Proc. of the IEEE Int. Symp. on Performance Analysis of Systems and Software(ISPASS), March 2005, Austin, TX, pp. 78-88.

Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification

Jeff Ringenberg, Chris Pelosi, David Oehmke, and Trevor Mudge
The University of Michigan
Electrical Engineering and Computer Science
1301 Beal Ave., Ann Arbor, MI 48109-2122
{jringenb, cpelosi, doehmke, tnm}@eecs.umich.edu

Abstract

With the proliferation of benchmarks available today, benchmarking new designs can significantly impact overall development time. In order to fully test and represent a typical workload, a large number of benchmarks must be run, and while current techniques such as SimPoint and SMARTS have had considerable success reducing simulation time, there are still areas of improvement. This paper details a methodology that continues to decrease this simulation time by analyzing and augmenting benchmark binaries to contain intrinsic checkpoints that allow for the rapid execution of important portions of code thereby removing the need for explicit checkpointing support. In addition, these modified binaries have increased portability across multiple simulation environments and the ability to be run in a highly parallel fashion. Average speedups for SPEC2000 of roughly 60x are seen over a standard SimPoint interval of 100 million instructions corresponding to a reduction in simulation time from 3.13 hours down to 3 minutes.

1. Introduction

Methods for properly simulating and testing new microarchitectural concepts have been proposed ever since computers were first designed. Most techniques require that a large number of benchmark programs be run with many different input datasets for an equally large number of sample configurations of the design. Unfortunately, these requirements create a burdensome number of simulations that can significantly increase the design time of a product. For example, if 30 benchmarks were used with an average of five datasets per benchmark and there were 20 different configurations to be tested, this would require a total of 3,000 individual simulation runs. With each simulation potentially taking days or even weeks to complete, this amount of simulation can quickly become a serious bottleneck. Compounding this problem is the fact that the most important and representative phases of a benchmark's code usually execute in the middle of the simulation and there is no easy way to execute only this portion without additional support.

Many different methods have been proposed to help alleviate these burdens; however, these methods often lead to decreased accuracy in the simulation or require that the simulator have additional functionality that does more than simply emulate the design. For instance, if benchmarks or their respective datasets were shortened or removed, this new set of benchmark and dataset combinations may no longer represent a proper workload profile for the design. In addition, methods such as checkpointing and fast forwarding meant to enable the simulation of only certain essential parts of the code require that the simulator have these abilities built in. For checkpointing, it would require the ability to capture the state of the machine at some point in the simulation so that later simulations could then load in the state and start again at that point. For fast forwarding, a simulator would need to have a separate, faster mode of simulation that ignores many of the details of the design being simulated while at the same time guaranteeing its functional correctness. These features are not always available or may not be economical to implement if the simulator will have limited distribution or is meant for a small amount of use.

This paper will present a method by which changes are made to a benchmark binary itself that will allow for the execution of only essential parts of the code. This is done by first analyzing the code within provided intervals of execution and then generating a set of checkpointing instructions. These instructions effectively recreate the system environment that the interval would see at its start had the binary been

executed normally. The instructions can then be run at the start of the benchmark followed by only the code in the interval. This technique, therefore, creates a form of intrinsic checkpointing that removes the requirements of checkpointing and fast forwarding from the simulator and places it into the program itself. When combined with tools such as SimPoint, it becomes possible to properly benchmark a system in a fraction of the time compared to using the original benchmarking binary. For simulators such as Verilog models where checkpointing and fast forwarding are very difficult to implement, this technique allows for the simulation of benchmarks that in the past were severely hindered by their execution times.

The rest of the paper is organized as follows. Section 2 will present previous work on techniques aimed at reducing simulation time. Section 3 will discuss the basic method of intrinsic checkpointing along with our implementation. Section 4 will give an example of some Alpha checkpointing code that is generated with our methods and section 5 will discuss our results. Finally, section 6 will compare and contrast a similar method to the one proposed and section 7 will give concluding remarks along with our thoughts on future work.

2. Previous work

There are many different techniques aimed at reducing simulation time and each has its various strengths and weaknesses. The majority can be broken into several groups including dataset reduction, statistical sampling, and program phase detection and classification.

One of the simpler methods to decrease simulation time, referred to as dataset reduction, reduces the amount of data that the program must process and therefore simulation will finish quicker than if the benchmark had massive amounts of data to handle. This method, used in the MinneSPEC [1][2] workloads from the University of Minnesota, has the advantage that, once the reduced workloads have been characterized, no analysis needs to be done on the program prior to simulation. The workload selection chooses smaller datasets through a process that systematically reduces larger workloads while still maintaining their simulation characteristics, however, this process requires many steps of manual tuning that threaten to remove the representative nature of the datasets. In addition, the reduced workloads, if sufficiently small, can fit entirely within a typical cache and no longer test the functionality of the memory subsystem.

