Efficient Checkpointing of Multi-Threaded Applications as a Tool for Debugging, Performance Tuning, and Resiliency

Max Grossman
Department of Computer Science
Rice University
Houston, USA
max.grossman@rice.edu

Vivek Sarkar
Department of Computer Science
Rice University
Houston, USA
vsarkar@rice.edu

***Abstract*—Past work on application checkpointing systems has either focused on enabling application resiliency or as a tool for debugging (as in record-replay literature). Each of these use cases for checkpoints places different constraints on the constructed checkpointing system. When used for resiliency, checkpointing systems must minimize their interference with the running application. When used for record-replay and numerical debugging, checkpointing systems instead must focus on correlating the contents of a checkpoint to user-visible data structures in order to aid in the debugging process.**

Past literature has ignored the use of checkpoints in application performance tuning. While existing performance profiling tools enable the identification of hotspots in an application, creating full application checkpoints immediately prior to a hotspot enables rapid iteration on the performance of that hotspot.

In this paper, we present a checkpointing system for all three of these use cases: resiliency, application debugging, and application performance tuning. We present a novel checkpointing framework that creates efficient checkpoints of multi-threaded C programs using a hybrid compile-time and runtime approach. This approach reduces the framework's dependency on platform-specific features and improves its efficiency using insights from static and dynamic application analysis. Across a wide range of benchmarks we demonstrate that our framework incurs low overheads: $\sim$5% on average and in many cases less than 1%.

***Keywords*-checkpointing; resiliency; debuggers; performance tuning; record-replay;**

I. Motivation

Application checkpointing is a well-studied problem with a variety of use cases. An application checkpoint is a snapshot of program state that includes sufficient information to resume execution of an instance of that application from an intermediate point in time. Application checkpointing is most applicable to the following use cases in scientific computing:

1) Resiliency against software or hardware errors.
2) Debugging of application failures or numerical errors.
3) Performance profiling and tuning of application hotspots.

Resiliency is the classic motivating example for checkpointing. Creating periodic checkpoints allows a crashed process to be immediately resumed from its most recent checkpoint.

Checkpointing also enables debugging and performance tuning. Creating periodic checkpoints simplifies the process of reproducing a program error or analyzing a performance hotspot by allowing programmers to resume from a checkpoint immediately prior to the relevant code region. For long-running scientific applications, this can save days or weeks of time and enable more rapid iteration on an application. Checkpoints can be integrated into automated testing environments to protect against future regressions, or used as representative application inputs for a performance auto-tuning framework.

Modern debugging and performance profiling tools generally induce a significant amount of overhead. This can make program behaviors difficult to reproduce. Checkpointing techniques are uniquely suited to resolve these issues by allowing the application developer to compile and run their application with the highest optimization settings by default, but re-compile with different compiler flags or added instrumentation before resuming from a checkpoint. To make it feasible to run with checkpointing permanently enabled, checkpointing frameworks must keep overheads low and retain original application behavior.

The existing research in application checkpointing has primarily focused on resiliency, with some attention given to debugging using record-replay techniques. In this work, we support all three use cases using a compiler- and library-based approach to checkpointing. This type of approach has a number of merits relative to more low-level techniques proposed in past work including fewer platform dependencies, reduced overhead, and a more user-tunable checkpointing process.

For the remainder of this paper, we refer to this work as CHIMES (CHeckpointing of In-MEmory State). The main contributions of CHIMES include:

1) Automated checkpointing of stack, heap, and other user-level objects through source code inspection and transformation.
2) An efficient, overhead-aware runtime for multi-

threaded programs that handles program state tracking and checkpoint creation.
3) Support for user specification of checkpoints.
4) Support for pluggable user functionality in the creation and restoration of checkpoints.
5) A combined compiler and library approach to checkpointing which uses insights gained from the source code to enhance efficiency.

This paper is organized as follows. Section II will summarize related work in the area of application checkpointing. Section III will then describe our approach to checkpointing. Section IV experimentally evaluates the techniques described. Section V concludes by discussing the contributions of this paper and future directions for this work.

II. Related Work

Arguably the most well-known tool for checkpointing is Berkeley Lab's Linux Checkpoint/Restart tool (BLCR) [1]. In [1], the authors of BLCR present three design choices for every checkpointing system: user-level vs. kernel-level, the amount of user- and kernel-level state that can be checkpointed, and the level of integration with other components in the platform (e.g. MPI). BLCR uses kernel-level checkpointing to support pausing and resuming of MPI applications. BLCR is implemented as a Linux kernel module and supports checkpointing a wide range of user-level and kernel application state. BLCR uses custom callbacks to enable integration with other tools or libraries. BLCR takes a stop-the-world approach to checkpointing, pausing all threads until a checkpoint is fully persisted to disk.

DMTCP[2] takes a similarly low-level approach to checkpointing. Instead of adding a kernel module, DMTCP uses the `fork` and `abort` system calls to create a core dump of application state that can be restored from. To support checkpointing of additional program state not captured in this core dump (e.g. file descriptors), DMTCP wraps system calls as well and tracks their usage. Like BLCR, DMTCP dumps all application state with each checkpoint, leading to multi-second checkpoint times for applications with working sets in the megabytes[2].

IGOR[3] uses dirty page tracking to reduce the size of checkpoints by only checkpointing heap regions that have been modified. An application image is constructed from the pages written to disk to enable resume of the application. On resume, the user can select a point to resume from and IGOR will restore from the preceding checkpoint before using interpreted execution to move program state to the desired point in the program.

