SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing

Peter K. Szwed, Daniel Marques†, Robert M. Buels, Sally A. McKee, Martin Schulz
School of Electrical and Computer Engineering †Department of Computer Science
Cornell University
Ithaca, NY 14853
{pkszwed, rbuels, sam, schulz}@csl.cornell.edu
†marques@cs.cornell.edu
http://simsnap.csl.cornell.edu

Abstract

As systems become more complex, conducting cycle-accurate simulation experiments becomes more time consuming. Most approaches to accelerating simulations attempt to choose simulation points such that the performance of the program portions modeled in detail are representative of whole-program behavior. To maintain or build the correct architectural state, "fast-forwarding" models a series of instructions before a desired simulation point. This fast-forwarding is usually performed by functional simulation: modeling the effects of instructions without all the details of pipeline stages and individual μ-ops. We present another fast-forwarding technique, SimSnap*, that leverages native execution and application-level checkpointing. We demonstrate the viability of our approach by moving checkpointed versions of SPLASH-2 benchmarks between an Alpha 21264 system and SimpleScalar Version 4.0 Alpha-Sim. Reduction in experiment times is dramatic, with minimal perturbation of benchmark programs.*

1 Introduction

Architectural simulation is the main tool computer architects use to explore new designs. Simulators vary in the details they model, and different kinds are suited for different types of evaluation: gate-level for low-level functional evaluation and timing analysis; trace-driven when relative timing of events is unimportant to the experiment; and cycle-accurate, execution-driven simulation for evaluating most microarchitectural components. The more detail modeled, the slower the simulation. As architectures and systems become more complex, conducting a single, cycle-accurate simulation experiment can take from days to weeks.

To reduce the time per experiment, architects may use reduced workloads [13], or may choose to run detailed simulation only on pieces of an application. The portions modeled may come from the beginning, or may be chosen (e.g., by random sampling) further into the application's execution. In the latter case, some *fast-forwarding* mechanism is used to model the program's execution up to the portion(s) of interest. Functional simulation is one means of implementing fast-forwarding [5, 11, 12, 22]. The parts of the application chosen to be modeled in detail (simulation points) affect the statistics generated by the experiment: ideally, one would choose those portions having the most influence on overall program behavior [15, 22, 23]. For instance, Sherwood et al. demonstrate that many popular benchmark programs exhibit periodic behavior in terms of dynamically executed basic blocks [23]. Choosing instruction stream fragments with a dynamic basic block profile similar to that for full execution can yield detailed simulation statistics representative of those from full execution.

Performing detailed, cycle-accurate simulation on only representative points of the instruction stream dramatically reduces the time to conduct a simulation experiment. Unfortunately, functionally simulating program behavior up to the simulation point at which to begin cycle-accurate modeling can still take a significant amount of time. In fact, execution times of such hybrid simulations are in many cases dominated by the functional simulation time, which can be up to several days for SPEC 2000 codes [6, 23].

To further reduce experiment time, we substitute functional simulation with *native*, real-time execution, using checkpointing to transfer application state to a simulator at a desired simulation point. Our technique — SimSnap — fully encapsulates any checkpointing and restart functionality within the application and hence allows any application to restart itself within any simulator using the same ISA and data-type sizes. Due to this encapsulation, also known as Application-Level Checkpointing (ALC), SimSnap requires no external support, especially *no modification of the simulator*.

benchmark	cycle-accurate simulation	functional simulation
barnes	8960	209
fft	11975	195
lu-cont.	37682	348
ocean-cont.	10433	132
radix	4083	78
water-n^2	19615	252
Average	9177	202

Table 1. Simulated vs. Native Run-time Factors

To quantify the potential of this approach, Table 1 shows slowdown for both cycle-accurate and functional simulation compared to native execution for codes from the Splash2 [24] suite (Table 2 gives parameters used for these runs). On average, functional simulation is 45 times faster than cycle-accurate simulation, but is still 202 times slower than native execution.

