calloc(...);

     while(...){
        ...
     }

     free(active);
     free(interact);
}
```

Figure 1: Overview of `treecode`

a linked list of all a `cell`'s children. The `next` pointer of the last element on such a list is then set to point to the same object as the parent's `next`. Each `cell` also contains a pointer, `more`, which points to the head of its child list. The `next` and `more` pointers create a *threading* of the tree's nodes, allowing the tree to be walked as a list. This threading was constructed so that a "tree search can be performed by a simple iterative procedure." A diagram of such a tree is in Figure 2. In that picture the `more` pointers are not shown, but can be assumed to be the same as the pointer to each `cell`'s left-most child.

When the application begins, it allocates an array to hold the specified number of `bodies`, and then initializes each of them. Then the simulation is conducted for the requisite number of time steps, each one computed by an iteration of the loop in `main()`. Each iteration has two main components. First, `maketree()` is called to construct an oct-tree with the `bodies` as its leaves. Then, `gravcalc()` is called to walk the tree to discover which `nodes` interact with each other and to calculate the forces acting on each `body`.

The function `maketree()` first deallocates all the `cells` in the existing tree by walking the threaded pointers. These objects are not explicitly deallocated but rather are placed on the `freecell` list to be reused as needed. Then a new tree is constructed, reusing objects from the `freecell` list when a new `cell` is needed. Only if that list is empty does `treecode` allocate more memory via a call to `calloc()`. Because of this behavior, `treecode` can be described as using a custom memory allocator.

The `gravcalc()` function calculates how the bodies interact with each other. It allocates two temporary arrays: `interact`, which contains lists of all the nodes that interact with a particular `body`, and `active`, which lists all the nodes that need to be examined when constructing these interaction lists [2]. It then walks the tree, determining the interactions between bodies, so that it can calculate the forces acting on each.

The application can be run in a mode where it will save its state to disk at the end of each iteration, via a call to `savestate()`. In the event of a failure, the restarted application can read that information and use it to resume at the next iteration. This example of ALC showing the efficiency of that technique - although the `cells` of the tree have not yet been deallocated, the knowledge that they will be allows `savestate()` to ignore them.

Because `treecode` constructs an 8-way tree, if the tree was complete the number of internal nodes would be approximately $\frac{1}{8}$ of the number of leaves; therefore, eliminating the `cells` from the checkpoint might only increase

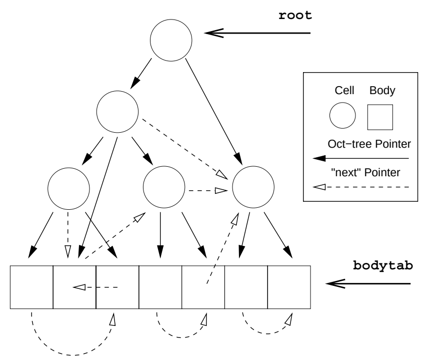

Figure 2: `treecode` tree data structure

Size	Configuration	**Time**		**Chpt Size**	
		Sec.	ovrhd	MB	ovrhd
10^4	Original - no chpt	3.61	-	N.A.	N.A.
	Original - chpt	4.70	30.2%	0.5	-
10^5	Original - no chpt	54.43	-	N.A.	N.A.
	Original - chpt	63.53	16.7%	4.97	-
10^6	Original - no chpt	714.95	-	N.A.	N.A.
	Original - chpt	804.66	12.6%	49.59	-

Table 1: Runtimes and Checkpoint Size

performance by $\frac{1}{9}$. However, the "fullness" of the tree is dependent on the underlying physics of the model being simulated: when using the included Plummer model generator to initialize the bodies, it appears that the average fullness is between 2 and 3 children per `cell`.

Table 1 reports the execution time of the `treecode` application running three different sized n-body simulations, (10^4, 10^5, and 10^6), each using the provided Plummer model to initialize the bodies. The table also shows the runtime required for the application when run in state-saving mode, the overhead that such checkpointing adds to the runtime, and the size of the average checkpoint file. For the largest simulation size, the hand-written checkpointing code adds 12.6% to the execution time. For smaller simulations, the overhead is greater.

One note: `treecode`'s original fault-tolerance routine used the C standard library function `fwrite()` to save checkpoint data. We converted this to use the `write()` system call instead. We also broke very large writes up into a series of smaller calls, each writing a page (4 KB) of data. For our test system, this transformation vastly improved the performance of the application's state saving routine, halving the time required to write the checkpoint. We needed to make this transformation in order to ensure that comparisons with our automatic checkpointing system were fair (C^3 checkpointing code uses a similar mechanism).

3 The $\mathbf{C^3}$ System

C^3 is a system for automatically adding ALC code to a C language program. It consists of two components, a source-to-source compiler (the C^3 pre-compiler) that converts the code of an application into that of a semantically consistent yet fault-tolerant version, and a library (the C^3 runtime) that contains fault-tolerant implementations of the standard C library functions, and some utility functions used by the inserted code. The output of the pre-compiler is then passed to the native compiler, where it is compiled and linked with the runtime, producing a fault-tolerant application.

