TECHNIQUES FOR EFFICIENTLY RECORDING STATE CHANGES OF A COMPUTER ENVIRONMENT TO SUPPORT REVERSIBLE DEBUGGING

BY

STEVEN ALLEN LEWIS II

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

ACKNOWLEDGMENTS

During the summer of 2000, Professor Dave Small suggested to me that the CISE department of the University of Florida would have interest in a modernized MIPS simulator. I would like to thank Dave Small for encouraging me to pursue such a project, which ultimately played a significant role for this thesis.

Following a brief discussion later in that year, Professor Small suggested that I contact Dr. Michael Frank, whose research includes many aspects of reversible computing. Indeed, Dr. Frank was instrumental in leading me to existing research regarding reversible debugging and reversible program execution. I would very much like to thank Michael Frank for his guidance, his patience, and his assistance in refining the techniques discussed in this thesis. Further, I would like to thank Dr. Frank for his role as my committee chairperson.

I would also like to thank Dr. Randy Chow and Dr. Manuel Bermudez for serving as committee members for this project. Their support for this thesis came indirectly from the courses that they taught, specifically the compiler courses by Dr. Bermudez and the operating system courses by Dr. Chow. I learned a great deal from these courses, including the following two concepts that I have found to be particularly useful: the use of context free grammars and the layered approach to software development.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

TECHNIQUES FOR EFFICIENTLY RECORDING STATE CHANGES OF A COMPUTER ENVIRONMENT TO SUPPORT REVERSIBLE DEBUGGING

By

Steven Allen Lewis II

August 2001

Chairman: Dr. Michael P. Frank
Major Department: Computer and Information Science and Engineering

The notion of a truly reversible computer offers many potential applications. One such application is reversible debugging, also known as bi-directional debugging.

Modern debuggers allow one to pause the execution of a program, generally for the purpose of trying to identify an error within the program. In the early stages of attempting to locate the cause of an error, the debugging process would be more intuitive if one could undo the effects of the program on the state of the environment. That is, execute the program in reverse.

The potential benefit of reversible debugging is that less time is spent on debugging, which is a process that is normally overly tedious due to the inefficient manner in which it is performed. Further, some programs, such as those that are non-deterministic and operate in real time embedded environments, cannot be debugged by traditional means. The use of state change recording, or history logging, is one means by which such programs could be debugged.

This thesis describes a set of techniques that, when used together, provide an efficient means of recording state changes. A case study is included that demonstrates the use of these techniques within a fully functional MIPS simulator implemented in Java. However, the techniques described are hardware and language independent.

These techniques rely on both state change recording and program re-execution. State changes are recorded incrementally during each cycle of execution. Periodically, the current set of state changes is accumulated into a checkpoint structure. A stack of checkpoints is maintained, allowing the executing program to be reversed indefinitely.

The primary concern of such history logging techniques is the unbounded growth of history information. A checkpoint culling algorithm is used to bound the size of the state history to a logarithmic function, as well as maintain a linear run time for undo operations.

CHAPTER 1
INTRODUCTION

This thesis is concerned with techniques regarding how state changes in a computer environment can be recorded efficiently. A case study for the use of these techniques is mentioned throughout the discussions. The study involves implementing a MIPS simulator (called JIMS) that records state changes, for the purpose of allowing simulated programs to execute in reverse (i.e. undoing the effects of the program to the state of the environment). The ability to execute programs in reverse is extremely useful [1-3], particularly in the context of debugging [4-6].

Reversible Debugging Concept

When an unexpected state is encountered during the execution of a program, the process of debugging involves three general steps: (1) locating the cause of the error, (2) correcting the error, and (3) testing to ensure that the error is in fact removed and that new errors have not been introduced. Depending on the complexity of the program and the nature of the error, the debugging process can be extremely tedious. The act of locating the error, or rather locating the cause of the error, can be particularly time consuming. Most debugging tools have the ability to pause the execution of a program (using breakpoints and special trap instructions). While a program is paused, one can examine the state of the environment and determine if that state is as expected (i.e. if an error has occurred yet or not).

Typically the execution of a program is stopped (by a breakpoint, for instance) at some point after an error has been encountered during execution. However, often the cause of an error precedes the manifestation of that error. Consequently, the general accepted practice of debugging is to mentally examine the logic of the program source code, specify a new breakpoint at some earlier statement of the program, and then re-execute the program from the beginning. These steps are repeated until the cause of the error is realized.