The Sampling Microarchitecture Simulation (SMARTS) framework [3] from Carnegie Mellon University is one simulation tool relying on statistical sampling that provides quick and accurate results while only having to simulate in detail a small portion of the benchmarking program. The program is divided into a series of equal length intervals each of which is further broken into a fast-forward component that stresses speed over accuracy, a warm-up component that refreshes certain architectural elements such as caches and branch predictor structures that were neglected in prior fast-forwarding segments, and finally a detailed simulation component that runs with the most accuracy. To begin the simulations, SMARTS starts the benchmark in fast-forwarding mode and runs for an initial number of instructions provided by the user. At this point, the first interval will begin the warm-up period that updates the state of necessary components mentioned above. The detailed simulation segment then begins and is followed by the fast-forwarding component that finishes the first interval. This systematic sampling of warm-up, detailed simulation, and then fast-forwarding repeats itself until the benchmark completes.

Another tool that is useful for fast and accurate benchmarking is SimPoint [4][5][6] from the University of California, San Diego. Like SMARTS, SimPoint only simulates in detail small portions of the benchmark leaving the rest for fast-forwarding. However, the methods it uses to decide which portions of the program will be used for the detailed simulation significantly differ from SMARTS. SimPoint makes use of the fact that most programs are comprised of similar sections of execution referred to as phases. Since each of these phases looks much alike, SimPoint attempts to find them and then simulates only one of them, referred to as a simulation point interval, from each set of similar phases. These simulation intervals can either be executed like those in SMARTS or they can be checkpointed. For execution like that in SMARTS, the intervals can be given preference if they occur earlier in time creating an “early SimPoint” method that makes it possible for the serial simulation to finish quicker. The checkpointed execution of the intervals requires the support of the simulation environment to handle the saving and restoring of checkpoints. After execution, results can then be extrapolated based on the frequencies of the different phase sets in the code.

A final method for decreasing simulation time, the newly proposed SimSnap tool [7] from Cornell University, is a similar approach to that proposed in this paper. Since a discussion of its techniques would be better suited in a compare and contrast fashion, its

description will be reserved until after our methodology has been described in detail.

3. Intrinsic checkpointing

Although current techniques like those outlined above are effective at decreasing the necessary simulation time of benchmarks, they still suffer from several problems that can be improved upon in order to obtain even faster results. For MinneSPEC, the reduced workloads may no longer represent a typical application and their reduced size hinders their ability to properly test cache functionality. For SMARTS, it is still necessary to run through the entire program in order to obtain results since the detailed simulation points occur at equal intervals through the entire simulation. For SimPoint, in order for the simulation to complete quickly the user either has to accept less accurate results by using early simulation points or must use a simulator that is capable of checkpointing.

The following section will propose a method by which simulation intervals, either provided by SimPoint or other tools, could be analyzed such that fast-forwarding, early SimPoints, and simulator checkpointing would no longer be necessary and each detailed simulation interval could be executed independently and in parallel. This method will, in effect, intrinsically checkpoint the simulation interval code allowing it to restore only the environment that an interval would need to utilize and nothing more.

3.1. Basic method

In the most basic sense, the components of a microarchitecture and execution environment that a program must deal with are the register file, memory, and system calls such as file I/O. If it were possible to track what changes these components underwent in the program preceding a detailed simulation interval and decide which changes the interval actually needed to occur for its proper execution, then a new portion of code could be inserted before the code for the simulation interval to effectively bring about these same changes. This checkpointing of only the components could then run in a fraction of the time compared to the original execution. The new intrinsic checkpoint code, along with the code from the simulation interval, could then be removed from the original code to create a new "microbenchmark".

This new code, as long as its representative weight is preserved for analysis and it does not contribute to statistical pollution, could be used exactly like a simulation interval that in the past would have required explicit checkpointing or excessive fast-forwarding. In addition, since the SimPoint analysis of the code is done independent of the underlying microarchitecture and simulator, these new microbenchmarks could be run on many different simulators, Verilog models, and even native hardware to represent the original benchmark as long as the original benchmark's code profile would not need to be changed. In a size limited environment, this would allow for the execution of prohibitively large benchmarks without a need for additional resources.

3.2. Caveats

There are, however, several things that need to be explored before this idea can be considered effective. The first of which is whether this methodology will be applicable across different benchmark suites. For example, memory intensive benchmarks such as TPC could work with an extremely large amount of data that would need to be loaded into the memory before the simulation interval could be run. However, the worst penalty, measured in the number of loads necessary to bring the memory up-to-date, would be equal to a small multiple of the instructions in the interval since the instructions in the interval can only reference a finite amount of memory defined by the ISA. This overhead, especially if the interval occurred late in the benchmark, would still be much less than the original method of fast-forwarding through all the pre-interval code. This same logic could be applied to the presence of large amounts of file manipulation or changes to the overall state of the system. Since there is only a limited amount of work being done in the interval, there should only be a limited amount of pre-interval work to be done.