The C^3 system has been designed to provide efficient checkpointing for all the constructs in the C language specification. Portable checkpointing systems (where a checkpoint taken on one architecture can be restarted on another), such as Porch [15], need to save checkpoint data in an architecture and operating system neutral format. On the other hand, C^3 saves a program's variables as binary data. Although this limits C^3 to providing homogeneous checkpointing (the application can only be restarted on a machine of identical OS and architecture), it allows for the efficient saving of data, and does not require complete type information. Requiring complete type information limits input programs to a subset of the C language that we believe is not sufficient for "real-world" computational science applications. `treecode`, in fact, uses ambiguous pointers.

The C^3 system requires no modifications to the input program, except that the locations where checkpointing should occur are marked with a pragma statement (`#pragma ccc PotentialCheckpoint`). These represent *potential* checkpoint locations: when execution reaches such a location, the runtime system determines if a checkpoint should indeed be taken. Because the modified application is only capable of checkpointing and restarting at the specified locations, static analysis can be used to reason about the behavior of the application at those points, and the pre-compiler can use the results of such analysis to optimize checkpointing.

For `treecode` we chose to place the checkpoint location in `maketree()`, right after the `cell`'s are freed. Compare this location with where the developer placed the call to `savestate()`, inside the main loop. At that point, the `cell`'s are still live, but the developer knows that they do not need to be saved. The analysis that we describe in Section 5 is not currently able to deduce this, so we have placed our checkpoint after the `cell`'s have been freed.

Because the C^3 pre-compiler can only add fault-tolerance to the code that it is invoked upon, if the application code utilizes a "state-full" library, a fault-tolerant version of that library must be provided for the application to correctly restart (the memory allocator is one such example). The C^3 runtime contains fault-tolerant versions of the standard C library calls. Additionally, we have developed a fault-tolerant version of the MPI library [6, 7].

3.1 The C^3 Pre-compiler

The code that the pre-compiler inserts must ensure that the application resumes at the instruction immediately following where the checkpoint was taken and that the application's variables are saved and restored correctly. Because a program's variables are saved as binary data, on restart the system must force each variable to be restored to its original address. This is necessary so that a variable dereferenced as a pointer will point to the proper object after restart. This requires two separate mechanisms.

3.1.1 Checkpointing the application's position

The C^3 system uses a data structure, called the *Position Stack* (PS), to record and recreate the application's position in both its dynamic execution and its static program text. At each checkpoint location in the code, the pre-compiler inserts a unique label. Additionally, a call-graph analysis is performed and a label is inserted before every function call that might eventually lead to such a location. The pre-compiler also inserts code to push and pop values onto the PS as these labels are encountered during execution. If a potential checkpoint location is reached, and a checkpoint is taken, the runtime saves the PS to the checkpoint file. In such a manner each checkpoint contains a record of the call sequence that led to the specific checkpoint location for which the data in the checkpoint file corresponds to.

Immediately upon restart, the runtime system pads the stack via calls to `alloca()`, such that all successive functions have their stack frames at the same addresses they had before the checkpoint. Then it restores the PS before handing control to the original `main()` function. Each procedure, in turn, uses the PS to call the same function that it had called immediately before the checkpoint was taken. When control arrives in the innermost function, the application jumps to just below where the checkpoint was taken. In such a manner, the stack is rebuilt with the local variables occupying the same addresses as they had before the restart, the program's dynamic position is as it was when the checkpoint was taken, and its position in the static text is restored to the point immediately following the code that saved the checkpoint.

3.1.2 Checkpointing the application's data

The pre-compiler uses another structure, the *Variable Description Stack* (VDS), to save and restore the values held by the stack variables. At the location where a variable enters scope, the pre-compiler inserts code to push the variable's address and size onto the VDS. Where a variable leaves scope, code is inserted to pop that the record from the VDS. As an optimization, the C^3 pre-compiler will rename and lift nested scoped local variables up to the function-scope level. This ensures that a variable scoped to a loop body will not be unnecessarily pushed and popped in each iteration of the loop. Figure 3 shows such manipulations.

```
function(int a) {
  int b[10];
  {
    int c;
    ...
  }
}

Before Pre-Compiler
```

```
function(int a) {
  int b[10];
  int c;
  VDS.push(&a, sizeof(a));
  VDS.push(&b, sizeof(b));
  VDS.push(&c, sizeof(c));
  {
    ...
  }
  VDS.pop(3);
}

After Pre-Compiler
```

Figure 3: Manipulating the *VDS*

When a checkpoint is taken, for each item on the VDS, the C^3 runtime copies the specified number of bytes from the specified address to the checkpoint. It also saves the VDS as part of the checkpoint. On recovery, after the stack is rebuilt, the VDS is restored and used to copy the values from the checkpoint file back to the proper addresses.

3.2 The C^3 Runtime

The C^3 runtime is a set of functions which perform two different duties - they are responsible for the saving and restoring of application state, and they provide a fault tolerant implementation of the standard C library. The most interesting of these functions are those that implement the memory allocator: these are the only ones that we discuss in detail. To employ these functions, the pre-compiler converts all calls to the native allocator (`malloc()`, `free()`, etc.) to the version provided in the C^3 runtime (`CCC_malloc()`, `CCC_free()`, etc.).

3.2.1 The C^3 Allocator

In addition to the usual requirements of providing an application with an efficient mechanism to support the creation and freeing of dynamic memory objects, C^3's allocator must ensure that when an application is restarted from a checkpoint, every allocated object will be restored to the same address that it originally held, that all such objects contain the same data as they did at checkpoint time, and that future calls to malloc and free behave correctly.