The work presented in [4] and [5] is the most similar to our work. The authors use a combined compile- and run-time approach to checkpoint OpenMP applications. At compile-time, the application code is analyzed to determine which arrays have been "dirtied" since the last checkpoint and need to be checkpointed. This information is propagated inter-procedurally at compile time. This analysis allows checkpointing of arrays to be started early, immediately after the last write to an array prior to a checkpoint being taken.

Power-Check [6] focuses on limiting the impact of checkpointing on power consumption. Assuming a checkpoint model that requires global synchronization and quiescing of application threads, they note that checkpointing periods are I/O intensive but not compute-intensive, offering the chance for power throttling techniques to be effective. They design an energy-aware I/O subsystem that can either sit under other checkpointing libraries (e.g. BLCR, DMTCP, CHIMES) or be used directly for application-specific checkpointing. In our work, we do not consider the opportunities for energy saving during checkpointing as our CHIMES system tries to overlap I/O intensive checkpointing with compute-intensive application execution, limiting the opportunities to throttle power without significantly degrading application performance.

Each of these checkpointing implementations has contributed to the state-of-the-art. However, each is limited in how it satisfies the motivation in Section I. All five previous works lack in transparency: checkpoints are mostly opaque containers whose contents are not easily mapped back to developer-visible constructs such as variables or functions.

BLCR, DMTCP, IGOR, and Power-Check pause all running threads in an application when creating a checkpoint and do not continue until the full checkpoint is flushed to disk, increasing overheads. The lack of source code analysis in these approaches exacerbates the efficiency problem even further. From BLCR [1], "large user applications often already do their own checkpointing for fault tolerance, and can often do it much more efficiently than an automated checkpoint system can, since they know exactly which parts of their application need to be saved and which can be discarded or regenerated". Only by analyzing application source code can we gain the insights necessary for efficient checkpointing.

The work in BLCR, DMTCP, IGOR, and Power-Check are also similar in that each is closely tied to the underlying platform. For example, BLCR is built as a Linux kernel module. While a kernel-level approach allows a checkpointing framework to manage state that user-level approaches do not have access to, this limits flexibility for future platforms and restricts the environments it can be deployed to. The compiler-based approach taken in [4] and in our work is more flexible and platform agnostic, tied only to the semantics of the programming model.

The work in [4] is also limited in several ways:
1) It cannot support multiple compilation units: the full call graph must be available at compile time.
2) It does not support resuming from the checkpoints it creates.
3) The target language is FORTRAN, which simplifies the problem of checkpointing by not considering

pointer aliasing.

In Section III we describe the techniques used in the design and implementation of our CHIMES checkpointing framework. We focus on how our work adds to the state-of-the-art and satisfies the requirements of the debugging, performing tuning, and resiliency use cases.

III. Methods

This section describes the main contributions of this work: a compiler- and library-based approach to checkpointing single-threaded and OpenMP programs. We start with an overview of what a CHIMES checkpoint contains and then cover step-by-step how the compile-time and run-time workflows 1) create a single checkpoint, and 2) resume from it.

A. Anatomy of a CHIMES Checkpoint

A CHIMES checkpoint includes the following application state:

1) Per-thread stack contents, including variable names, sizes, types, and values.
2) Global state, including variable names, sizes, types, and values.
3) Constant state, including variable names, sizes, types, and values.
4) Function addresses and function names.
5) Thread hierarchy information indicating which OpenMP threads spawned other OpenMP threads.
6) Heap state changed since the last checkpoint.
7) Metadata on aliased pointers in the host application.
8) Metadata on the pointer hierarchy in the host application (i.e. pointers that point to other pointers).
9) User-provided checkpoint data.

We assume that the heap of the host application accounts for the majority of its in-memory state. Checkpoints are stored on disk as binary files and all checkpoints are incremental in their storage of heap contents. To restore the full contents of an application's heap from a checkpoint, it may be necessary to also traverse backwards through a chain of predecessor checkpoints to find the current state of all bytes in the heap.

Checkpoints are created by programmer-inserted calls to `checkpoint()`, giving the programmer the ability to place checkpoints prior to important, buggy, or long-running code regions.

Note that the current checkpoint format does not include metadata on open files, signal handlers, or other system-managed state. Handling these types of system-specific state leads to less platform flexibility and higher overheads as more system interaction must be instrumented. These types of objects can be restored by custom user callbacks during checkpoint creation and resume.

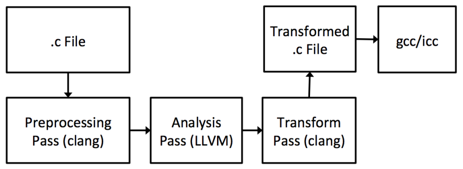

Figure 1. The CHIMES compilation workflow.

B. Compile-Time Analysis and Transformations

Figure 1 shows the high-level workflow of the CHIMES compilation pass. CHIMES processes one .c file at a time. The input file first goes through a preprocessing pass that performs some lightweight transformations to simplify the main transformation pass later, including hoisting expressions with side effects out of return statements or function call parameters.

1) Analysis Pass: Following the preprocessing pass, LLVM bitcode is generated from the preprocessed file and passed through an LLVM analysis pass. The analysis pass's primary purpose is to produce information on 1) intra-procedural pointer aliasing, and 2) the memory locations modified within each procedure.

During intra-procedural alias analysis, pointer variables within a function that may alias are marked as part of a single alias group. A globally unique alias group ID is generated for each alias group in each function. The analysis pass also tracks which alias groups are pointed to by other alias groups.