The aforementioned debugging process is well suited for small programs that are well understood and deterministic. However, there are two primary drawbacks to this process: (1) Deciding an appropriate position of a breakpoint is often a matter of speculation. The first breakpoint that is set is almost always after the occurrence of an error becomes apparent. However, there is no guarantee that a newly chosen breakpoint will lead to being closer to the cause of the error. For instance, a breakpoint might be set too early, thus requiring the program to be slowly executed forward such as not to inadvertently overstep the error. (2) Re-executing a program from the beginning, for the purpose of stopping the program at a point that is hopefully closer to the cause of the error, is inefficient. The runtime of a program, up until the first breakpoint is encountered, could be very long. In addition, if the program is non-deterministic, returning to the state in which the error was encountered might not be easily achieved.

For some programs (such as deterministic programs that have a fast runtime), program re-execution can be used to emulate reversible debugging by automating the traditional debugging process [7]. For more complex and non-deterministic programs, an alternative approach is necessary. Such an alternative would include recording changes to the state of the environment over time. The primary drawback of state change recording, or *history logging*, is that a great deal of information is involved. Moreover, in order to offer practical performance, this information must be readily available, such as by being stored in main memory or some readily accessible storage facility. However, state change recording (of some form) is necessary to fully support reversible debugging of non-deterministic programs.

Other approaches to supporting reversible execution include a reversible instruction set [8] or program inversion algorithms [9]. These approaches are highly promising, though they tend to require fundamental changes to existing hardware and software systems.

Overview of Thesis Content

The content of this thesis describes techniques that can be employed to implement instruction level reversibility of a computer system, using state change recording. The JIMS project demonstrates these techniques in a useful and fully functional application. However, JIMS is not the focus of this thesis. Instead, the intention of this thesis is to describe the design of, and discuss issues related to, techniques for efficient state history recording.

No attempt is made to justify that state history recording is a superior means of accomplishing the application of reversible debugging. Nor are the techniques that are described considered to be the most efficient. However, my opinion is that the techniques described herein are highly general, being hardware and programming language independent. Moreover, these techniques can be applied by augmenting existing hardware or software systems.

Should one need instruction level reversibility, this thesis describes a possible means of satisfying such a need and at least some of the associated technical issues. After having implemented JIMS, the experience suggests that many aspects of reversibility are a compromise between computational complexity and memory space usage. As I see it, the problem of supporting reversibility is analogous to modeling the human brain's ability to remember a considerable amount of information over a very long time. Undoubtedly, we would benefit from further research in both areas.

Outline of Thesis

Chapter 2 contains a verbose example that demonstrates what is meant by a *state change*, and what is necessary in order to record a state change. The example also includes the use of checkpoints, discusses the association between state change records and checkpoints (within the context of this thesis), and discusses an efficient means of representing state history.

Chapter 3 discusses the task of how to create a checkpoint efficiently, which involves identifying only those state variables that have been modified. Chapter 4 describes several techniques for determining when a checkpoint should be created.

Chapter 5 addresses the issue of unbounded growth of state history information. This chapter describes a checkpoint culling process that maintains a logarithmic growth of recorded state history space usage.

Chapter 6 concludes this thesis by demonstrating the growth of state history when using these techniques, and commenting on related issues not mentioned in the previous chapters.

Appendix A contains information regarding the JIMS project.

CHAPTER 2
COMPACT REPRESENTATION OF STATE HISTORY

For the purpose of this thesis, the term *state history* refers to the entire structure used to hold all state change information. State history should be represented as a vector to which new entries can be appended and existing entries can be deleted. It is generally not necessary to insert entries into the state history. The state history vector consists of two types of entries: (1) *checkpoints* and (2) *state change records*. Checkpoints record the entire modified state of the machine at a particular point in time. State change records store only incremental changes to the state. When a checkpoint is created, essentially all the information stored in the current set of state change records is compacted into a single checkpoint. As a result, checkpoints are always at the beginning of the state history, followed by the current set of state change records. Figure 2.1 shows the general structure of the state history model.