The C^3 allocator manages the heap objects in a pool of memory that it requests from the operating system. For simplicity's sake, we model that pool as a contiguous region of bytes; however, in actuality the pool consists of a collection of contiguous regions, called sub-pools, which may or may not be contiguous with one another. On restart, the C^3 system requests the same pool of memory from the operating system, copies objects' data from the checkpoint file into the proper addresses, and reconstructs the free lists.

SLC systems save the heap by writing the entire region of memory that the native allocator had control over to the checkpoint file. An advantage that C^3 has over such systems is that, because it implements its own memory allocator, it only needs to save the portion of the pool that was ever actually used. Another, even greater advantage is that C^3 does not need to save the objects that have been deallocated by the application. For certain codes, the amount of deallocated memory can be significant; not saving that memory could dramatically decrease the overhead of taking a checkpoint. The allocator still needs to ensure that future calls to `malloc` and `free` behave as expected.

Although the presence of deallocated objects allows the C^3 allocator to save less data, the overhead of checkpointing the heap is not just a function of the amount of data to be saved. We illustrate this with a sample application that is `treecode`-like in its memory requirements - it first allocates 2,000,000 objects of 64 bytes each, and then frees alternate ones. We implemented three different heap-saving algorithms, and applied them to this sample application. The runtime in seconds for these three algorithms is shown in Figure 2.

These results, and all others presented in this paper, were obtained on a 2.20 GHz Intel Xeon based system containing 1.0 GB of RAM. Hyper-Threading was disabled for these measurements. Microsoft Windows Server 2003 was the operating system, and all code was compiled with the MinGW version of the gcc 3.2.3 compiler, with the optimization level set to -O3. The checkpoints were written to a network file server over a 100 Mbit Ethernet connection.

Algorithm	Time, seconds
Naive	1027.91
Copy to buffer	26.33
C^3 2 color	22.94

Table 2: Runtimes for three algorithms for excluding freed object

The first algorithm implemented a "naive" strategy: starting at the first object on the heap, visit each object, if an object has not been deallocated, save its address, size, and data to the checkpoint file. The time to checkpoint the heap in this method was over 1000 seconds. This astronomical time is due to the very high number of system calls that the checkpointer makes - three per live object.

The second strategy used a similar algorithm, but instead of writing objects to the checkpoint file as they are encountered, they are copied to a buffer. That buffer is then saved to the checkpoint file, in chunks of pages. For this strategy, the runtime falls to below 26.4 seconds.

We believe that to efficiently checkpoint the heap, the system needs to quickly partition allocated and deallocated objects at checkpoint time. The third algorithm, which is used by C^3, uses multiple, disjoint memory pools. The motivation behind this concept is that, if the objects in the program can be partitioned so that, at a checkpoint, all of the objects allocated to a particular pool have been freed then this pool can be trivially excluded from the checkpoint. In order to keep the processing cost small, pools that have at least one live object are saved entirely. By carefully assigning objects to multiple pools, we gain the benefit of using a contiguous buffer of non-free objects, without needing to perform any copying. This third algorithm required only under 23 seconds, a 13% improvement over the second strategy.

The C^3 allocator manages a fixed number of memory pools, each with its own region of address space, with no page belonging to more than one pool. The C^3 allocation routines all take an extra parameter, the *color*, which specifies the pool into which the new object should placed.

Each pool has its own free list and a counter, *live count*, that keeps track of how many objects in that pool have not yet been freed. If, at checkpoint time, a pool's live count is zero, then all of the objects in that pool have been deallocated, and the pool does not need to be saved.

Clearly, carefully assigning objects to colors is necessary to obtain good performance. An optimally bad assignment of colors would not only require the contents of every pool to be saved to disk, but could potentially obliterate the performance improvement that comes from reusing reclaimed objects that might now reside is a disjoint pool. Since there is a minimal overhead associated with saving a pool there is an incentive to prevent a few small objects from being allocated to a pool of their own. Finally, the number of pools is bounded by a fixed value at compile time. While the opportunity for a performance gain from a good coloring is substantial, these competing pressures, taken over the space of perhaps many checkpoints, makes finding a coloring a potentially very hard problem. In Section 5 we present one possible technique for deriving a good coloring.

3.3 Overhead

The C^3 system adds two different kinds of execution time overhead: (1) the cost of executing the compiler inserted code and using C^3's heap implementation, and (2) the cost for taking a checkpointing and writing it to disk.

One change that we make to the `treecode` application (before feeding it to the C^3 compiler) is to explicitly deallocate the `cell` objects, via a call to `free()` rather than place them on the application's internal free list.

Size	Configuration	Time	
		Sec.	ovrhd
10^4	Original - no ckpt	3.61	-
	C^3 - no ckpt	3.67	1.7%
10^5	Original - no ckpt	54.43	-
	C^3 - no ckpt	54.53	0.2%
10^6	Original - no ckpt	714.95	-
	C^3 - no ckpt	716.41	0.2%

Table 3: Non-checkpointing overheads

This transformation is semantically correct because none of these objects are ever accessed between their placement on and removal from the list. We justify making this alteration because recent work [5] has shown that custom allocators often degrade performance for most applications.

The reason for this change is because, by explicitly calling `free()` (by conversion `CCC_free()`) the C^3 system is informed that such an object is no longer in use, and could use that knowledge to optimize the checkpointing of the heap.