The analysis pass identifies alias group change locations, i.e., source code locations where a `STORE` to a member of an alias group occurs. This information is used at runtime to calculate heap state that may have changed since the last checkpoint was taken.

Alias groups that may be modified between two checkpoints are collected by propagating alias group change information from the original change location down control flow paths until it encounters 1) a `checkpoint` call, or 2) a redirection of control flow that may lead to a checkpoint being created. This generally leads to the aggregation of alias group change locations at checkpoint calls, at function calls, at conditional branches, and at return statements. These aggregate alias group change locations record all of the alias group IDs that may have been modified by `STORE` operations since the last aggregate alias group change location.

In addition to storing alias groups that have definitely changed, it is necessary to store alias groups that may be changed by a call to an externally defined function or function pointer. Any arguments passed by reference and all global values are conservatively marked as "possibly changed" and added to the next alias group change location as such. If at runtime CHIMES finds that the external call was instrumented by CHIMES, these "possibly changed" alias groups are removed from the change location. Otherwise, these alias groups and any alias groups indirectly reachable from them are conservatively added to the change location as definitely changed. This ensures that if an external library modifies any state tracked by CHIMES, the changes are included in the next checkpoint.

The analysis pass also generates metadata on:

1) Global, constant, and stack variables
2) Alias groups passed as parameters to function calls or returned by functions
3) Heap management locations, such as calls to `malloc`, `calloc`, `realloc`, and `free`
4) The call tree for this compilation unit, including any externally defined functions that are called but are currently unresolvable.
5) The OpenMP pragmas and clauses in the source code.

2) Transformation Pass: Once the analysis pass completes, the metadata generated by it is passed to the transformation pass, which is implemented as a standalone clang tool using LibTooling [7]. The CHIMES transformation pass performs a source-to-source transformation of the preprocessed source code. This transformation primarily inserts calls to CHIMES library functions that track application state identified by the analysis pass (e.g. stack variables, globals, function addresses).

Every compilation unit (i.e. input file) has a static, one-time module initialization function inserted which passes module information to the CHIMES runtime prior to entering the application's `main`:

```
static int module_init() {
  libchimes_init_module(...);
  return 0;
}
static const int __libchimes_module_init =
    module_init();
```

The transformation pass also instruments each function with a variety of CHIMES runtime callbacks that are used for tracking stack variables, heap allocations, inter-procedural alias creation, alias group change locations, entrance or exit from OpenMP parallel regions, or changes to the call stack.

As part of the transformation pass, jumps and labels must be inserted in any functions that may be on the stack when a checkpoint is created. The labels allow a resume of a checkpoint to skip to the original checkpoint location while reproducing the original call stack using only jump operations and function calls. A label is added to every callsite that may directly or transitively create a checkpoint, to each CHIMES callback that registers stack variables, and before each OpenMP parallel region that may have a checkpoint created inside.

By inserting jumps between these labels, we build a control flow tree within each function that allows the transformed application to jump from the entry point of a function, through stack variable registrations, and into any parallel regions or function calls necessary to reproduce the stack and thread state of the application when checkpointed. The root of the tree is the entrypoint of the function. The children at each layer of the tree are any parallel regions spawned from the current node of the tree or checkpoint-causing function calls made. This model supports resume from arbitrary call stacks and nested parallel regions, including recursive ones. The use of this control flow tree will be illustrated further in Section III-D1.

C. CHIMES Checkpointing Runtime

The transformations described in Section III-B2 add instrumentation to the host application. This instrumentation registers all checkpointable state with the CHIMES runtime. This section expands on the state stored by the CHIMES runtime and how that state is used to create checkpoints.

At a glance, the CHIMES runtime stores the following:

1) A mapping from the address of a heap allocation to its metadata.
2) A list of global and constant variables, along with associated metadata.
3) A mapping from function names to their addresses in the running application.
4) A mapping from each alias group to all other alias groups that have become aliased with it at runtime.
5) Points-to information for each alias group.
6) Per-thread stack trace information, stored as a stack of integer IDs.
7) A full call tree for the program, dynamically constructed from the per-compilation unit call tree information passed to `libchimes_init_module` as described in Section III-B2.

Precise and correct alias analysis is vital for CHIMES checkpointing: the mapping from aliases to heap allocations is used to determine what heap regions may have changed since the last checkpoint and need to be included in the next checkpoint. We perform inter-procedural alias analysis at runtime by passing the alias group information for function parameters and return values to CHIMES callbacks at the entry of each function, exit of each function, and before each callsite. For example, the alias groups of a formal parameter and an actual parameter will be merged following a function call. This analysis works across functions in different compilation units and through function pointer calls as long as both the source and target are transformed by CHIMES.

To illustrate the CHIMES runtime, we consider a simple example function in Listing 1 that creates a checkpoint. Pseudocode of the transformed code generated by the CHIMES transform pass for this function is shown in Listing 2.