Another point to consider is the importance of warming up the various microarchitectural elements before the detailed simulation interval can occur. This problem is addressed in both SMARTS and SimPoint, and there are many different methods that can be used. A recent paper on SimPoint published in SIGMETRICS [5] summarized the various techniques including assuming all first accesses to important structures were hits [8][9], calculating the working set of the important structures in the interval and executing all instructions before the interval that will satisfy this working set [10][11], or keeping certain structures "warm" during the fast-forwarding stage by updating only important parts [3]. The paper states that for many benchmarks, the method that assumes hits on first accesses is fairly accurate, but that the two other techniques provide the most accurate results. The technique that keeps the structures warm is obviously not a good candidate for the proposed approach of creating pre-interval code since it requires the entire

fast forwarding segment to be run. However, the technique that does a working set analysis on the structures is very promising since the code to warm the structures can be built into the pre-interval code and not included in the statistics. One possible side effect of the pre-interval code execution is that the structures could be warmed up simply by its execution alone. This would remove the need for any of the above methods and the code would not only be checkpointing itself, but also intrinsically warming up the interval.

A final issue that will need to be dealt with is the method by which the simulation ends following the execution of the interval. In a typical simulator, it is possible to simply tell the simulation when to stop by providing a count for the number of cycles to simulate. This, however, requires that the simulator have the ability to track the number of cycles on a per cycle basis from within the simulation. In addition, if the goal of creating a set of microbenchmarks is to be able to move them to native hardware, the hardware may not have the ability to track cycles. Thus, it will become necessary to insert a halt operation at a logical point in the code that will allow only the interval and its respective checkpointing code to be executed. Unfortunately, it is not as simple as replacing the instruction corresponding to the end of the interval with a halt instruction since this instruction may be encountered prior to the interval's end due to things such as looping code. Therefore, there will need to be an additional level of analysis that occurs during the checkpoint code generation phase that identifies an instruction that can be replaced with a halt that will still preserve the most of the interval's behavior as possible.

3.3. Implementation

In order to create the necessary code to bring the simulation interval up to date, an analysis of the benchmark must be done to find out what state needs to be restored for the proper execution of the simulation interval. The register file is by far the easiest component whose usage can be tracked and restored. To do this, a simple snapshot of the register file is taken immediately before the first instruction in the simulation interval and its current values are noted. These values are then restored with simple load instructions in the checkpoint code to bring the register file back up to date.

Memory, on the other hand, proves to be much more troublesome, especially if there are many instructions within the simulation interval that depend on values in memory that were modified before the interval began. To track these modifications, a copy of the initial memory, INIT_MEM, is created at program start and the program is then allowed to fast-forward up to the simulation point in question. At that point, a copy of memory, CHECK_MEM, is saved to hold any changes to the memory from the pre-interval code. Next, as the simulation interval progresses, each time a load is encountered within the simulation interval, if the value in INIT_MEM differs from the value in the current memory, CURRENT_MEM, then it is known that some instruction in the pre-interval code stored a value to that memory location. Consequently, the memory location and its value from CHECK_MEM are logged so that later it can be turned into a "fixup" store in the checkpoint code. The memory location in INIT_MEM is then updated with the value from CURRENT_MEM so that this situation will not be run across again. In the case of a store to memory within the simulation interval, the value is simply stored to both CURRENT_MEM and INIT_MEM since this store effectively overwrites any loads that occurred in the pre-interval code.

System calls (syscalls) also create a rather difficult problem. File output syscalls are ignored since, unless the output file is used later in the code as input, it doesn't affect the results of the program. File input, however, modifies file pointers in the program and the usage of all input files is logged so that all relevant pointers can be updated in the checkpoint code. This is done by noting the work that occurs in the simulation for file handling and keeping track of the various values that control file manipulation. Other system calls, such as those that change directories, modify file permissions, or open/close files also have these types of methods that track their usage. Again, code is inserted into the system call handlers of the simulator to have their requisite input values loaded using analysis similar to the register and memory analysis for the simulation interval above. Then, the actual system call is run from within the checkpoint code. Since there are only a few syscalls that can actually affect the system environment, the generated code for syscall checkpointing is small. The remainder of the syscalls that occur prior to the interval are either ignored or, if they modify memory or register state, the normal simulation interval analysis discovers these changes and generates the corresponding checkpoint code.

```
# <syscall stores>
st  <addr>,<imm> # store imm @ addr
# <syscall register loads>
ldi rX, <imm>    # load imm into reg rX
SYSCALL(example) # execute the syscall

# Memory Restoration
stb  <addr>, <8-bit>  # store byte @ addr
sth  <addr>, <16-bit> # store half @ addr
stw  <addr>, <32-bit> # store word @ addr
stqw <addr>, <64-bit> # store quad @ addr
# Register File Restoration
ldi rX, <imm> # load imm into INT reg rX
ldi fX, <imm> # load imm into FP reg fX

jmp <interval_start>  # jump to interval
```

Figure 1. Pseudo-assembly template for intrinsic checkpointing code

```
ldi  r0,  0x000000000000002d
ldi  r16, 0x00000001200028f8
ldi  r17, 0x0000000000000000
SYSCALL(open)

ldi  r0,  0x0000000000000003
ldi  r16, 0x0000000000000006
ldi  r17, 0x0000000140039d08
SYSCALL(read)

stb  0x000000014003a200, 0x45
stb  0x000000014003a201, 0x6f
ldi  r0,  0x0000000000000004
ldi  r16, 0x0000000000000001
ldi  r17, 0x000000014003a200
SYSCALL(write)
```