3.3.1 The overhead of the transformations

Table 3 shows the non-checkpointing runtime, in seconds, of both the original `treecode` application and the version produced by C^3. The results are for six iterations of three different sized n-body simulations, 10^4, 10^5, and 10^6, where the initial conditions of the bodies was produced by the built-in Plummer model generator.

The difference in runtimes includes the costs of (1) using the C^3 memory allocator, (2) explicitly deallocating the objects originally placed on the `freecell` list, (3) executing the code to manage the VDS and PS, and (4) checking if the application needs to take a checkpoint or if it is in recovery mode.

For the largest problem size, the overhead that the C^3 system added to the original `treecode` application, is $\frac{2}{10}$ of 1%. The fact that the C^3 source transformations, and the C^3 memory allocator add very little overhead to the application means that, for the goal of providing efficient fault-tolerance, we

Size	Configuration	**Time**		**Chpt Size**	
		Sec.	ovrhd	MB	ovrhd
10^4	Original - no chpt	3.61	-	N.A.	N.A.
	Original - chpt	4.70	30.2%	0.5	-
	Baseline C^3	6.14	70.2%	1.09	118.7%
10^5	Original - no chpt	54.43	-	N.A.	N.A.
	Original - chpt	63.53	16.7%	4.97	-
	Baseline C^3	71.55	31.4%	8.64	74.1%
10^6	Original - no chpt	714.95	-	N.A.	N.A.
	Original - chpt	804.66	12.6%	49.59	-
	Baseline C^3	868.18	21.4%	83.38	68.1%

Table 4: Runtimes and Checkpoint Size, C^3 Baseline

only need to concern ourselves with the overhead of the actual state saving routines.

3.3.2 The overhead of state-saving

Table 4 measures the overhead that take checkpoints adds to the `treecode` application. For the same simulations as above, we compare the overhead added by `treecode`'s own state-saving code to the overhead added by the C^3 version. The times measured here include the time to take a checkpoint once each iteration.

The rows labeled "Original - no chpt" shows the running time of the original `treecode` with state-saving turned off. Running time overheads are measured relative to these rows. The rows labeled "Original - chpt" shows the running time and checkpoint sizes of the original `treecode` with state-saving turned on. Checkpoint size overheads are measured relative to these rows. The rows labeled "Baseline C^3" show the running time and checkpoint sizes of the code emitted by C^3 without any of its optimizations enabled and using only one memory pool.

Observe that for the 10^6 sized simulation, the manually written checkpointing code saves an average checkpoint size of just below 50MB, and imposes an overhead of 12.5% on the runtime of the, non-fault-tolerant version. The C^3 generated version writes an average checkpoint of more than

83MB and imposes an overhead of 21.5% on execution time. The differences in checkpoint size and execution time between the handwritten and compiler generated code is fairly large. Primarily, this is because the handwritten fault-tolerance takes advantage of the fact that it does not need to save the `cell`s which are on the free list.

The following sections of this paper discuss static and dynamic techniques that are used to reduce the amount of checkpoint data.

4 Optimizing Lexicial Variables

Previous work [3] has shown that checkpoints can be reduced by performing a static liveness analysis over the set of variables in the program to determine, for each checkpoint, the set of variables whose values are required after the checkpoint. Variables that are not required can safely be excluded from the checkpoint. Our work differs in that, rather than using the analysis to make a static decision about each variable at each checkpoint, we use the analysis as a driver for a three-tiered approach to deciding this question.

In this section, we will first describe our context-sensitive liveness analysis and then describe how it is used to drive our optimizations.

4.1 Analysis

A liveness analysis requires that, for each program statement, the set of locations that may be used and the set that must be defined are identified. We define these sets as,

Use(s) is the set of locations whose value may be used in the evaluation of the statement s. For pointer expressions, this may include both the pointer location as well as the location pointed-to by the pointer.

Def(s) is the set of locations that must be defined in the evaluation of the statement s. By convention, the set for the statement at the beginning of any lexical block includes all of the locations that are entering scope. For assignments through pointers, this set includes the target of the pointer only if the pointer target can be unambiguously determined.

These sets can be determined by a local syntactic analysis that makes use of an underlying pointer analysis [16]. For this paper, compound locations such as arrays and structures are treated monolithically.

Data-Flow Equations:

$$\phi_s = \begin{cases} id & \text{if } s \text{ is } P^{exit} \text{ for some procedure } P \\ F_s \circ \phi_{P^{entry}} \circ \phi_{s^{ret}} & \text{if } s \text{ is a } call \text{ to procedure } P \\ F_s \circ (\bigsqcup_{s' \in succs(s)} \phi_{s'}) & \text{otherwise} \end{cases}$$

$$\text{where } F_s = \lambda X.Use(s) \cup (X - Def(s))$$

Operations on Data-Transforming Functions:

Data-flow functions:	$F_s = (Use(s), Def(s))$
Initial function:	$\bot = (\emptyset, \mathcal{U})$ where $\mathcal{U}$ is the universal set of locations
Identity function:	$id = (\emptyset, \emptyset)$
Application:	$(G, K)(X) = G \cup (X - K)$
Union confluence:	$(G_1, K_1) \sqcup (G_2, K_2) = ((G_1 \cup G_2), (K_1 \cap K_2))$
Composition	$(G_1, K_1) \circ (G_2, K_2) = (G_1 \cup (G_2 - K_1), K_1 \cup K_2)$
Canonical form:	$\langle (G, K) \rangle = (G, K - G)$

Figure 4: Context-Sensitive Liveness Analysis

Given these sets, the liveness analysis is performed by computing the least-fixed point of the second-order equations shown in the top part of Figure 4. This fixed point is computed over the interprocedural control flow graph of the program using the usual lattice of functions. The analysis is context-sensitive modulo the flow-insensitive pointer analysis we use to construct the Def and Use sets.