Listing 1. A simple code example calling `checkpoint`.

```
int *sum_alloc(int *a, int b) {
  int sum = *a + b;
  *a = sum + b;
  int *alloc = (int *)malloc(sum *
      sizeof(int));
  checkpoint();
  return alloc;
}
```

Listing 2. An example of the transformed code generated from Listing 1.

```
int *sum_alloc(int *a, int b) {
  libchimes_enter_func("sum_alloc", sum_alloc,
      ...);
  if (____libchimes_resuming) goto lbl_0;

  lbl_0: int sum;
  libchimes_register_stack_var(&sum, ...);
  if (____libchimes_resuming) goto lbl_1;
  sum = *a + b;
  *a = sum + b;

  int *alloc;
  lbl_1: libchimes_register_stack_var(
      &alloc, ...);
  if (____libchimes_resuming) {
    switch (libchimes_next_call()) {
      case (0): goto lbl_2;
      default: abort();
    }
  }
  alloc = (int *)malloc(sum * sizeof(int));
  libchimes_register_heap(alloc, ...);

  libchimes_alias_groups_changed(...);
  lbl_2: checkpoint();

  libchimes_leaving_func(...);
  return alloc;
}
```

Upon entering `sum_alloc`, the CHIMES runtime is notified that a new entry should be pushed on the current thread's stack by the `libchimes_enter_func` callback. The information passed to `libchimes_enter_func` also assists with inter-procedural alias analysis for the parameters of `sum_alloc`.

Then, the transformed code checks to see if the current program execution is a resume from a checkpoint using `____libchimes_resuming`. We assume it is not for this example, Section III-D will provide more detail on how a checkpoint is resumed. Next, the `sum` and `alloc` stack variables are registered using `libchimes_register_stack_var`, and the heap memory allocated in `alloc` is registered with the runtime using `libchimes_register_heap`.

Immediately before creating a checkpoint, `libchimes_alias_groups_changed` is called to inform the runtime of which alias groups have been modified since the last checkpoint. This call would inform the CHIMES runtime that `sum` and `alloc` have both had their values set.

The `checkpoint` function has three main steps. First, it serializes all program state outside the heap into byte buffers, including stack variables, per-thread stack traces, global variables, and constants.

Second, the checkpoint function determines what parts of the heap need to be checkpointed based on 1) alias group change tracking, and 2) hashing of heap contents. The first stage is straightforward: CHIMES has been collecting a set of modified alias groups since the last checkpoint. Combined with a mapping from alias groups to heap allocations, CHIMES can construct a set of user heap allocations that may have changed since the last checkpoint.

In the second stage, CHIMES subdivides heap allocations into evenly sized chunks and computes a hash for the contents of each chunk. The chunk size is configurable, but defaults to 4MB. Hashing is done using the xxHash library [8]. Hashes are stored between checkpoints and only chunks whose hashes have changed since the last checkpoint are added to this checkpoint. Once `checkpoint` has determined exactly which regions of the heap need to be checkpointed, in-memory copies of each region are made.

Finally, the serialized byte buffers from the first step of `checkpoint` and the heap contents from the second step are passed to a dedicated checkpointing thread which writes them out to disk. The checkpointing thread uses asynchronous writes to keep the checkpointing thread off-core. If an out-of-memory error occurs while preparing data for checkpointing, the `checkpoint` function becomes blocking and writes heap state directly from the application buffers.

After the call to `checkpoint` returns in Listing 2 we call `libchimes_leaving_func` and return from `sum_alloc`. `libchimes_leaving_func` aids with inter-procedural alias analysis for return values and pops from the stack trace for this thread.

D. Resuming From a Checkpoint

The previous section covered checkpoint creation. In this section, we look at how a checkpoint can be used to resume program execution from the point-in-time that the checkpoint was created.

1) Restoring Program State: During initialization, the CHIMES runtime will detect check for a `CHIMES_CHECKPOINT_FILE` environment variable and, if found, load the serialized program state from it. Restoring program state from the serialized state is a three step process.

First, during runtime initialization at the start of program execution CHIMES reads the contents of the checkpoint file and stores the deserialized data. CHIMES uses the deserialized heap, constant, and globals data to construct a mapping from the addresses of objects in the address space of the original execution to their addresses in the current execution. This information is used to update pointers stored in the stack, heap, and globals. The pointer translation process uses a self-balancing binary tree to store the mapping from addresses in the checkpointed address space to their addresses in the current address space. Each node in this tree is an address in the checkpointed address space and the number of allocated bytes that follow it. A binary tree is used to keep lookups efficient.

The second step of the restore process is to restore the thread and stack state of the program using the labels and jumps discussed in Section III-B2. Listing 2 shows an example of the code generated to support this step. Upon entering a function with `____libchimes_resuming` set to true, control flow will jump to each stack variable registration, passing updated addresses for each of these variables to the CHIMES runtime. Once all stack variables have been traversed, `libchimes_next_call` is used to pop the next entry from the checkpointed stack for the current thread. The value popped determines which label to jump to next. This jump may target a function call or a nested parallel region. At the completion of this step, all threads will be inside a call to `checkpoint` with the same stack trace that the original program followed to create the checkpoint being restored, but with stale stack state. Note that this approach does not support restoring checkpoints taken from beneath function pointer calls, as there are no guarantees that the function pointer's value will be correct on resume. Future work could remove this restriction by special-casing the restore of function pointers.

This label-jump approach is the main reason for the CHIMES preprocessing stage. During the CHIMES preprocessing stage one of the transformations performed is to hoist any expressions with side effects out of function argument lists. If this step were not taken, jumping to a function call would cause its arguments to be evaluated with only partial program state restored.

The third step of the restore process happens from inside the final `checkpoint` call. First, the values of all stack variables are restored using the values deserialized from the checkpoint file. Then, all pointers in the stack, heap, and global variables are translated from the old address space to the new address space using the address information collected from the previous two steps. This step also finds all variables whose type is either a pointer-to-pointers or a pointer-to-structs and recursively performs the translation for all pointers reachable from each variable.