SimSnap is based on the the *Cornell Checkpoint Compiler* (C^3) [2, 3]. C^3 transparently transforms a given application into a checkpoint-enabled code by introducing the necessary functionality to track, save, and restore its internal state. In this paper, we use the methods of the C^3 system to transform a set of scientific benchmarks. Using a checkpoint generated by the *native* execution of the code, we are able to continue with *simulated* execution from where the checkpoint was taken. Likewise, we can resume *native* execution of the code using a checkpoint generated during the *simulated* execution. Comparing simulation statistics and dynamic basic block profiles for the original and instrumented applications shows that SimSnap fast-forwarding minimally perturbs the target application.

This paper lays the groundwork for our system, provides a demonstration of the feasibility of the approach, and serves as a snapshot of current progress. We are encouraged by initial results: our methods show promise for efficiently migrating application execution between a native environment and a cycle-accurate simulation engine. The benefits of this research direction are manifold: using native execution to bring a program to completion after modeling a simulation point provides a small measure of validation of the simulation model; using detailed simulation is useful in debugging new checkpointing protocols; and the process of merging checkpointing and simulation-point technologies has inspired optimizations in both areas.

2 Related Work

2.1 Related Simulation Techniques

In Section 1 we discuss some of the problems with modern simulation experiments; here we briefly survey a few recent approaches to those problems. Space limitations preclude us from discussing more than just the solutions that are complementary to or could be viable alternatives to the approach we propose with SimSnap.

The MinneSPEC input set for the SPEC CPU 2000 benchmark suite provides reduced workloads that allows computer architects to conduct simulation experiments relatively quickly using existing simulators. Although derived from the standard SPEC CPU 2000 workloads, the SPEC CPU benchmarks using MinneSPEC should be considered a separate benchmark suite, since the benchmark programs exhibit different behaviors from their executions using the official inputs, MinneSPEC constitutes a valuable tool for exploring a large parameter space efficiently, allowing architects to choose which configurations to simulate in detail with the full workloads [14].

Another approach samples portions of benchmark execution, performing *warmup* functional simulations of a certain number of instructions (called pre-cluster instructions) before detailed simulation begins [5, 11]. Doing so attempts to create the correct cache and branch predictor states (i.e., the states that would exist if full execution were simulated) for the portions of the benchmark being simulated in detail. Haskins and Skadron exploit *Memory Reference Reuse Latencies* (MRRLs) to choose the number of pre-cluster, warm-up instructions to simulate functionally before a desired simulation point [12]. An MRRL represents the number of instructions that elapse between successive references to a given address. This method of selecting warm-up periods about halves simulation running times with minimal effect on IPC accuracies.

Generating accurate statistics requires that simulation points be chosen carefully: Sherwood et al. find that simulating the first million instructions yields an average error of 85% for SPECint 2000, and fast-forwarding one billion and then simulating 100 million yields an average error of 51% [23]. SimPoint uses *Basic Block Distribution Analysis* combined with machine learning approaches to clustering analysis to concisely summarize the behavior of an arbitrary section of execution in a program. This information can then be used to select representative samples to be simulated in detail, greatly reducing simulation time without sacrificing statistical accuracy.

Our research agenda includes using the SimSnap framework to model SPEC 2000 benchmarks using SimPoint techniques and native execution to fast-forward between multiple simulation points. We expect our code additions for recording state and taking checkpoints will have minimal effect on the simulation points for these benchmarks, but we have yet to verify this.

2.2 Checkpointing

Checkpointing refers to saving program state, usually to stable storage, so that it may be reconstructed later. Checkpointing provides the backbone for rollback recovery (fault-tolerance), playback debugging, and process migration. Early checkpointing efforts provide building blocks for protocols that enforce correct semantics in distributed simulation environments (as in Time Warp [10]) or that determine the global state of a distributed system [4]. Here we focus on checkpointing techniques most closely related to the approach we use in SimSnap.

System-Level Checkpointing (SLC) encodes the state of a process by capturing the contents of its address space, plus the process-specific data structures in the operating system. SLC is most often provided by the operating system, or by a library linked to the application. It is therefore transparent to the application programmer, but also not able to take advantage of application-specific optimization opportunities. SLC has been used to facilitate process migration for load-balancing in parallel systems [16, 21]. Plank et al. develop efficient, portable checkpointing in Unix [18]. SLC is frequently easy to use compared to other checkpointing techniques — it usually requires nothing extra from the programmer — but it may save significantly more state than that necessary to restart the process correctly. Furthermore, SLC process descriptions are specific to a given system. Plank et al. increase the efficiency of their approach by developing *memory exclusion* techniques to reduce the amount of state saved [19]. This SLC approach is then expanded by the addition of a user-directed checkpointing infrastructure, yielding a hybrid approach that exploits data flow equations allowing the compiler to generate correct memory exclusion calls for both clean and dead variables [17].