This analysis is efficient since each liveness function can be represented by a pair of variable sets, (G, K). Each operation required to compute the fixed point is then reduced to a constant number of set operations. A canonical form reduces the necessary equality test to syntactic equality. The complete list of the operations is shown in the lower part of Figure 4. Given a statement, s, and a stack-context for s, $\bar{\sigma} = \sigma_0 \ldots \sigma_n$, where each σ_i is a call-statement, the set of live variables associated with s in context $\bar{\sigma}$ is

$$L(s, \bar{\sigma}) = \phi_s \circ \phi_{\sigma_n^{ret}} \circ \ldots \circ \phi_{\sigma_0^{ret}}(\emptyset),$$

where σ_i^{ret} refers to the return-statement corresponding to the call-statement σ_i.

4.2 Optimizations

Liveness analysis can be used to answer questions about the relationship between checkpoints and live variables:

1. Given a variable, v, is there any checkpoint at which v is live in some valid context?
2. Given a variable, v, and a specific checkpoint, c, is there any valid context in which v is live at c?
3. Given a variable, v, a specific checkpoint, c, and a specific context, $\bar{\sigma}$, for c, is v live at c in context $\bar{\sigma}$?

These questions are the basis of a tiered system of checkpointing optimizations. Questions 1 and 2 can be answered by performing a live-context analysis to merge the live variable sets for each live context at each statement into a single set for that statement. Question 3 must be answered at runtime, as described below.

The first tier of optimization identifies variables that are never live at any checkpoint statement in the program. Since these variables are not used after any checkpoint, the VDS push and pop instructions for these variables may be eliminated.

The second tier of optimization identifies, for each checkpoint c, variables that are not live at that checkpoint. A list of these variables is constructed at compile-time and passed to the C^3 runtime, which safely excludes these variables from all checkpoints taken at c.

The third tier of optimization is for variables whose liveness at a particular checkpoint is context-dependent. In this case, a finite automaton is statically derived by converting the data-transform functions of the analysis to a state transition function over the state space of possible contexts [9]. Accepting states are then the contexts such that the variable appears in the output of the data-transform function associated with the checkpoint. At checkpoint-time, the automaton is executed with the actual dynamic stack context, PS, that led to the checkpoint, and the variable is then included or excluded from the checkpoint accordingly.

By choosing an optimization strategy based on the liveness characteristics of each variable, we are able to minimize the runtime overhead of the state-saving mechanism while at the same time retaining the ability to utilize the full power of the context-sensitive analysis. This is a capability that is unique to our system.

4.3 Experiments

For treecode, the total analysis, including the computation of the Def/Use sets requires about five seconds, with less than half of that going to computing the fixed point. Because of the representation we use for the data-transforming functions, it is actually faster to compute all of the functions explicitly than to use any demand-based techniques for context-sensitive analyses. Since there is only a single checkpoint and the live variable set at that checkpoint is context-independent, only first tier optimizations are performed on this code.

Figure 5 lists the live variable at the checkpoint statement in treecode. Variables marked "yes" in the column labeled *treecode?* are those variables saved by the manual state-saving mechanism provided in the code.

Notice the variables that our analysis marks as live but that are not included in treecode's state-saving,

- btab and nbody are formal parameter that contain copies of global variables passed to the function where we take a checkpoint. Their inclusion is a consequence of our checkpoint location.
- The variable cpustart stores a time that is used when each iteration terminates to compute the elapsed time of the iteration. The manual restoring mechanism restores to a point that recomputes this value.
- The values of ncell and firstcall are constant at each invocation of the checkpoint.
- File pointers outfile and savefile are respecified when the program is manually restarted.
- namebuf is a buffer that will be completely overwritten after the checkpoint before it is used. Our construction of *Def* sets treats arrays monolithically and is unable to detect this.
- The standard stream variables are automatically reinitialized by the C library during a recovery and are never explicitly recorded in the VDS or saved at a checkpoint.

Using the liveness results to eliminate VDS pushes and pops, the size of the saved lexical variable set is reduced from 748 bytes to 160 bytes, a reduction of 78%. In addition, the size of the VDS, which is saved at each checkpoint, was reduced from 476 bytes to 212 bytes. Taken together, our optimization system reduced by more than 75% the total storage required to save and restore the static memory.