Figure 2. Pseudo-assembly for pre-interval syscall

The final step for creating an intrinsically checkpointed benchmark that will only execute a given interval of code is the actual insertion of the checkpoint code into the original benchmark binary which we accomplished by using a Perl script. The simplest method for inserting the new code is to either put it at the end of the code section in the binary or to place it into a separate section; our method makes use of the latter method. During insertion, it is important that the restoration of the registers occur at the end of the checkpointing code since the operations for restoring the memory need to make use of the registers. After the insertion, the new binary has its starting point set to the beginning of the checkpointing code. Next, at the end of the checkpointing code a final jump is inserted that will move the execution to the beginning of the simulation interval. Depending on the instructions available, this can be done with either an explicit jump to a PC provided in the instruction, or a jump to a PC value that is stored in a register. If a register is needed, as is the case with our method, then it will have to be one that is going to be overwritten before it is used inside the interval so that proper execution will be guaranteed. Figure 1 shows a template of how a typical, pseudo-assembly checkpoint file is organized prior to being converted into machine code and inserted into the binary. First, any syscalls that must be run prior to the interval are output, followed by the section to restore the memory, and then register restoration occurs. Finally, the jump to the start of the simulation interval is executed.

4. Alpha Example

As an example of this method, the following section will present an analysis of the eon benchmark from SPEC2000 [12]. The SimpleScalar [13] simulator targeted at the Alpha ISA [14] was modified to do the analysis described in the previous sections and to output a pseudo-assembly file that contains the essential syscalls, stores for memory restoration, and loads for register restoration all of which will comprise the checkpointing code that will need to be inserted into the original binary. Figure 2 lists a small part of the output file that was generated for a syscall and Figure 3 gives a sample of the loads and stores for memory and register restoration.

For the syscalls, only the registers and memory that the syscall will use are restored. A good example of this is the write syscall where the data to write to the file is loaded into memory prior to being written to memory with the syscall. As the results show, the pseudo-assembly will generate only one instruction for each memory location restoration. However, this pseudo-assembly cannot be directly translated into machine language since the Alpha only has 32-bit instructions. Therefore, the 64-bit addresses and data will need to be broken up into a series of optimized machine instructions that can produce the same effect. For example, all the 64-bit addresses and data that are needed in the checkpointing code for memory

```
# Stores for memory restoration
stqw 0x000000011ff95948, 0x3ff688818757f6a3
stw  0x000000011ff95914, 0x00000001

# Integer registers restoration
ldi  r0,  0x000000000000000c
ldi  r1,  0x0000000140003f060
# Floating Point registers restoration
ldi  f0,  0x3fc7cd8dad5988b8
ldi  f1,  0x000000011ff95c20
# Special/Control registers restoration
ldi  FPCR, 0x0000000000000000
ldi  UNIQ, 0x0000000140008f08
```

Figure 3. Pseudo-assembly for register and

```
#1 stqw <Addr1 = 0xaaaabbbbccccdddd>,
        <Value1 = 0xddddccccbbbbaaaa>
#2 stqw <Addr2 = 0xbbbbccccdddd0000>,
        <Value2 = 0xeeeeddddccccbbbb>
#3 stqw <Addr3 = 0xbbbbccccdddd0008>,
        <Value3 = 0xffffeeeeddddcccc>
.data:
0 : 0xaaaabbbbccccdddd (Addr1)
8 : 0xddddccccbbbbaaaa (Value1)
16: 0xbbbbccccdddd0000 (Addr2)
24: 0xeeeeddddccccbbbb (Value2)
32: 0xffffeeeeddddcccc (Value3)

.text:
# Load start address of .data section
# into r1 (4 instructions)
ldqw r2,0(r1)      # set up Addr1
ldqw r3,8(r1)      # set up Value1
stqw r3,0(r2)      # checkpoint store #1
                   # with Value1 @ Addr1
ldqw r2,16(r1)     # set up Addr2
ldqw r3,24(r1)     # set up Value2
stqw r3,0(r2)      # checkpoint store #2
                   # with Value2 @ Addr2
ldqw r3,32(r1)     # setup Value3
stqw r3,8(r2)      # checkpoint store #3
                   # with Value3 @
                   # Addr3 = Addr2 + 8
```

Figure 4. Converting pseudo-assembly to Alpha machine instructions

restoration could be written into a section of the binary set aside for data. A register could then hold the starting address of this section of the code and offsets could be used in conjunction with the register to load data into registers that would then be stored into the appropriate location in memory. Figure 4 gives an example of converting several 64-bit address/data stores into a series of machine instructions. As the figure shows, it is possible to convert one checkpoint store that requires both a 64-bit address and 64-bit data into a series of instructions that could take at most seven, or as little as two, machine instructions. For register restoration, the same process could be used to restore a register in at most five, or as little as one, machine instruction.