Application-Level Checkpointing (ALC) techniques are integrated within the application, enabling it to checkpoint and restart itself without external system support and to optimize the checkpointing process using application specific information. ALC is mostly applied manually by the application programmer during program development. Applying it transparently is still an area of active research and may be implemented by a compiler, preprocessor, run-time library, or some combination of these. Nonetheless, ALC strives to provide the portability that SLC cannot, and with portability comes greater process migration capabilities and fault tolerance. For instance, Ferrari et al. look at a heterogeneous checkpoint/restart mechanism based on automatic code modification [9]. They introduce the notion of *process introspection*, or modifying the program to capture its own internal, dynamic state in a form such that the program can be restarted (recovered) on a different architecture. Consisting of a set of semi-automated tools built around a flexible, abstract design pattern for constructing checkpointable programs, their system constitutes the first portable, extendable, platform-independent checkpointing mechanism.

Ferrari et al.'s system is targeted at large-scale, high-performance, heterogeneous distributed environments; other approaches target environments more limited in size, architectural variety, or application scope. For instance, Dome (Distributed Object Migration Environment) supports checkpointing and restart for Networks of Workstations via a C++ library of data parallel objects [1]. ALC provides the necessary support for heterogeneity: the checkpoint and restart mechanisms are placed within Dome's C++ objects, which allow processes to be restarted on different numbers of machines and different architectures in a manner that is semi-transparent to the programmer. The Porch system supports portable ALC for programs written in a subset of C [20].

All these ALC approaches must insert code to save the function call sequence along with run-time meta-information on data sizes, alignment, and layout. The meta-information allows the checkpointer to convert all data to a universal checkpoint format. Special routines adjust stack and heap pointers appropriately, allowing the data layout from the checkpointed system to be "translated" to the appropriate layout for the restart system. In many cases the ALC infrastructure is added manually, which can be a tedious and error-prone process. Instead, we rely on an automatic preprocessor, the C^3 system, which transparently instruments C source code to checkpoint itself. The C^3 instrumentation inherently alters the given application, which can lead to slight perturbations in execution. We discuss our tools and the consequences of using them in the following sections.

3 SimSnap Approach

Figure 1 illustrates the differences among common approaches to simulation experiments: *Full Simulation* (Full); *Fast Forwarding* to regions of interest (FF); switching between Fast Forwarding and Full Simulation to avoid multiple forwarding passes, each starting from the beginning of the benchmark, but continuing to different simulation points (Switch); and native execution with checkpointing using SimSnap (CP). The left side of the figure conveys the relationships among the number of benchmark instructions modeled (on this side, the top of the figure corresponds to the beginning of the benchmark), and the right side conveys the corresponding relative relationships with respect to wall-clock time spent using each method.

With SimSnap, the application saves and restores its own state without external intervention. The restore can therefore be done within *any* binary-compatible simulator by simply running the restore routine within the application.

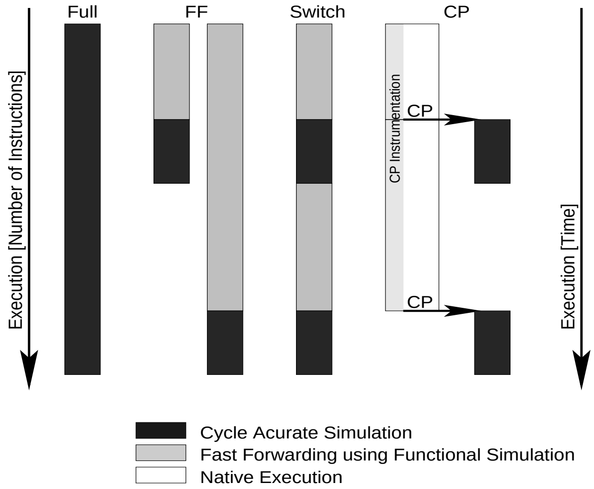

Figure 1. Instructions Executed and Time Spent in Simulation

In other words, the restore process is simulated along with the rest of the application *without the simulator being aware of it*.