Variable	treecode?	Comments
Global Variables		
bodytab	yes	
dtime	yes	
dtout	yes	
eps	yes	
ncell	no	always 0 at checkpoint
nbody	yes	
nstep	yes	
options	yes	
outfile	no	respecified at restart
rsize	yes	
savefile	no	respecified at restart
theta	yes	
tnow	yes	
tout	yes	
tstop	yes	
usequad	yes	
File Static Variables		
paramvec	yes	
progname	yes	
Local Variables		
btab	no	copy of global bodytab
cpustart	no	stores timing information
nbody	no	copy of global nbody
Local Static Variables		
firstcall	no	always FALSE at checkpoint
namebuf	no	fully overwritten before use
Standard Streams		
stderr	no	handled by C library
stdin	no	handled by C library
stdout	no	handled by C library

Figure 5: Result of Liveness Analysis at Checkpoint

Table 5 shows the aggregate performance results, with new rows labeled "+ stack opts.", which give the checkpoint size and corresponding execution time of the `treecode` with the optimizations described in this section enabled.

Size	Configuration	**Time**		**Chpt Size**	
		Sec.	ovrhd	MB	ovrhd
10^4	Original - no chpt	3.61	-	N.A.	N.A.
	Original - chpt	4.70	30.2%	0.5	-
	Baseline C^3	6.14	70.2%	1.09	118.7%
	+ stack opts.	6.19	71.4%	1.08	118.5%
10^5	Original - no chpt	54.43	-	N.A.	N.A.
	Original - chpt	63.53	16.7%	4.97	-
	Baseline C^3	71.55	31.4%	8.64	74.1%
	+ stack opts.	70.97	30.4%	8.63	74.0%
10^6	Original - no chpt	714.95	-	N.A.	N.A.
	Original - chpt	804.66	12.6%	49.59	-
	Baseline C^3	868.18	21.4%	83.38	68.1%
	+ stack opts.	867.42	21.3%	83.38	68.1%

Table 5: Runtimes and Checkpoint Size, using variable optimizations

Because the overhead of checkpointing `treecode` is dominated by the cost of saving the heap, the total performance gain achieved by optimizing the lexical variables alone is minimal. These optimizations would have a significantly greater impact on codes that utilize large, statically allocated arrays and structures (e.g., some Fortran programs). In such codes, if checkpoints are placed inside common routines, the ability to exclude these elements in certain contexts would also have a significant impact.

5 Automatic Coloring

In Section 3, we showed that a color-based heap allocation can reduce the overhead of checkpointing heap objects. The performance results shown in row "+ stack opts." of Table 5 correspond to implicitly assigning all of these sites to a single default color. In this section, we show how the liveness analysis developed in Section 4 can be used to automatically assign multiple colors to allocation sites.

In the pseudocode shown in Figure 1, we have shown the allocation sites using the standard function `calloc()`. In the discussion below, we will refer to these sites by the variable that is assigned the result of `calloc()`, namely, `active`, `interact`, `btab`, `cell`, and `pvec`. After the colors are computed, the calls to `calloc` are modified so that the color is passed as an additional argument.

Very often, the application developer will write a wrapper to functions like `calloc` that checks for error conditions. Allocations are then made via this wrapper function. This is true of the original version of `treecode`, which defines a function `allocate` which contains the program's only call to `calloc`. In programs like this, a small collection of static allocating statements may be responsible for all or most of the memory allocation in a program. This severely limits the number of possible colorings. We handle these cases by recognizing when a function returns the output of a standard allocation routine. Our system then treats calls to these wrapper functions as the allocation sites.

5.1 Conservative Coloring

A simple coloring algorithm is based on the "liveness" of the output of each allocation sites. The output of an allocating function is said to be live at a checkpoint if it is in the transitive points-to set of one or more live variables, as determined by the analysis presented in Section 4, that may be dereferenced at point after the checkpoint. A color consists of the set of allocation sites whose output has the same liveness at any checkpoint.

For `treecode`, this partitions the allocations into two sets, {{`active`, `interact`}, {`btab`, `cell`, `pvec`}}. The performance relating to this coloring is shown in the rows labeled "+ conserv. coloring" in Table 8.

This coloring is not competitive because it causes the `cell`'s, which are all deallocated, to be saved, whereas the hand-written code does not. The problem arises because of the cycles present in the `treecode` data structures, as illustrated in Figure 2. It is impossible for our pointer analysis to determine that at the checkpoint all of the allocated cells were reclaimed.

5.2 Optimistic Coloring

The conservative coloring algorithm does not take into account the fact that the C^3 runtime system uses a live object count objects in order to determine whether each color needs to be saved. Thus, a coloring does not have to be "correct", in the sense that it accurately partitions the live and dead objects at each checkpoint; correctness is ensured by the runtime system. Therefore, our coloring algorithm should strive to produce a "good" coloring.

Our current approach to coloring is based on a set of intuitions about what constitutes a "good" coloring,

- To the greatest extent possible, objects that are known not to be live at a checkpoint should not share a color with objects that are known to be live.
- Since there is a minimal cost associated with saving a color, small, infrequently created, objects with similar liveness characteristics should share a color.
- Objects that may be freed before a checkpoint are more amenable to sharing a color than objects that are certainly not freed.
- The total number of colors cannot exceed the maximal number of colors provided by the system.

We have developed a heuristic that captures these intuitions. Our heuristic starts by assigning each allocation statement to its own color. Then a *rating* is assigned to each allocation statement at each checkpoint. This rating is a four-tuple of the following metrics,

1. **Live?: {Y,?,N}** An allocation is live if it unambiguously pointed to by a live variable that is dereferenced in the future. It is not live if it is not transitively pointed to by any live variable. Otherwise, its liveness is unknown.
2. **Size: {L,S}** An allocation is small if it returns space for a single structure or base-type object. Otherwise it is large.