5. Results

The following section will present results of the proposed methods when used on 19 of the SPEC2000 benchmarks targeted at the Alpha ISA using the reference datasets. It should be noted that if the datasets need to be changed, entirely new intrinsically checkpointed binaries must be created since, as with explicit checkpointing, the system state may be different prior to the interval of execution. For our results, the simulation interval used was obtained using SimPoint with a specified interval length of 100 million instructions. Since the current intrinsic checkpointing method only creates code for a single interval, SimPoint was limited to only choosing one interval that would attempt to represent the entire benchmark. Thus, the representative nature of the intrinsically checkpointed binary will be, as in all situations, governed by the accuracy of the interval choosing mechanism.

For validation purposes, the simulation intervals for both the original binaries and the intrinsically checkpointed binaries were simulated in detail and statistics such as IPC, memory references, and branch counts were recorded for comparison. For each of the benchmarks, variations in the statistics were exceedingly rare and IPC had no variation. In addition to checking statistics, we verified our technique was checkpointing properly by doing cycle-level register file comparisons during the simulation interval on several benchmarks. Therefore, it can be concluded that the interval was properly checkpointed.

5.1 Code breakdown

The first set of results divides the checkpointing code into four different sections: syscall loads for register restoration, syscall stores for memory restoration, interval loads for register restoration, and interval stores for memory restoration. The results are further broken down into whether or not the possibly unnecessary file output syscalls encountered prior to the simulation interval are included in the checkpointing code, referred to as “dowrites” if file output is included and “nowrites” if not. The syscall data corresponds to the number of loads and stores that must be run in order for the essential, pre-interval syscalls to provide proper program behavior. The interval data lists how many loads and stores are required to checkpoint the simulation interval. Note the number of loads needed for register restoration of the simulation interval is always constant since there are only a finite number of registers needing restoration. Table 1 shows the breakdown with the results averaged for integer, floating point, and all SPEC2000 benchmarks. As the table shows, register restoration for the interval is inconsequential compared to the rest of the checkpointing code.

Syscalls play a larger part in the checkpoint code if file output is included; however, they still only comprise 2.4% of the total checkpointing code on average. In addition, the results for the integer benchmarks have much less checkpointing code due to the fact that they are less focused on memory activities and will, therefore, require less code to restore it.

Table 1. Breakdown of pseudo-assembly checkpointing instructions

	SpecInt Avg		SpecFP Avg		Spec2K Avg	
	dowrites	nowrites	dowrites	nowrites	dowrites	nowrites
syscall loads	2,035	884	16,400	9,974	10,351	6,146
syscall stores	37,565	57	6,998	49	19,868	52
interval loads	68	68	68	68	68	68
interval stores	485,417	485,417	1,792,717	1,792,717	1,242,275	1,242,275

5.2. Code overhead

The next set of results displays how many Alpha instructions the checkpointing code needs to execute compared to the number of instructions that would need to be executed in both the simulation interval and the fast-forward section prior to the interval if intrinsic checkpointing were not available. As before, the results are broken into whether or not file output is included in the syscall statistics. As Figures 5 and 6 show, the checkpoint code represents only a small part, less than 3% on average, of the executed interval code and an extremely small part, less than 0.01%, of the fast-forwarding interval. Therefore, it is expected that the execution of the code with checkpointing included will be much faster than having to fast-forward to the simulation interval itself.