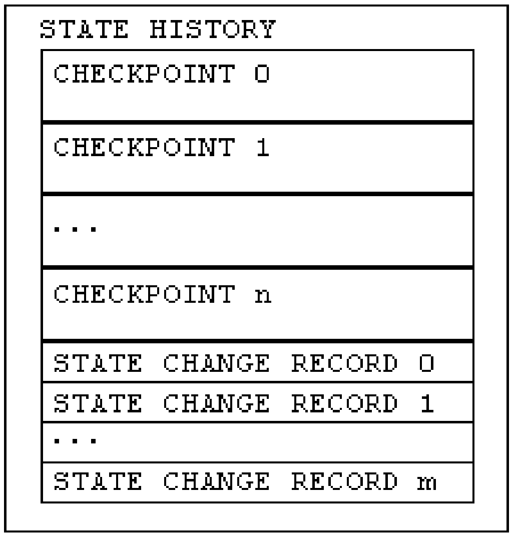


Figure 2.1: State History Model

The remainder of this chapter discusses, by means of an example, how both checkpoints and state change records can be efficiently represented in a data stream.

State Change Example

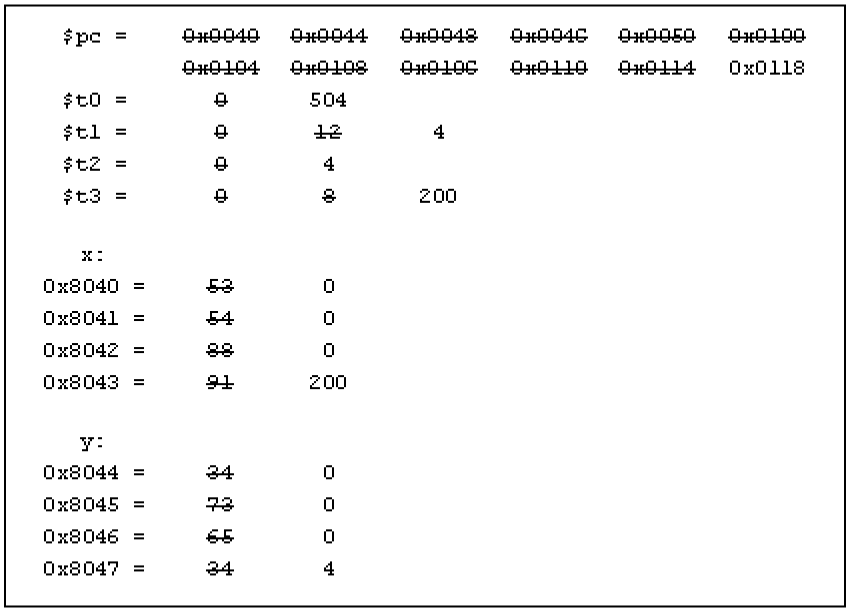
Consider the hypothetical MIPS program shown in Figure 2.2. While this program does nothing special in particular, the instructions used are a representative combination of load, store, branch, and arithmetic operations. The hexadecimal values on the left represent a supposed address associated with each instruction and data entry.

```
                .text
0x0040          li      $t2, 4
0x0044          sw      $t2, y
0x0048          li      $t3, 8
0x004C          j       main
0x0100  main:   addi    $t0, $t2, 500
0x0104          or      $t1, $t2, $t3
0x0108          beq     $t1, $t3, eq
0x010C          li      $t3, 200
0x0110  eq:     sw      $t3, x
0x0114          lw      $t1, y
0x0118          ...
                .data
0x8040  x:      .word   0x53548891
0x8044  y:      .word   0x34736534
```

Figure 2.2: Sample MIPS Assembly Program

Although tedious, it is instructive to manually demonstrate how the state changes of this program would be recorded. By examining this short program thoroughly, one can gain a better understanding of the general pattern of information that is changed during each cycle of program execution.

Let us first consider executing the sample program without recording state changes, but rather just examining the effect that the program has on the state. Assume the program begins at address `0x0040`. Figure 2.3 shows the state changes that occur during the first ten cycles of execution. The crossed out values represent old information that was replaced by some new value. The values that are not crossed out represent the current value of the corresponding state variable. Notice that the Program Counter was changed during each cycle. In fact, there is an instance where the Program Counter is modified twice during the same cycle.