Using SimSnap requires several steps:

1. **Instrument application for ALC.** The C^3 preprocessor adds code to track program location and state, as well as routines to store and retrieve a checkpoint.

2. **Choose code region(s) of interest.** The means to accomplish this depend on the concrete simulation method. Examples include checkpointing after a fixed number of instructions; checkpointing at desired simulation points, as in the SimPoint approach [23]; or checkpointing after reaching a user-specified location in the code.

3. **Execute natively to beginning of a region.** The application runs on a host compatible with the simulated machine.

4. **Create checkpoint.** When the application reaches a location specified in the second step, it activates the checkpoint routines inserted by the ALC preprocessor, and dumps its state to disk.

5. **Restore application within simulator.** The application begins executing within the simulator in restore mode, i.e., the application loads the checkpoint and restores its state.

6. **Perform cycle-accurate simulation during region of interest.** After the restore completes, the simulator models the chosen region.

7. **Repeat three to six, as required.** If more than one region of interest is to be modeled, the application execution continues to take the remainder of the required checkpoints. In our current framework, these are loaded into separate instances of the simulator.

An additional advantage of SimSnap is that it is possible to store the checkpoints on disk for later use. It is therefore not necessary to rerun the fast forwarding process for each experiment, as long as only the simulation environment changes and not the application (which is the most common case in architectural research).

Since the same binary is used for native execution and simulation, it is possible to checkpoint and restore in both environments. This can be used for fast simulation validation: after completing cycle-accurate simulation, the application checkpoints its state, and completes with native execution. Results can then compared with those of a pure, native execution, verifying the functional correctness the simulated portions in the SimSnap approach.

4 ALC Instrumentation via C^3

The C^3 system 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 equivalent version that can save and restore its own state; and a library (the C^3 runtime) that contains checkpointable implementations of several standard C library functions, plus utility functions used by the inserted code. The output of the pre-compiler is passed to the native compiler, where it is compiled and linked with the runtime, producing an ALC-enabled application.

4.1 Checkpoint Insertion

The C^3 system requires no modifications to the input program, other than marking checkpoint locations with a pragma statement. There are two types of checkpoint locations, each marked with a different pragma statement. Those marked (`#pragma ccc PotentialCheckpoint`) are converted by the pre-compiler into code to take a checkpoint if the runtime determines that one should be taken. This could be used to take a checkpoint after a specified amount of time has passed, after a specified number of instructions have been executed, or if the runtime has received a signal from another process. Locations marked (`#pragma ccc ExplicitCheckpoint(expression)`) are converted by the pre-compiler into code to take a checkpoint whenever the specified expression holds true. For example, the expression could be used to allow for a checkpoint to be taken upon entering a critical section of the code. The modified application can only checkpoint and restart at the specified locations, so static analysis can be used to reason about the behavior of the application at those points. The pre-compiler uses the results of such analysis to optimize checkpointing.

4.2 The C^3 Pre-Compiler

The code the pre-compiler inserts must ensure 1) that the application resumes at the instruction immediately following where a checkpoint was taken, and 2) that the application's variables are saved and restored correctly. A program's variables are saved as binary data, and on restart, the system must restore each variable to its original address — this insures that a dereferenced pointer will still point to the proper object after restart. Accomplishing this requires two separate mechanisms: one to track the execution location, and one to track the application's data.

4.2.1 Checkpointing Application Location

The tracking mechanism involves more than just saving the application PC, it includes saving all information necessary to correctly rebuild the program stack. 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. Figure 2 illustrates the C^3 application transformation and PS manipulation. The pre-compiler inserts a unique label at each checkpoint location, and performs call-graph analysis to insert a label before every function call that might eventually lead to such a location. The pre-compiler inserts code to push and pop the appropriate values onto the PS as these labels are encountered during execution: when a potential checkpoint location is reached and a checkpoint is taken, the runtime saves the PS to the checkpoint file. Each checkpoint thus contains a record of the call sequence leading to the specific checkpoint location to which the checkpoint file data corresponds.

```
function1()
{
        //...
        function2();
        //...
#pragma ccc PotentialCheckpoint
        //...
}
```