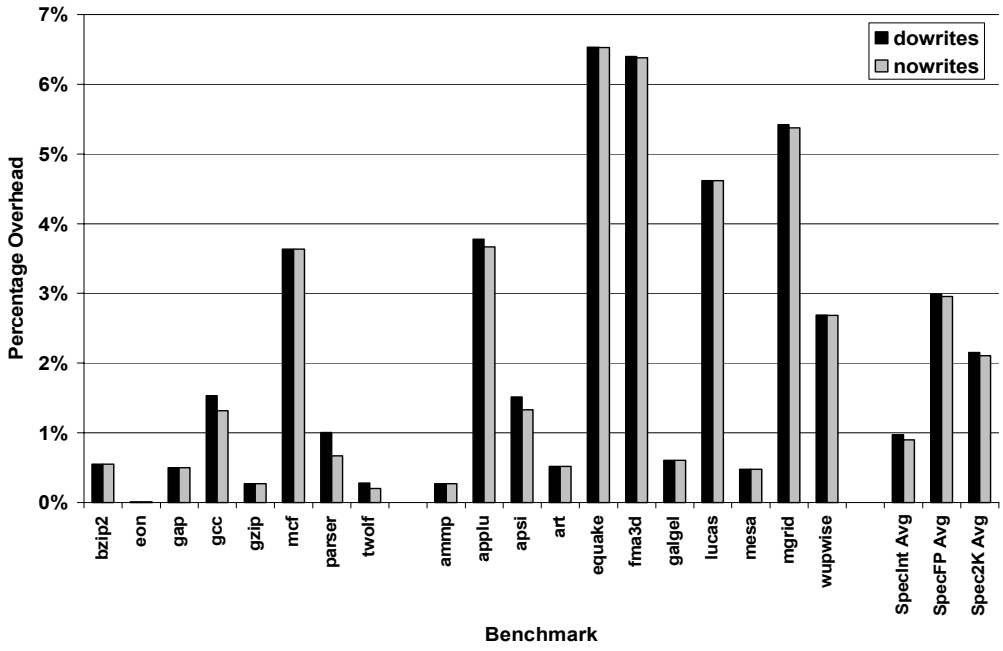

Figure 5. Overhead of checkpoint code compared to simulation interval

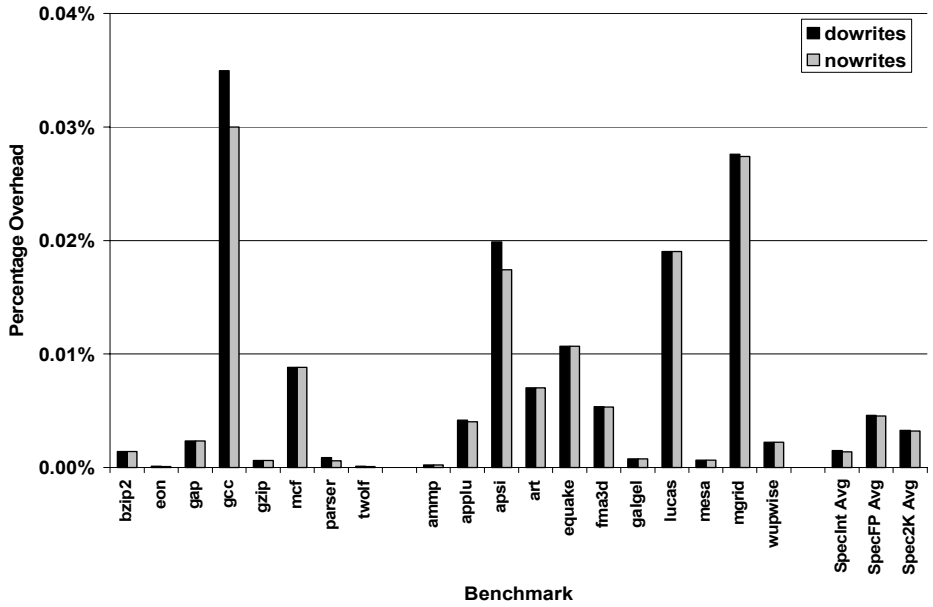

Figure 6. Overhead of checkpoint code compared to fast-forward interval

5.3. Speedup

To measure the decrease in simulation time, the original code and the intrinsically checkpointed code were both fast forwarded to the start of the simulation interval and then the interval was simulated in detail using the default configuration of the SimpleScalar sim-outorder simulator which is essentially a four instruction wide fetch/decode/issue/commit out-of-order datapath processor with speculative execution. In the case of the original binary, a considerable amount of time was needed for fast forwarding, whereas for the intrinsically checkpointed binary little time was needed. Figure 7 shows the speedup in runtime of the checkpointed binary, which includes pre-interval syscall writes, over the initial binary when run on a Pentium4 2.4GHz+ processor with 1 GB of RAM. The speedup ranges from roughly 5x for gcc up to nearly 180x for fma3d with an average of roughly 60x. The relatively small speedup of gcc is due to the