$pc =	~~0x0040~~	~~0x0044~~	~~0x0048~~	~~0x004C~~	~~0x0050~~	~~0x0100~~
	~~0x0104~~	~~0x0108~~	~~0x010C~~	~~0x0110~~	~~0x0114~~	0x0118
$t0 =	~~0~~	504				
$t1 =	~~0~~	~~12~~	4			
$t2 =	~~0~~	4				
$t3 =	~~0~~	~~8~~	200			
x:						
0x8040 =	~~53~~	0				
0x8041 =	~~54~~	0				
0x8042 =	~~88~~	0				
0x8043 =	~~91~~	200				
y:						
0x8044 =	~~34~~	0				
0x8045 =	~~73~~	0				
0x8046 =	~~65~~	0				
0x8047 =	~~34~~	4				

Figure 2.3: State Changes of Sample Program

By examining Figure 2.3, it is evident that in order to record the state change history of this program, we simply need to remember all of the crossed out values. However, we must not forget one particularly important limitation: when constructing this figure, as new values were entered, the corresponding old value was crossed out and the new value was written in the space available to the right. If we had continued to do this, eventually we would have run out of space in the diagram. If this had happened, we could have just used a larger figure. The point is, however, that when recording state changes in this fashion, it is inevitable that we would run out of space to store all of the past values of each state variable.

In an environment that records state changes, it is important to do so in an efficient manner due to physically limited storage capacity. This means recording information promptly while using no more storage space than necessary. We will now consider executing this program and efficiently recording the changes made to the state of the environment.

State History Stack

Suppose it were possible to maintain a stack associated with each state variable (e.g. each register, memory cell, etc.). Each stack would record any change to its respective state variable. That is, before a new value is applied to any state variable, its current value is pushed onto the corresponding stack of that variable. To perform an undo operation, one would pop the top value from the Program Counter, and then simply undo the effect of the instruction at the address pointed to by the Program Counter. As a generic example, if an `ADD` instruction was issued that wrote its result to register `$t0`, then to undo its effect we need only to pop the top value of the `$t0` stack and apply that value to register `$t0`.

Ideally, this technique would work. Though, in reality, the cost of maintaining a stack for each state variable is prohibitive. However, with this insight in mind, the remainder of this chapter discusses an alternative approach to recording state history information.

In the state history stack approach, only one stack is needed. When a state change is made, two pieces of information are pushed onto the stack: (1) an indication of what state variable was modified, (2) the pre-modified value of the state variable. For example, suppose an `XOR` instruction was issued that wrote its result to register `$t7`. Before storing the result into register `$t7`, and assuming that the current value of `$t7` is `5`, we would (for example) push onto the state history stack "`REG $t7 5`" (the actual encoding is discussed later in this discussion, though for now we can assume a literal sequence of ASCII characters or tokens is used). If an undo operation was then requested, we simply pop from the state history stack, and see that register `$t7` should be restored to the value `5`. The records pushed onto the state history stack should indicate the original value of each state variable modified during the corresponding cycle. Keep in mind that during each cycle of execution, multiple state variables might be modified.

Before discussing how state change information is actually encoded, it is important to first clarify the relationship between checkpoints and state change records. The next two sections of this chapter describe this relationship.

Checkpoints

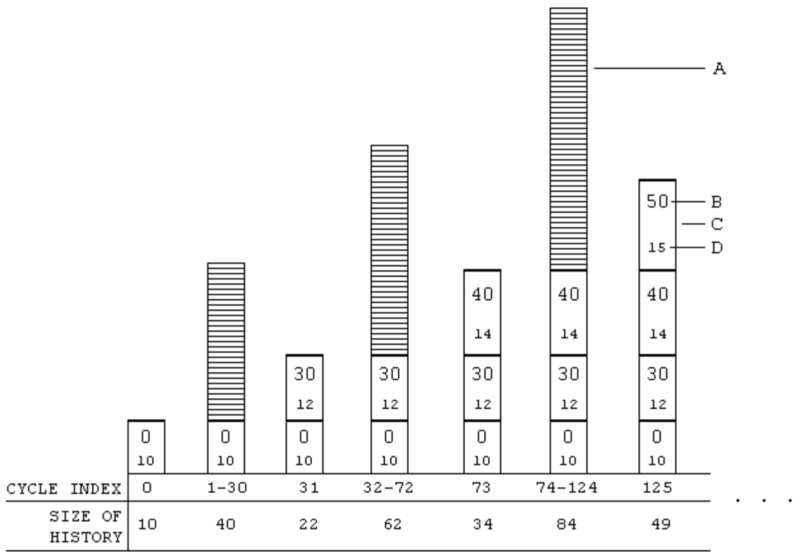
The growth of the state history stack, as described above, is obviously unbounded. However, we can take advantage of the manner in which computers execute programs. When a certain condition is met (refer to Chapter 4), we can construct a checkpoint that records the value of all modified state variables at that time. Then we can replace the entire contents of the state history stack with this checkpoint.