(a) Before C^3

```
function1()
{
        if(restart)
                  goto (PS.item(i++))
        //...
        PS.push(1);
label_1:
        function2();
        PS.pop();

        //...
        PS.push(2);
        if(ccc_NeedToCheckpoint())
              ccc_TakeCheckpoint();
label_2:
        PS.pop();
        //...
}
```

(b) After C^3

Figure 2. Position Stack Manipulation

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 as before the checkpoint. The runtime then restores the PS before handing control to the original `main()` function. Each procedure, in turn, uses the PS to call the same function it had called immediately before the checkpoint. When control arrives at the innermost function, the application jumps to just below where the checkpoint was taken. In this 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 the program's position in the static text is restored to the point immediately following the code that saved the checkpoint.

To ensure the restarting application does not execute any code that may affect its state until *after* that state has been entirely restored, the pre-compiler needs to replace the complex arguments (those that require computation to evaluate) passed to a labeled function with temporary variables containing the appropriate values. Similarly, to insure that the PS correctly reflects which function call is currently active, the pre-compiler needs to decompose certain complex statements, such as a statement containing two calls to checkpointable functions, or a return statement that makes a call to one.

4.2.2 Checkpointing Application Data

The techniques described above ensure that each of the restarting application's stack frames begin at the same virtual address as in the original run. The pre-compiler uses a second 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 record from the VDS. Figure 3 shows such manipulations.

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

(a) Before C^3

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

(b) After C^3

Figure 3. Manipulating the Variable Description Stack

When a checkpoint is taken, for each item on the VDS, the C^3 runtime copies the specified number of bytes from the given 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.

Global, local, and file-scoped static variables are pushed and popped as if they were local to the application's `main()` function. In order to handle these variables, C^3 requires access to all of an application's source files. Each local and file-scoped static variable is replaced with an appropriately renamed global variable. A declaration of each of these variables is created in the file that contains the definition of the application's `main()`. Similarly, if such a variable is of a type declared by a `typedef`, the system copies the corresponding `typedef` declaration to that file as well.

4.2.3 Analysis-Driven Optimizations

To minimize the amount of perturbations made to the application code, the C^3 pre-compiler performs some static analysis and code restructuring to reduce the number of inserted instructions that are dynamically executed. Above, we discussed how a call-graph analysis is used to determine which functions could possibly lead to a checkpoint location, and how the pre-compiler will only insert labels (and manipulate the PS) before calls to them. Additionally, and more importantly, functions that are determined to never be on the call stack when a checkpoint is taken do not need to have their local variables saved; therefore, no pushes and pops to the VDS need to be inserted within them.

Another optimization involves hoisting nested scope declarations. To illustrate, assume that the variable `c` in Figure 3 was declared inside a loop body — the code to push and pop `c` to the VDS would be executed for each iteration. Instead of doing this, the C^3 pre-compiler will move (and rename uniquely) `c` to the function-level scope, so that it is pushed and popped only once. Similarly, if a loop body contains only one call to a labeled function, the push and pop to the PS are moved outside the loop body, provided there are no unusual escapes from the body (e.g. `goto`).

4.3 C^3 Runtime

The C^3 runtime is a set of functions that perform two different duties — they are responsible for the saving and restoring of application state, and they provide a checkpointable implementation of select functions from the standard C library. The most interesting of these functions are those that implement the memory allocator — since these are the routines that affect our infrastructure most, only these are discussed here. (Most standard library functions do not need any special checkpointing considerations.) The C^3 pre-compiler converts all calls from the standard (e.g. `malloc()`) function to calls of the version provided in the runtime (e.g. `CCC_malloc()`).

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 virtual address 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 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. 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.

Most SLC systems checkpoint the heap by saving the entire region of memory over which the native allocator has control. 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 actually used. Another, even greater advantage is that C^3 need not save objects that have been deallocated. For certain codes, the amount of deallocated memory can be significant; not saving that memory can dramatically decrease the overhead of taking a checkpoint. The allocator still must guarantee that future calls to `malloc` and `free` behave as expected.