3. **Frequency: {*,0,1}** An allocation has frequency 0 it has not occurred before the checkpoint, frequency 1 if it has occurred a non-looping number of times before the checkpoint and * otherwise.
4. **Free?: {Y,?,N}** An allocation has been freed if it is the unambiguous target of a free statement possibly occurring before the checkpoint. It is not if it is not in the points-to set of any free statement that possibly occurs before the checkpoint and unknown otherwise.

Each of these metrics forms a lattice, with the expected join operations. Ratings also forms a lattice, whose join operation is the pairwise join of each metric. The rating of a color is defined as the join of the ratings of all allocations assigned to that color.

Table 6 shows the ratings assigned to the allocation statements in `treecode`.

Allocation	Live?	Size	Freq	Free?
active	N	L	*	Y
interact	N	L	*	Y
btab	Y	L	1	?
cell	?	S	*	?
pvec	Y	L	1	N

Table 6: Ratings for `treecode`'s Allocations

Each pair of colors can be assigned a *compatability preference*, which indicates the desirability of merging the two colors and is based on the intuition presented above. Each preference is one of four values and reflects the desirability of merging colors with the those ratings at a particular checkpoint, *Strong Merge (SM)*, *Merge (M)*, *Seperate (S)* and *Strongly Separate (SS)*

The compatability preferences for `treecode` are shown in Table 7, For example, merging a color containing `active` with a color containing `pvec` is strongly undesirable according to our intuition since `active` is dead at the checkpoint whereas `pvec` is live and neither is a small, infrequently occurring allocation. This is reflected in the table by the pair having a rating of *SS*.

	{N,L,*,Y}	{Y,L,1,?}	{?,S,*,?}	{Y,L,1,N}
{N,L,*,Y}	SM	SS	S	SS
{Y,L,1,?}	SS	SM	S	M
{?,S,*,?}	S	S	SM	S
{Y,L,1,N}	SS	M	S	M

SM = Strong Merge, M = Merge, S = Separate, SS = Strongly Separate

Table 7: Color Compatibility Preferences

For a program with multiple checkpoints, the aggregate preference for a pair of colors is the average of the preferences at each checkpoint. Intuitively, it takes three preferences to counteract a strong preference of the opposite type.

Once the preferences have been computing, a pair of colors with the highest compatibility score is chosen and merged. The rating is assigned to the new color is the join of the previous ratings. The heuristic continues greedily choosing pairs to merge until there are no remaining desirable merges (ratings of *SM* or *M*) and the number of colors does not exceed the maximal allowable number of colors.

For `treecode`, the heuristic begins by merging the `pvec` and `btab` colors. The rating of the new entry is then {Y,L,1,?}. In the second iteration, the `active` and `interact` colors have the only remaining desirable merge preference and are merged. The new rating is {N,S,*,L}. The algorithm terminates at this point since there no remaining desirable merges to be made and the system allows for more than three colors. The result is a 3-coloring of the allocation statements: {{`active`, `interact`}, {`btab`, `pvec`}, {`cell`}}.

5.3 Experiments

Table 8 shows the aggregate performance results with new rows labelled "+ conserv. coloring" and "+ heuristic coloring". The 3-coloring computed by the heuristic has the very desirable property that the `bodytab` and `cell` allocations are in distinct colors. When a checkpoint occurs, the color containing the `cell` allocations will not be saved as no objects assigned to that color are live

As this example illustrates, the key to unlocking the performance of the colored heap allocation is carefully chosen colors. This requires an analysis that is more sophisticated than a simple liveness analysis. The problem is also more complicated than region analysis, because the decision to share a color is based not only on liveness but also on the size of the allocation, the pattern of invocations and reclamations, and the presence of multiple checkpoints.

Recall that the purpose of this study was to establish whether or not program analysis and transformation could be used to automatically derive ALC code that is competitive in performance with hand-written code. Comparing the results in Table 8, which are summarized in Figure 6, we can see that the answer is "yes", at least in the case of `treecode`.

Size	Configuration	Time		Chpt Size	
		Sec.	ovrhd	MB	ovrhd
10^4	Original - no chpt	3.61	-	N.A.	N.A.
	Original - chpt	4.70	30.2%	0.5	-
	Baseline C^3	6.14	70.2%	1.09	118.7%
	+ stack opts.	6.19	71.4%	1.08	118.5%
	+ conserv. coloring	5.71	58.2%	0.87	75.9%
	+ heuristic coloring	5.46	51.3%	0.69	38.8%
10^5	Original - no chpt	54.43	-	N.A.	N.A.
	Original - chpt	63.53	16.7%	4.97	-
	Baseline C^3	71.55	31.4%	8.64	74.1%
	+ stack opts.	70.97	30.4%	8.63	74.0%
	+ conserv. coloring	69.49	27.7%	8.34	68.1%
	+ heuristic coloring	64.10	17.8%	5.15	3.9%
10^6	Original - no chpt	714.95	-	N.A.	N.A.