Furthermore, we can maintain a stack of checkpoints. In doing so, if one ever desires to return to a state before the creation of the last checkpoint, we can set the state to that of an earlier checkpoint and re-execute the program until the desired execution point is reached. While there is a performance penalty (among other issues) in performing re-execution, the benefit of using checkpoints is that some space is conserved and the amount of time spent for re-execution is reduced.

Checkpoints are often used in database systems and operating systems, typically for data recovery reasons [10]. However, checkpoints have numerous other applications [1], particularly for reverse execution of parallel programs [11].

Analysis of State History Growth

Figure 2.4 and Figure 2.5 show the growth and reduction of the state history, during forward and reverse program execution respectively. The numerous smaller blocks represent individual state change records (indicate by label A in Figure 2.4), while the larger blocks represent one checkpoint (C). Within each checkpoint block, the larger sized number represents the corresponding time index, such as a cycle count (B), while the smaller sized number represents the relative size of the checkpoint (D).



```
A = State Change Record Structure
B = Time Index of Checkpoint
C = Checkpoint Structure
D = Relative Size of Checkpoint
```

Figure 2.4: State History Growth During Forward Execution

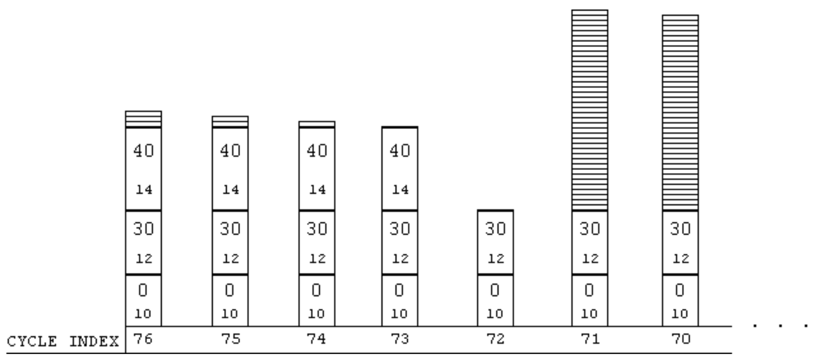


Figure 2.5: State History Reduction During Reverse Execution

Figure 2.6 shows the growth rate of state history in a graph, comparing the use of checkpoints versus the non-use of checkpoints (i.e. use of state change records only). The data used is that of Figure 2.4. Notice that without checkpoints, the growth rate is linear. With checkpoints, the rate of growth is still linear (as emphasized by the dashed line near the center of the graph), but at a much slower rate.