This pointer translation step is complicated by the flexibility of the C programming language. `void*` pointers may point to data structures that contain pointers which need to be translated. If these "hidden" data structures are unreachable from anywhere else, the obfuscation of a `void*` type prevents CHIMES from identifying all pointers in the program. We have implemented a feature (disabled by default) that brute force searches any heap allocations behind `void*` pointers for pointers that can be updated. While this feature has not caused unexpected behavior when enabled, it is possible it could mutate data which appears to be a pointer from the old address space but which is not.

After the address translation completes each thread returns from the `checkpoint` call and execution continues as usual with a fully restored program.

E. Pluggability

In CHIMES, we include a number of hooks to allow users to add custom checkpoint and restore functionality to their applications as needed.

Users can insert custom data in CHIMES checkpoints using `register_checkpoint_handler`:

```
void register_checkpoint_handler(
    void (*handler)(void *, void **,
        size_t *),
    void (*restore)(void *, heap_tree *,
        chimes_stack *),
    void *data);
```

During checkpoint creation, `handler` is called and passed the pointer `data` as its first argument. If handler wishes to add state to the checkpoint, it must set its second argument to be a valid buffer on the heap and set the third argument to be the length of this buffer.

On resume, `restore` is called and passed the address of the restored buffer, a data structure that can translate pointers in the old address space to the new address space, and a data structure that can look up stack variables by name and scope.

Users can also register custom handlers for restoring objects of a certain type:

```
void register_custom_init_handler(
    const char *type_name,
    void (*handler)(void *));
```

If CHIMES finds an object whose type matches `type_name`, it will pass the address of this object to `register_custom_init_handler`. This is useful for restoring objects specific to third-party libraries (e.g. CUDA, pthreads).

F. Optimizations

Section I pointed out that for a checkpointing system to be feasible it must be efficient, adding little overhead. A naive implementation of the techniques described in this section would lead to significant overheads for many applications: taking the address of stack variables and functions impedes compiler optimization, excessive function calls and runtime logic adds overhead, and frequent checkpointing would considerably add to execution time. In this section we describe techniques used to limit the overhead incurred by the CHIMES runtime, and evaluate their effectiveness in Section IV.

One of the most effective optimizations implemented is the CHIMES ShortCut Mode (SCM). For SCM, a duplicate version of each function is emitted with most of the

CHIMES instrumentation removed. The only instrumentation kept is heap registration callbacks, which are necessary to associate each allocated buffer with an alias group.

Calling the SCM version of a function reduces overhead, but has some constraints. The called function and all of its callees must be known functions which definitely will not checkpoint. This can be determined using the global call tree constructed at runtime. If the SCM version of a function is called then all changes to and aliasing of alias groups must be evaluated ahead-of-time based on statically known information. This can reduce the accuracy of this information, but not the correctness.

The CHIMES runtime is also aware of the overhead it adds to the host application and limits checkpoint creation to keep overhead below a certain threshold, when possible. Expensive CHIMES runtime callbacks are instrumented to measure the time spent inside. The total time in the CHIMES runtime is tracked and compared to the overall wallclock time of the application. If this ratio exceeds a threshold, checkpoints are not created. This threshold was set to 5% in our experiments. This system includes a maximum allowable period between checkpoints and forces checkpoint creation if that period is exceeded, even if the estimated overhead is greater than the allowable threshold. In our experiments, we set this period to be 60 seconds.

During experimentation, we also found that taking the address of functions and stack variables can significantly degrade the ability of the compiler to optimize application code. We addressed this problem by using POSIX `dlsym` to fetch function addresses, and used liveness analysis to reduce the set of stack variables that had to be registered.

IV. Experimental Evaluation

We evaluate CHIMES performance based on three metrics: overhead added, checkpoint size relative to the size of the application in memory, and number of checkpoints created. We use benchmarks from the Rodinia benchmark suite [9], benchmarks from the SPEC benchmark suite [10], the Lulesh [11] application, the CoMD [12] application, the UTS [13] application, and a custom 3D stencil benchmark called Iso3D that is representative of wavefront propagation simulations from the energy industry.

All benchmarks and metrics are evaluated on two hardware platforms. Platform A contains a 12-core 2.80GHz Intel X5660 CPU, 48GB of system RAM, and is connected to a GPFS storage system by QDR Infiniband. Platform B is an IBM Power 755 node containing 4 eight-core 3.86GHz POWER7 CPUs with 4-way simultaneous multithreading (128 hardware threads in total), 256GB of system RAM, and is also connected to a GPFS storage system through QDR Infiniband. The GNU C Compiler was used on both platforms, v4.8.5 on Platform A and v4.4.7 on Platform B.

We compare performance of both single-threaded and OpenMP multi-threaded programs. Single-threaded tests are denoted with the label "CPP". Multi-threaded tests are denoted with the label "OMP".