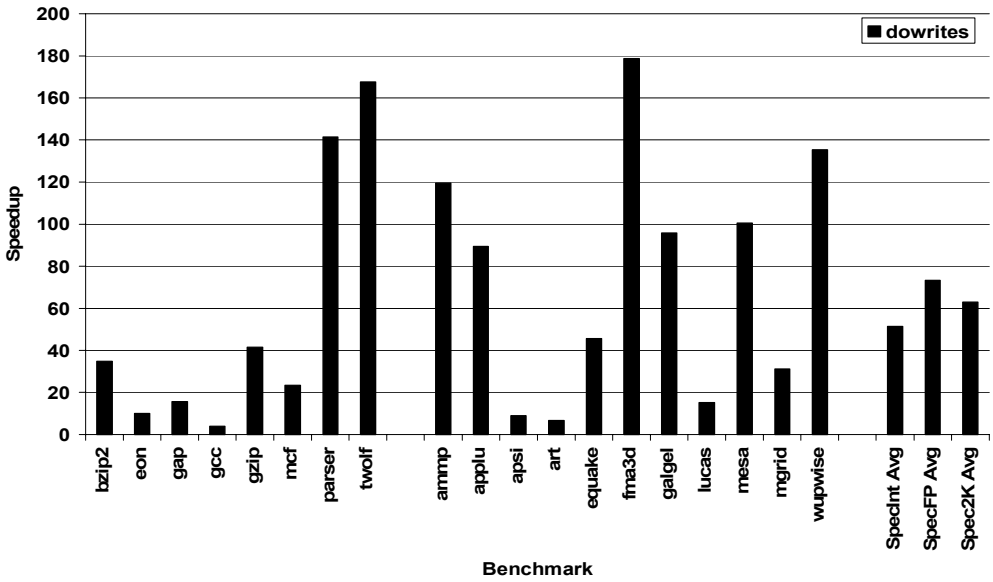

Figure 7. Speedup of intrinsically checkpointed code over initial code

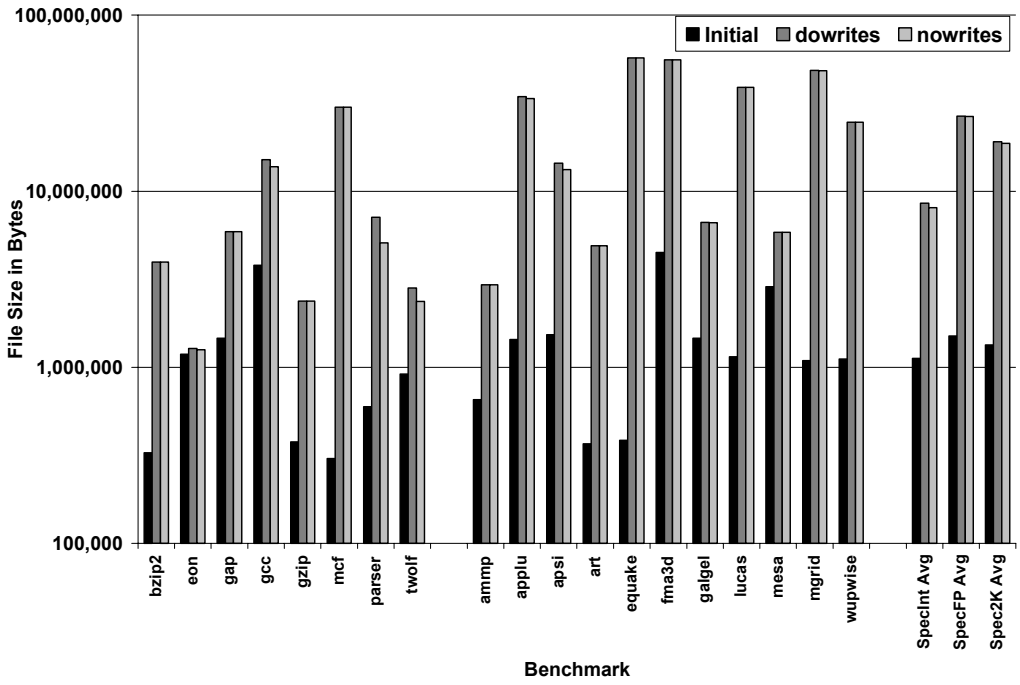

Figure 8. Increase in file size (measured in bytes)

fact that the fast forwarding interval prior to its simulation interval is less than 5 billion instructions whereas for fma3d it is 184 billion instructions. In terms of wall clock time, gcc took 3.7 minutes when checkpointed and 14.5 minutes when not and fma3d took 3 minutes when checkpointed and 8.9 hours when not. On average, the runtime of an intrinsically checkpointed binary was 3 minutes as opposed to 3.13 hours when not.

An interesting note is that if the simulator being used does not support fast forwarding, such as with a Verilog model, the overall speedup would be orders of magnitude better since, as was demonstrated in Figure 6, the intrinsically checkpointed binary has drastically fewer instructions to execute prior to the interval. For example, if the runtime of each checkpointed binary is used as a rough estimate for the amount of time it would take the simulator to simulate in detail 100 million instructions of the benchmark, then the non-checkpointed versions of gcc and fma3d would take 2.7 hours and 2.7 days, respectively, to finish the execution of the SimPoint interval. The average time to finish the SimPoint interval for a benchmark if fast forwarding is not available is 1.5 days and the twolf benchmark would take a maximum of 5.9 days.

Table 2. Increase in file size (measured in bytes) for specINT, specFP, and spec2000

	Initial	dowrites	nowrites
SpecInt Avg	1,120,256	8,551,333	8,075,642
SpecFP Avg	1,506,583	26,719,511	26,502,684
Spec2K Avg	1,343,919	19,069,751	18,743,930

5.4 Binary file size increase

For the final set of results, Figure 8 and Table 2 show the increase in size measured in bytes of the benchmarking binary when the intrinsic checkpointing code is inserted. Due to the fact that the memory restoration requires 64-bit data both for the values and addresses, it is expected that there will be a fairly significant increase in size since 64-bit values and 64-bit addresses that are not close to each other in the address space must each occupy 8 bytes of code in the data section. In addition, highly data intensive benchmarks such as the scientific computation based floating point benchmarks of SPEC2000 will show a much larger increase in code size, approximately 25 megabytes on average, since it will take more to set up the memory that they will be using for their computations. However, for the integer benchmarks, the increase in size is rather modest, roughly 7 megabytes on average, compared to other forms of checkpointing which checkpoint the entire system state and it is always possible to further decrease the final code size by decreasing the size of the simulation interval.