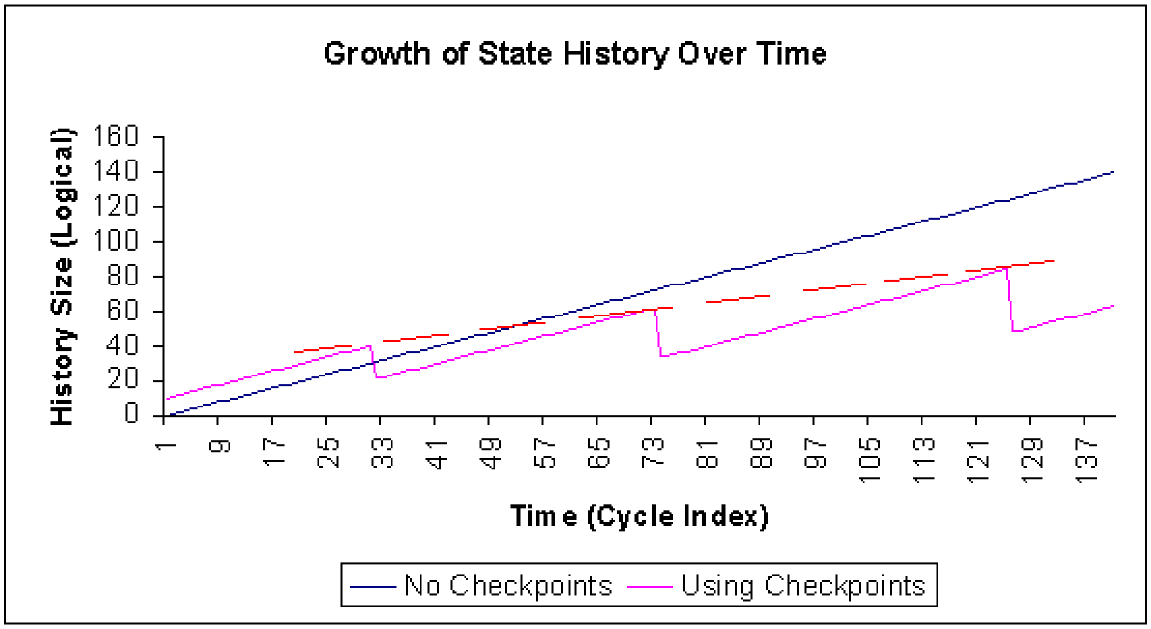


Figure 2.6: Growth of State History Over Time

The details regarding when to create checkpoints and issues related to re-execution are discussed in later chapters. For now, we are concerned with how to efficiently represent both incremental state change records and checkpoints.

Data Representation of State History

Returning to the example discussed earlier in this chapter (see Figure 2.2), when the first instruction of the sample program is executed, a checkpoint is immediately created. This first checkpoint represents the *initial state* of the environment. It is worth mentioning that anytime a checkpoint is created, it is not necessary to record the state changes of the current instruction. When such an instruction is executed, the checkpoint just created implicitly contains the necessary undo information.

The purpose of a checkpoint is to record the current state of the environment at a particular point in time. To accomplish this, a checkpoint contains three pieces of information: (1) The current time index, which is used to indicate how far back in time the checkpoint represents. This is usually a simple integer value that is always positive and increasing, such as a cycle counter. (2) The current Program Counter. This information is necessary in order to know from what address the program should resume from, should this checkpoint be applied. (3) The value of all currently modified state variables. Obviously, this last piece of information is (potentially) a large amount of data. Effort must be made to ensure that this information is recorded as optimally and as quickly as possible.

The initial state checkpoint for the example program would be recorded as shown in Figure 2.7 (again, how this information would actually be encoded is discussed later in this discussion).

```
CHKPNT @ 0
PC = 0x0040
0x0040 [li] [sw] [li] [j]
0x0100 [addi] [or] [beq] [li] [sw] [lw] […]
0x8040 53 54 88 91 34 73 65 34
```

Figure 2.7: Contents of Example Initial State Checkpoint

The first two lines in Figure 2.7 represent the first two pieces of necessary checkpoint information just discussed. That is, the first line indicates that this checkpoint was recorded at time index zero. The second line records the Program Counter for this particular checkpoint. The remaining lines represent the modified state information, caused by the fact that a program has been loaded. Notice that there are MIPS mnemonics enclosed in brackets (e.g. "`[li]`"). This is a shorthand notation for the purpose of this example. In reality, each such enclosed instruction mnemonic is represented by some bit sequence (i.e. each MIPS instruction is a sequence of 32-bits).

Although no instructions have yet been executed, the state was modified when the program was loaded to hold the programs text and data. However, this information does not necessarily need to be recorded. In fact, one can readily return to this state simply by resetting the environment and reloading the program (thus the entire checkpoint is not necessary). Alternatively, since self-modifying code is rare, one could choose to record only the program data and not the text (since program data is often modified, unlike the text). On the other hand, creating a checkpoint to record the entire initial state can have two benefits: (1) It can maintain consistency and ease implementation, (2) It can save time by not having to reload the program when returning to the initial state.

Generally speaking, it would probably be more optimal not to use a checkpoint to record the initial state. But a more important issue is how the checkpoint information is represented and stored. Although it is desirable to store information in a format that is as compact as possible, it is also necessary to ensure that such encoding does not require too much time. For instance, compressing the data representation of this checkpoint might incur adverse computational overhead [12]. Although checkpoints should only be created periodically, creating checkpoints is inherently computationally expensive. This is because it is necessary to determine what state information to store, in addition to actually retrieving said information. The issues regarding checkpoint creation are discussed in Chapter 3.