5 Simulation Engine

By using ALC and letting the application checkpoint and restart itself without the support of system components, the restore process is simulated along with the application execution. This technique can be used with any binary-compatible simulator, as long as it supports all system calls required by the C^3 system along with those required by the application. Most importantly, this includes the ability to take a snapshot of the heap of the application and to restore that heap snapshot at the original virtual memory addresses. For instance, the C^3 `CCC_malloc()` allocates memory within an explicitly managed region of virtual memory that has been allocated using the `mmap()` system call. This allows the specification of the starting address for the requested memory chunk, matching memory layouts from the original run and restored execution.

For this study we use SimpleScalar Version 4.0 Alpha-Sim [8]; this is the latest version of one of the most widely used simulation infrastructures for Computer Architecture research. Unfortunately, the publicly available version of the simulator does not have an implementation for `mmap()`. Given that this simulator executes system calls by proxy, we developed a `mmap()` proxy that manages a heap allocated in the application's virtual memory space as maintained by the simulator. The resulting code for `mmap()` is about 20 lines of C.

6 Early Experiences

We configure the simulator as closely as possible to the validated model of a Compaq DS-10L Alpha Server, as described in previous studies [7, 8]. The memory system is a 64KB, two-way associative L1 cache with 64-Byte lines and three-cycle latency followed by a 2MB direct-mapped L2 cache with a 13-cycle latency. We use the default eight-entry victim cache, and eight MSHRs per cache. This version of the simulator models the bus and the SDRAMs, in contrast to previously released versions of the toolset.

For our initial experiments, we use six codes from the Splash-2 suite of benchmarks. These numerically intensive kernels and applications are designed for the evaluation of shared memory architectures; we use them for this initial study because we had already instrumented them with the C^3 state-saving methods for research into the issues surrounding ALC for shared memory applications. All benchmarks demonstrate a cyclic behavior in that they have well defined loops for the manipulation of the input data set. In instrumenting the codes, we place our (`#pragma ccc PotentialCheckpoint`) statements at the inner edges of these loops, as that seems to be a reasonable place where an architect may wish to commence cycle-accurate simulation. A listing of the problem sets for the benchmarks and a description of the (`#pragma ccc PotentialCheckpoint`) locations can be found in Table 2. Table 2 also lists the sizes of the heap checkpoint file for each of the applications.

6.1 Instrumentation Overhead

Since ALC instruments the application to save its own state, we would expect simulation statistics (such as IPC and cache miss rates) of the ALC version to be different from those of the uninstrumented version. Ideally, we would like these differences to be small. Figure 4 shows a comparison of several simulation statistics for each benchmark. The bars represent a normalized comparison of the instrumented code to the original code. For all of the applications, the results are encouraging. For most statistics, the measured change is less than 3% in the instrumented code, although there are a few notable exceptions.

The lu-c (lu-continuous) kernel shows an increase in the IPC by nearly 10% for the instrumented code over the original code. We believe this variation can be traced to the L1 data cache hit rate for lu-c, which is measured to be almost 3% greater in the instrumented code. With a higher cache hit rate, there would be fewer stalls waiting for data from memory, and thus, a higher measured IPC. The increase in the L1 cache hit rate is difficult to attribute directly to the instrumentation code. We expected to see a slight decrease in this statistic due to increased occupancy for the instrumentation data structures, but, across the board, we see a slight increase, except in the case of barnes and radix. This increase cannot be due to the instrumentation code warming the caches, since the instrumentation code does not directly access any data structure associated with the original code except when checkpointing. The measurements presented here do not include checkpoint overhead. Neither can the increase be due to the instrumentation code itself; this code is executed very few times compared to the original program. Instead, we believe that the increase is due to either a change in the compiled code for the instrumented version, or a change in memory contents because of address skewing due instrumentation data structures. Finally, our cus-

benchmark	problem size	checkpoint location	checkpoint size (MB)
barnes	16384 bodies, 4 steps	in code.C after each time step (call to `stepsystem()`)	51.8
fft	2^{20} points	after each call to `FFT1DOnce()`	50.7
lu-c	512×512 matrix	at the end of the outermost loop of `lu()`	2.1
ocean-c	514×514 ocean, 4 steps	in slave1.C after each time step (call to `slave2()`)	58.6
radix	5 Million keys	at the end of the main loop of `slave_sort()`	45.0
water-n^2	512 molecules, 4 steps	in mdmain.C at the end of each time step	0.5

Table 2. Application-Level Checkpoint Characteristics

tom memory allocation routine might be better suited for the data structures in these benchmarks than the standard `malloc`. Verifying these hypotheses is part of our research agenda.