6. Similar work – SimSnap

The above method of adding intrinsic support for checkpointing into a benchmark binary with the goal of reducing simulation time is still a relatively new area of research. As an example, the newly proposed SimSnap tool [7] from Cornell University is an alternate approach to the one proposed above. Although its goals are the same, the method with which the binary is checkpointed occurs when the benchmark is compiled instead of when it is simulated. The user has to modify the original source code of the benchmark to include checkpointing routines and also has to supply, on a function level, a location for where the code should checkpoint itself. After the benchmark is compiled, it is run and at the predefined location in the code it will output checkpointing data to a separate file. Subsequent runs of the benchmark, when executed in a restore mode, will read in the data from the checkpoint file and resume operation where the benchmark left off.

Because of the fact that SimSnap relies heavily on the compiler and has to checkpoint not only the data required for the proper execution of the simulation interval, but the entire system state, it has several drawbacks when compared to the proposed method above. The first is that the checkpointing files can be quite large. In the published results of SimSnap, a typical checkpoint file is on the order of 50 megabytes. When compared to the above method that adds and average of 17 megabytes to the original binary, this amount of checkpoint data can be significantly larger. The second drawback is that SimSnap not only requires the user to make changes to the source code, but also that the source code itself be available. For benchmarks that do not supply source code, this will prevent them from utilizing the intrinsic checkpointing features of SimSnap. The final drawback is that SimSnap will only work if the programming language the benchmark is written in is supported by their compiler. If the program was written in a propriety language or one that is no longer heavily used, such as FORTRAN, then SimSnap would not be able to generate checkpoint code. The method for intrinsic checkpointing in this paper has none of these drawbacks. It doesn't require recompilation, supports any programming language, does not require source code modification, and leads to smaller checkpointing data.

7. Conclusions and Future Work

In this paper, we have presented a methodology that dramatically decreases the simulation time of a benchmarking binary by removing the need to fast forward through massive amounts of instructions without relying on the simulator to explicitly checkpoint the code. By analyzing the code of a desired simulation interval, our method generates a portion of checkpointing code meant to replicate the outcome of the instructions that were executed prior to the interval. This checkpointing code is then inserted at the start of the original binary. When the checkpoint code finishes its execution, control is transferred to the beginning of the simulation interval and execution can continue until the end of the interval as before. Results of this method show an average speedup for SPEC2000 of 60x for a SimPoint interval of 100 million instructions with a max speedup of 180x for fma3d. This corresponds to a decrease in simulation time of the SimPoint interval from over 3 hours to 3 minutes on average and a decrease from 8.9 hours to 3 minutes for fma3d. The increase in file size of the checkpointed binary is also kept to a reasonable 17 megabytes on average.

For our future work, we plan on studying methods to further decrease the file size of the checkpointed binaries. Some ideas that we are considering are combining contiguous byte stores into larger store types such as word stores, aggressive data packing, using looping to reduce code size, guaranteeing only unique values in the .data section of the checkpoint code, and possibly data compression. Another area of future study is creating a method for removing syscalls from the binary altogether and instead generating checkpoint code that emulates their effects. For example, each syscall would be converted into a jump to a portion of code that updates any registers or memory locations that the syscall originally would have modified. A final area of future work is in studying the feasibility of creating a single benchmark that consists of many small, intrinsically checkpointed benchmarks. This "pseudo-synthetic" benchmark could represent an entire suite of applications in one individual package and help to further decrease the amount of time needed for simulation.

8. References

[1] A.J. KleinOsowski, J. Flynn, N. Meares, D. J. Lilja, "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research", *Workshop on Workload Characterization, International Conference on Computer Design,* Austin, TX, September 2000.

[2] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Designing Computer Architecture Research Workloads – MinneSPEC", *Computer*, Volume 36, Issue 2, February 2003, pp. 65-71.

[3] R.E. Wunderlich , T.F. Wenisch, B. Falsafi, and J.C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", *Proceedings of the 30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[4] E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points", *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, LA, October 2003.

[5] G. Hamerly, E. Perelman, and B. Calder, "How to Use SimPoint to Pick Simulation Points", *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[6] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior", *Proc. of 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, October 2002.

[7] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz, "SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing", *The 8th Workshop on the Interaction between Compilers and Computer Architectures*, Madrid, Spain, February 2004.

[8] D. A. Wood, M. D. Hill, and R. E. Kessler, "A Model for Estimating Trace-Sample Miss Ratios", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, May 1991.

[9] R. E. Kessler, M. D. Hill, and D. A. Wood, "A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches", *IEEE Transactions on Computers*, Volume 43, Number 6, June 1994, pp. 664-675.

[10] T. M. Conte, M. A. Hirsch, and W.W. Hwu, "Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation", *IEEE Transactions on Computers*, Volume 47, Number 6, June 1998, pp. 714–720.

[11] J. Haskins and K. Skadron, "Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation", *Proceedings of the 2003 International Symposium on Performance Analysis of Systems and Software*, Austin, TX, March 2003.

[12] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium", *IEEE Computer*, Volume 33, Number 7, July 2000, pp. 28-35.

[13] SimpleScalar, LLC. http://www.simplescalar.com/.

[14] Compaq Computer Corporation, Alpha 21264 Microprocessor Hardware Reference Manual, July 1999.