Benchmark	**CPP**		**OMP**	
	Time	**Space**	**Time**	**Space**
Iso3D	147.86s	3.21GB	36.38s	3.22GB
Lulesh	32.63s	2.76MB	167.60s	80.32MB
CoMD	101.23s	308.32MB	101.63	314.66MB
UTS	64.07s	15.26MB	5.84s	183.11MB
RodBackprop	103.00s	19.97GB	44.35s	19.97GB
RodBfs	124.11s	2.50GB	121.71s	2.50GB
RodB+tree	10.96s	73.97MB	2.00s	73.97MB
RodHeartwall	112.24s	28.73MB	11.70s	28.73MB
RodHotspot	54.52s	384.00MB	18.65s	384.00MB
RodKmeans	101.46s	132.11MB	16.92s	132.11MB
RodLavamd	122.70s	20.56MB	11.68s	20.56MB
RodLud	8030.28s	16.00MB	8207.23s	16.00MB
RodMyocyte	227.46s	286.91MB	18.81s	286.91MB
RodNn	184.50s	15.32MB	29.06s	15.32MB
RodNw	49.62s	19.20GB	27.99s	19.20GB
RodParticlefilter	9.62s	200.33MB	9.29s	3.07GB
RodSrad	91.47s	85.14MB	12.25s	85.14MB
SPECBotsAlgn	761.60s	1.35MB	63.82s	1.35MB
SPECBotsSpar	791.65s	757.83MB	737.69s	72.31MB
SPECSmithwa	97.51s	0.27MB	0.38s	11.33MB
SPECKDTree	0.22s	40.06MB	40.10s	6.96MB

Table I
Median execution time and peak memory consumption for the baseline version of each application on Platform A.

All tests are repeated 10 times and the median result is used to build the graphs below. Table I lists the execution time and memory consumed for each application running on Platform A without CHIMES.

A. Overheads

To evaluate the overhead of CHIMES, we start by running a transformed version of the application linked with an empty runtime library (referred to as Empty tests). This evaluates the overhead added by only the inserted function calls and other source code instrumentation. Then, we use a CHIMES library that implements all of the functionality from Section III-C but does not actually create checkpoints (referred to as No-Checkpoint tests). This measures the overhead added by tracking the state of the application. Finally, we test with the full CHIMES runtime library and measure any increase in overhead caused by creating checkpoints on disk (referred to as Checkpoint tests). All overheads are measured relative to the original application, compiled with gcc -O3.

Note that in some cases the Empty tests may demonstrate higher overhead than the others because No-Checkpoint and Checkpoint tests are often able to enter SCM mode in cases where Empty tests do not.

Figures 2 and 3 show the results of running the single-threaded tests on both hardware platforms. In general, we see an expected trend of increasing overhead from the Empty tests to the No-Checkpoint tests to the full Checkpoint tests. The median overhead for the Checkpoint tests across all

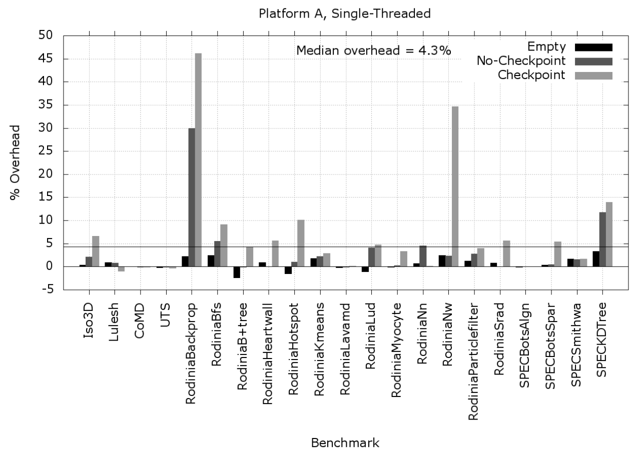

Figure 2. Overheads on Platform A during single-threaded tests.

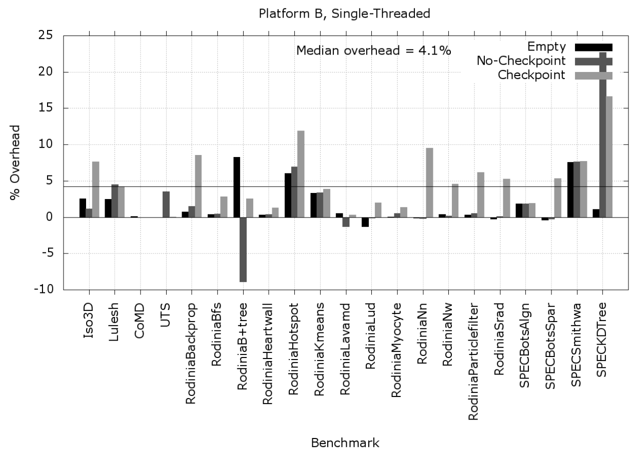

Figure 3. Overheads on Platform B during single-threaded tests.

applications on Platform A is 4.3%, and on Platform B is 4.1%.

Figure 2 shows a significant slowdown for RodiniaBackprop caused by the CHIMES code transformations interfering with compiler optimizations when stack variable and function addresses are taken, as discussed in Section III-F.

In Figure 2, the RodiniaNw results show significant overhead with checkpointing enabled, and in both Figures 2 and 3 we see similar behavior for SPECKDTree. We find that the added execution time comes from a wait at execution termination for the last and only checkpoint to complete being written to disk. These benchmarks are not characteristic of the long-running iterative scientific applications targeted by this and other checkpointing work. They are short-lived applications for which checkpointing offers little value.

In Figure 3, the RodiniaB+tree results show negative overheads when running the No-Checkpoint test. This result is caused by cache behavior. RodiniaB+tree performs many small heap allocations. In CHIMES, a small header (8 bytes) is added to each of these allocations, improving the cache characteristics of RodiniaB+tree by pushing more allocations onto separate cache lines. If these allocation headers are removed, No-Checkpoint overhead becomes 1.5%. Note that Platform A and B both have the same L1 cache line size (64 bytes), but that Platform B has a longer L2 cache line (128 bytes vs. 64 bytes for Platform A). This explains why Platform A does not demonstrate this behavior for RodiniaB+tree: its smaller L2 cache lines cause similar caching behavior with and without the added header.