We also saw an increase in IPC for the ocean-c (ocean-continuous) and radix benchmarks. Here, we cannot link this increase to the L1 data cache hit rate as that statistic was unchanged for ocean-c, and reduced for radix between the two runs. Instead, the increase is probably related to the increase in the number of instructions executed in the instrumented version. We believe that these additional instructions are either directly related to the instrumented code, or can be attributed to a change in the compilation of the base code due to the insertion of the instrumentation code. Further, we expect that these additional instructions execute with an IPC higher than the rest of the code, thus boosting the total IPC.

One other statistic that shows a surprising result is the number of instructions executed for water-n^2 and barnes. In both cases, the instrumented code shows a slight decrease. We attribute this to a difference between the custom memory allocation procedure versus the standard `malloc` call.

Another way that we quantify the amount of change our instrumentation introduces is by analyzing the dynamic basic block profile of the benchmark executions. We measure a subset of the changes by noting the number of basic blocks executed that correspond to instrumentation code from the C^3 library. Our results are shown in Table 3 .

Here the results are also encouraging, as they show that new basic blocks comprise only a small fraction of the total number of executed basic blocks. We expect that with a larger number of new basic blocks, we will see a larger perturbation of the simulation statistics. This theory is shown to hold true in the case of radix — this benchmark showed the largest number of new basic blocks as well as the largest change in IPC. These numbers only tell part of the story. We believe that most of the (minimal) change to the simulation statistics is due to changes in the way that the code compiles due to instrumentation code being inserted. Our research agenda includes determining the number of *changed* basic blocks that are executed for a given instrumented benchmark, as well as the number of new basic blocks.

7 Conclusions and Future Work

In this paper we have presented SimSnap — a method to replace time-consuming fast forwarding via functional simulation with native execution. We use Application-Level Checkpointing mechanisms to save program state at the end of the native execution and to resume within a cycle-accurate simulation. For this purpose, we use the C^3 system to transparently instrument target applications with code for state saving and restoration. Because it is the application that saves and restores its own state, any simulator that can execute the application can take advantage of restoring program state from a checkpoint.

Using SimSnap, we have demonstrated the potential for dramatic reductions of simulation experiment times, while keeping the the perturbation of the simulation statistics due to the instrumentation minimal. Although we use a specific simulator to demonstrate the utility of our approach, our methods are widely applicable to any simulation environment.

These initial results are encouraging, and hence we plan to expand this work on several fronts. Most importantly, we will reduce the already small amounts of perturbation introduced into application codes by our instrumentation methods. We will expand the C^3 system's data flow analysis to minimize the amount of state saved at a checkpoint. And we will study optimizations that reduce the execution of state-saving code such that we only save static variable pointers when a checkpoint is scheduled to be taken.

In our current work, we perform the native execution of the code on a machine whose ISA and architecture match that of the machine being simulated. Because the SimSnap checkpoint is at the application level, we postulate that it is possible to run natively on a machine that does not have the same ISA as the simulated machine. If we can demonstrate this, it will further generalize the utility of the SimSnap methods. This problem is akin to migrating applications within a heterogeneous grid computing environment.

To evaluate the general utility of SimSnap, we will use SimPoint [23] to select checkpoint locations. We then instrument the SPEC CPU benchmark suite, and will offer these instrumented SPEC codes and the accompanying checkpoints to the research community. This will give re-