	Original - chpt	804.66	12.6%	49.59	-
	Baseline C^3	868.18	21.4%	83.38	68.1%
	+ stack opts.	867.42	21.3%	83.38	68.1%
	+ conserv. coloring	867.07	21.2%	83.01	67.5%
	+ heuristic coloring	807.23	12.9 %	49.79	0.4%

Table 8: Runtimes and Checkpoint Size, using two and three colors

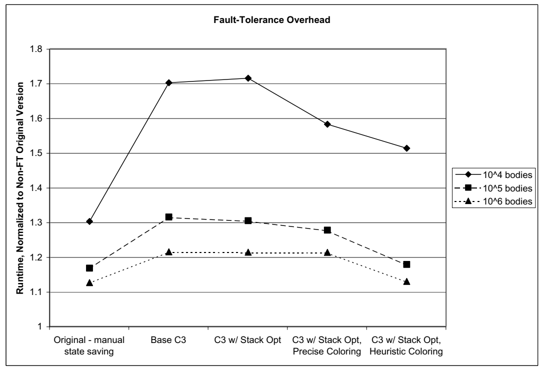

Figure 6: Overheads

6 Related Work

Manual Application-level Checkpointing Several systems have been developed to make ALC easier to program. The Dome (Distributed Object Migration Environment) system [4] is a C++ library based on data-parallel objects. SRS [17] allows the programmer to manually specify the data that needs to be saved as well as its distribution. On recovery the system uses this information to recover the program's state and redistribute the data on a potentially different number of processors.

Automatic Application-level Checkpointing Porch [15] supports portable ALC for programs written in a restricted subset of C. It generates run-time meta-information that provides size and alignment information for basic types and layout information, which allows the checkpointer to convert all data to a universal checkpoint format. The APrIL system [10] uses techniques similar to Porch, but uses heuristic techniques for determining the type of heap objects.

Reducing Checkpoint Size Beck and Plank [3] used a context-insensitive live variable analysis to reduce the amount of state information that must be saved when checkpointing. In this sense, their analysis is less precise than ours, however, their analysis is also able to compute information for *incremental* checkpointing.

The CATCH [12] system uses profiling to determine the likely size of the checkpoints at different points in the program. A learning algorithm is then used to choose the points at which checkpoints should be taken so that the size of the saved state is minimized while keeping the checkpoint interval optimal.

Automatic Memory Management There are many connections between our heap allocation techniques and other work on automatic memory management. First, our notion of heap "colors" is similar to "regions" in region-based allocation. However, there is an important difference: A color is a set of memory objects that are likely to have similar checkpoint requirements, while a region is a set of objects that can safely be deallocated all at once. Nevertheless, because both approaches are concerned with the lifetime of objects, there are similarities between our analysis and region analysis [8].

Second, there are connections with garbage collection [18]. For instance, both are inhibited by custom memory management and imprecise type information in C programs. Furthermore, with more precise type information, many garbage collection techniques (e.g., copying, generations), would be useful additions to our heap implementation.

7 Conclusions

A significant contribution of this work is that it demonstrates that it is possible for efficient application-level checkpointing code to be generated automatically. Other significant contributions include the following:

- our three-tiered approach to utilizing an inter-procedural program analysis that allows progressively more accurate information to be computed, as it is required,
- our novel design of a heap management system that facilitates efficient checkpointing of the heap, and
- our heuristic for automatically assigning colors to heap allocations.

In our current work, we are addressing the following,

Effectiveness and Efficiency. How effective and efficient is our system for other codes? We are in the process of collecting other applications for evaluation.

Automatic Checkpoint Placement. Our system currently requires the programmer to manually determine the program locations at which checkpoints will be taken. Can this be automated?

Saving vs. Recomputing. There are some cases where it is possible to avoid saving data by recomputing it on recovery. There are several examples among the variables in Figure 5: *btab* and *ncell* are copies of the global variables *bodytab* and *ncell* respectively. The real savings will come from recomputing heap data structures. This will be key to competing with hardwritten application-level checkpointing.

Reclaiming memory management. [5] demonstrates that custom memory management is usually less desirable than relying on the system-provided memory management. Furthermore, as we have seen in `treecode`, custom memory management can make it difficult to modify an application to use advances memory management features. A very interesting research problem would be to develop program analyses and transformations to replace custom memory management routines with calls to the standard routines. This would be useful for other memory management system, such as region-based allocation, garbage collection, etc.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] J. E. Barnes. Treecode guide. `http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.h` February 23 2001.

[3] M. Beck, J. S. Plank, and G. Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, Dept. of Computer Science, University of Tennessee, 1994.

[4] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, Nov. 2002.

[6] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *Principles and Practices of Parallel Programming*, San Diego, CA, June 2003.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[8] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *Static Analysis Symposium*, pages 463–482, June 2003.

[9] J. Ezick. Resolving constrained existential queries over context-sensitive analyses. Technical Report TR2003-1913, Cornell University Computing and Information Science, June 2003.

[10] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw. Process introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification. Technical Report CS-97-05, Department of Computer Science, University of Virginia, 25, 1997.

[11] IBM Research. Blue gene project overview. `http://www.research.ibm.com/bluegene/`, 2002.

[12] C.-C. J. Li and W. K. Fuchs. Catch – compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.

[13] J. B. M. Litzkow, T. Tannenbaum and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[14] National Nuclear Security Administration. ASCI home. `http://www.nnsa.doe.gov/asc/`, 2002.

[15] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[16] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[17] S. Vadhiyar and J. Dongarra. Srs - a framework for developing malleable and migratable parallel software. *Parallel Processing Letters*, 13(2):291–312, June 2003.

[18] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, Kinross, Scotland, UK, Sept. 1995.