In the OpenMP results, most of the outliers mimic the results from the single-threaded programs. The main difference is the CoMD Empty test on Platform B, where $\sim$50% overhead is recorded. This is caused by the lack of SCM mode in the Empty tests. Without SCM mode, some tight parallel loops are run with instrumentation enabled, which

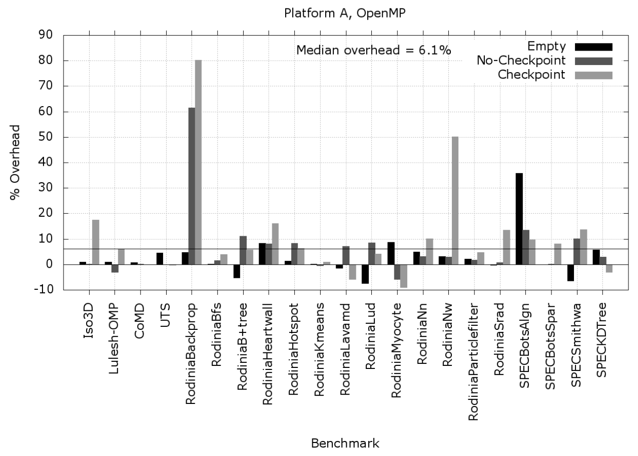

Figure 4. Overheads on Platform A during multi-threaded OpenMP tests.

includes one inserted synchronization point. This synchronization adds significant overhead on Platform B because it is more parallelism than Platform A (128 hardware threads on Platform B vs. 12 on Platform A).

Otherwise, the OpenMP tests perform similarly to the single-threaded tests, with an average overhead of 6.1% on Platform A and 5.3% on Platform B.

1) Number of Checkpoints: When evaluating the overhead of CHIMES, it is important to also consider how many checkpoints are being created. Figure 6 shows the number of checkpoints created by each application on Platforms A and B.

Some benchmarks execute for an insufficient amount of time to create more than one checkpoint, though many produce on the order of tens or hundreds of benchmarks. Note that OpenMP applications tend to produce fewer checkpoints than single-threaded applications as instrumentation

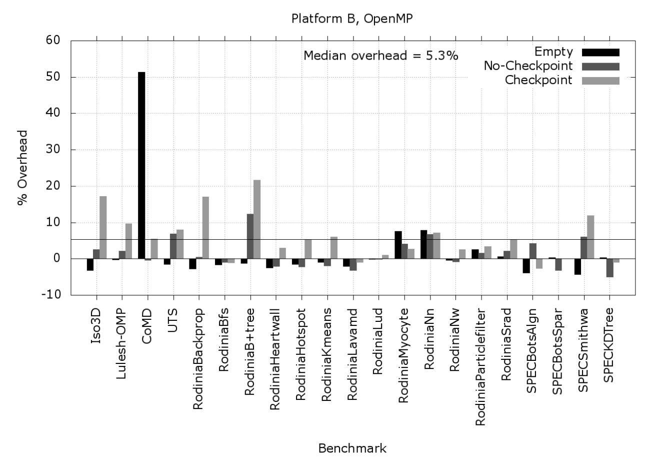

Figure 5. Overheads on Platform B during multi-threaded OpenMP tests.

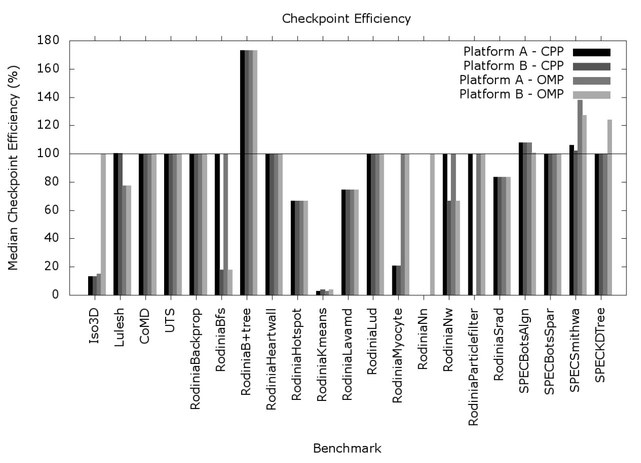

Figure 7. Median checkpoint efficiency on Platforms A and B across all checkpoints created by test runs of all applications.

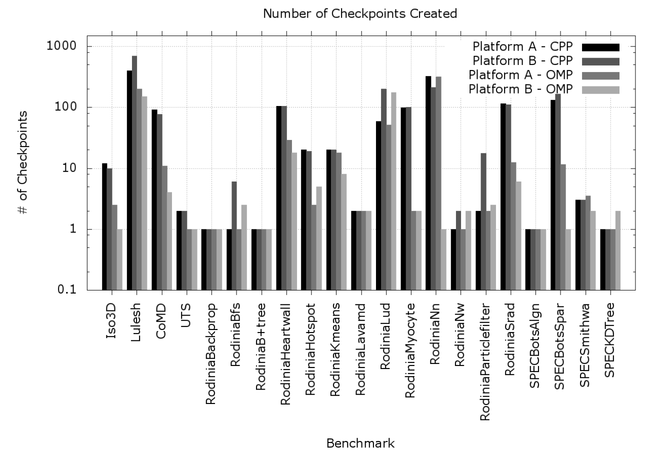

Figure 6. Median number of checkpoints created on Platforms A and B for each benchmark.

is added to track thread state, thereby increasing overheads and leading to more checkpoint throttling.