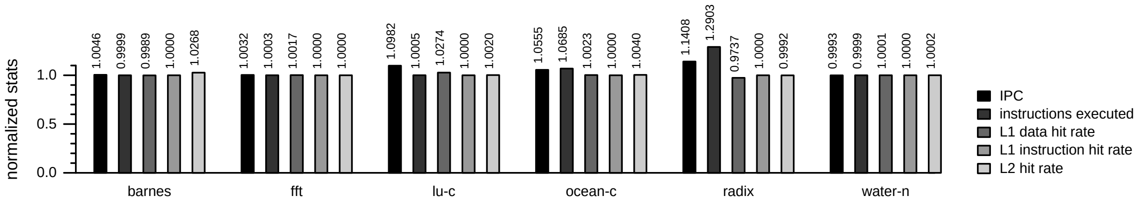

Figure 4. ALC-Enabled vs. Uninstrumented Benchmarks

benchmark	original version basic blocks		instrumented version basic blocks			instrumented version C^3 library blocks	
	static	dynamic	static	dynamic	% dynamic increase	static	dynamic
barnes	14741	389749369	15259	391377819	0.41	462	730
FFT	11321	170801497	11907	172584578	1.04	462	70593
lu-c	11188	94194286	11805	94298725	0.11	462	1382
ocean-c	13225	72074717	14106	73978821	2.64	462	12021
radix	8805	45015951	9423	46447626	3.18	437	840
water-n^2	14334	152280215	14962	152318318	0.02	462	37729

Table 3. Application-Level Checkpoint Effects on Basic Block Counts

searchers easy access to fast and accurate simulation technology, and will help to achieve equivalent experimental setups and comparable (and repeatable) results. The potential uses for our technology are far-reaching. For instance, we envision using SimSnap for extensive performance debugging and engineering: a critical loop of an application could be instrumented for state saving and restoration, and if the real-time evaluation of a particular statistic (e.g., cache miss rate) were to reach a given threshold, the application would save a checkpoint. This could then be restored within a cycle-accurate simulator, allowing for a detailed evaluation of the anomaly. This approach would yield a powerful performance analysis tool for application programmers, efficiently combining the advantages of native execution and detailed simulation information. Other applications include experiment validation when partial simulation is used to speed experiment time: running the remainder of the benchmark natively to completion allows us to better verify the functional correctness of the simulator. We expect that applying this approach to parallel system simulation will expand the kinds and sizes of the experiments researchers can perform. Finally, using ALC-enabled benchmarks that periodically checkpoint their state makes long-running simulations fault tolerant. We are eager to make our tools available to the research community, which will no doubt find other interesting uses for the technology we're developing.

Acknowledgments

The authors would like to thank Keshav Pingali and the other members of the Intelligent Software Systems group for their help in developing the Cornell Checkpoint Compiler system. We also thank Shafat Zaman for his support in getting the Splash codes to work with the C^3 system. This work was supported in part by the National Science Foundation under awards ITR/NGS-O325536, CCR-0121401, EIA-0103723, ACI-0090217, and ACIR-0085969.

References

[1] 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.

[2] G. Bronevetsky, . Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant mpi system. In *Proceedings of the 2003 International Conference on Supercomputing*, pages 234–243, June 2003.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 84–94, June 2003.

[4] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 IEEE International Conference on Computer Design*, pages 468–477, Oct. 1996.

[6] S. P. E. Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2000/, 2000.

[7] R. Desikan, D. Burger, and S. Keckler. Measuring Experimental Error in Multiprocessor Simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, June 2001.

[8] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2001.

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

[10] D. Jefferson. Virtual time. *ACM Trans. Prog. Lang. Syst.*, 7(3):404–425, 1985.

[11] J. H. Jr. and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 IEEE International Conference on Computer Design*, Sept. 2001.

[12] J. H. Jr. and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2003.

[13] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. http://www.arctic.umn.edu/˜lilja/papers/minnespec-cal-v2.pdf, 2000.

[14] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 11, June 2002.

[15] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Proceedings of the Third IEEE Workshop on Workload Characterization*, pages 102–110, Sept. 2000.

[16] M. Litzkow and M. Solomon. Supporting checkpoint and process migration outside the UNIX kernel. In *Usenix Winter Technical Conference*, pages 283–290, Jan. 1992.

[17] J. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, winter 1995.

[18] J. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under UNIX. In *Usenix Winter Technical Conference*, pages 213–223, Jan. 1995.

[19] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software — Practice and Experience*, 29(2):125–142, 1999.

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

[21] J. Robinson, S. H. Russ, B. Heckel, and B. Flachs. A task migration implementation of the message-passing interface. In *Proceedings of High Performance Distributed Computing*, pages 61–68, Aug. 1996.

[22] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Sept. 2001.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[24] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.