B. Checkpoint Efficiency

Checkpoint efficiency is a measure of the size of the checkpoints created for an application, relative to its total size in memory. Figure 7 shows the checkpoint effienciesfor all benchmarks on Platforms A and B. 100% efficiency indicates that the size of the application's in-memory state and the size of the checkpoints are the same. For some applications, we see that the change set tracking and hashing described in Sections III-B1 and III-C successfully reduced the amount of application state that had to be checkpointed. However, it is quite common for applications to regularly touch all application state (e.g. on every time step), so in many cases a checkpoint is a full copy of the running application. In some cases where the application working set is small, the checkpoint is appreciably larger due to CHIMES-specific objects added to the checkpoint. For instance, CHIMES includes alias set information in the checkpoint, which is not considered a part of the running application's working set.

C. Comparison to Related Work

Section IV-A showed that the CHIMES framework adds on average $\sim$5% of overhead to our benchmarks. In this section, we compare this level of overhead to the most related frameworks from Section II: BLCR, DMTCP, and the incremental checkpointing system described in [4] and [5].

It is difficult to find published performance numbers for BLCR. [1] reports multi-second pause times for applications with working sets as small as 50 MB, while for CHIMES we generally see application pause times on the order of milliseconds thanks to the offload of checkpoints to a dedicated thread.

DMTCP [2] does not report the overheads introduced by their system, but do report an example checkpoint time of 25.2 seconds for an application with a 680 MB working set, indicating a checkpointing rate of 26.98 MB/s. Considering our similarly sized SPECBotsSpar benchmark, we observed a worst case overhead on Platform A of 8.05% to produce an average of 11.5 checkpoint files for a 791.65 second execution when the working set size was 757.83 MB. This translates to a checkpointing rate of 136.75 MB/s, more than five times that of DMTCP.

In [4] the authors observe that the checkpointing runtime itself adds little overhead, even with a dedicated checkpointing thread. This agrees with our results for the No-Checkpoint tests in Section IV-A. In [5], the authors demonstrate similar checkpointing overheads to this work, on the order of 5-10%. This is to be expected due to the similarity in approaches.

V. Conclusions

There are many tradeoffs in the design of a checkpointing framework: how comprehensive the checkpointable state is, how much attention is given to efficiency, how much control the user is given over checkpoint creation, the layer in the software stack to implement the framework, etc. In this work we present a novel checkpointing framework that has the following characteristics:

1) Automated checkpointing of stack, heap, and other user-level objects through source code inspection and transformation.
2) A highly efficient, overhead-aware runtime for multi-threaded programs that handles program state tracking and checkpoint creation.
3) Support for user specification of checkpoints.
4) Support for pluggable user functionality in the creation and restoration of checkpoints.
5) A combined compiler and library approach to checkpointing which uses insights gained from the source code to enhance efficiency.

There are many possible future directions for this work. Integration with OpenMPI's custom checkpointer framework [14] or beneath multi-level checkpointing libraries, such as SCR [15], would enable support for distributed checkpointing. Preliminary investigations show that the techniques presented in this paper also work well for heterogenous systems that include GPU or MIC accelerators.

While this work focuses on checkpointing efficiency, future work should also focus on the efficiency of resuming. Preliminary investigation shows that the main bottleneck in resuming CHIMES is the number of incremental checkpoints that must be traversed. This suggests an interesting runtime tradeoff between creating incremental checkpoints to reduce checkpointing overheads, versus full checkpoints to reduce resume latency.

The evaluation in Section IV shows that this framework is not only flexible enough to handle checkpointing of real-world scientific applications, but that it does so efficiently and transparently to the user. CHIMES supports real-world, long-running, scientific applications that have large memory footprints, use function pointers, use complex types, and use complex build systems. Not only does CHIMES support them, but CHIMES makes it easier to build and improve them by easing debugging, performance hotspot analysis, and resilient application development.

Acknowledgment

This work was supported in part by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097 and Rice University. This work was also supported in part by NIH award NCRR S10RR02950, an IBM Shared University Research (SUR) Award in partnership with CISCO, Qlogic and Adaptive Computing, and Rice University.

References

[1] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," *Lawrence Berkeley National Laboratory*, 2005.

[2] J. Ansel, K. Aryay, and G. Coopermany, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[3] S. I. Feldman and C. B. Brown, "Igor: A system for program debugging via reversible execution," in *ACM SIGPLAN Notices*, vol. 24, no. 1. ACM, 1988, pp. 112–123.

[4] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[5] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental evaluation of application-level checkpointing for openmp programs," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 2–13.

[6] R. R. Chandrasekar, A. Venkatesh, K. Hamidouche, and D. K. Panda, "Power-check: An energy-efficient checkpointing framework for hpc clusters," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 261–270.

[7] "Clang libtooling," http://clang.llvm.org/docs/LibTooling.html.

[8] "xxhash," https://github.com/Cyan4973/xxHash.

[9] S. C. et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[10] K. M. Dixit, "The spec benchmarks," *Parallel computing*, vol. 17, no. 10, pp. 1195–1209, 1991.

[11] I. K. et al., "Lulesh programming model and performance ports overview," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-608824, December 2012.

[12] "Comd," http://www.exmatex.org/comd.html.

[13] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*. Springer, 2007, pp. 235–250.

[14] "Openmpi self-checkpointer," http://www.crest.iu.edu/research/ft/ompi-cr/.

[15] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–11.