For this example, we will assume that the checkpoint for the initial state is created. On the next cycle of execution, we need only to record the state changes made as a result of executing the instruction at that address, which is "`sw $t2, y`." This instruction stores the contents of the `$t2` register into the word address labeled `y` (that is, `mem[y] = reg[$t2]`). The only state change, resulting from the executed instruction itself, is that the content of the word address starting at `0x8044` (the address associated with label *y*) is changed to a `4`. However, other state changes may have also occurred. Internal CPU state variables or flags may have been changed, which may be pertinent for the purpose of undoing the effect of the instruction. More obvious, the Program Counter is changed during each cycle. Typically the program counter is simply incremented by a constant value, but it may have also been changed by a branch or jump instruction. The state changes for this cycle can be recorded as shown in Figure 2.8.

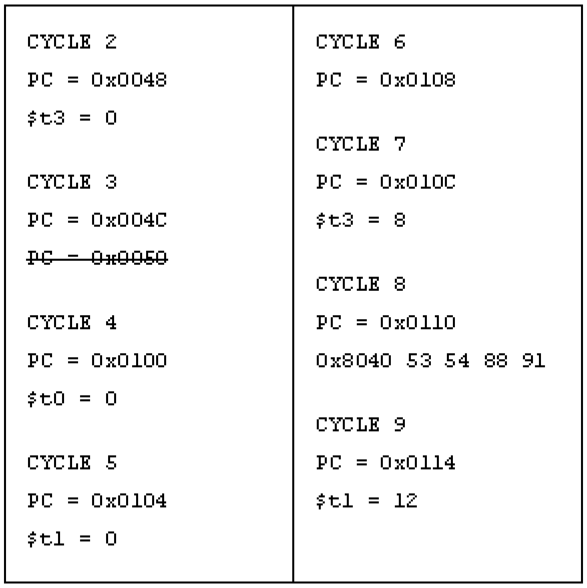
```
CYCLE 1
PC = 0x0044
0x8044 34 73 65 34
```

Figure 2.8: Contents of Example First State Change Record

The amount of information recorded in a state change record is much less than that of a checkpoint. Notice, however, that three general pieces of information are recorded: the time index, the Program Counter, and modified state values. This is the same general pattern as the checkpoint created earlier. The modified state value recorded is the content of the word at `0x8044` before the `SW` instruction applies the new value.

When storing this state change record, there are two general optimizations that can be applied: (1) It might not be necessary to record the time index. If these state change records are stored in a vector, such as an array, then the index of the record into that vector can be used to indicate the time index. This may require another variable to hold a base time index value, which would most likely be the time index of the most recently created checkpoint. In this case, the time index for any arbitrary state change record `R[I]` would be `BASE_TIME_INDEX + I`. (2) Since the Program Counter is typically incremented by a constant value, one could store a flag that this is the case rather than store the entire Program Counter. However, there may be some overhead in determining if the Program Counter was simply incremented or has a new value entirely. Certainly one should not assume that the Program Counter is always incremented by a constant value, as quite frequently this is not the case (such as during a branch or jump).

The state change records for the remaining eight cycles of execution for this example program are shown in Figure 2.9.



CYCLE 2
PC = 0x0048
$t3 = 0

CYCLE 3
PC = 0x004C
~~PC = 0x0050~~

CYCLE 4
PC = 0x0100
$t0 = 0

CYCLE 5
PC = 0x0104
$t1 = 0

CYCLE 6
PC = 0x0108

CYCLE 7
PC = 0x010C
$t3 = 8

CYCLE 8
PC = 0x0110
0x8040 53 54 88 91

CYCLE 9
PC = 0x0114
$t1 = 12

Figure 2.9: Remaining State Change Records for Example Program

In Figure 2.9, one state change record of interest is that of `CYCLE 3`. Notice that the Program Counter is modified twice during this cycle (once during the Instruction Fetch stage, then again during the execution of the Jump instruction). If the same state variable is modified more than once in the same cycle, only the first instance needs to be recorded. Or alternatively, when performing an undo operation, state changes should be applied in the reverse order they were recorded in. In doing so, even if a particular state variable is modified more than once in the same cycle, the earliest value will always be the final value applied. This means that it is not necessary to use computation time to remove duplicate entries. Of course, for the sake of optimizing space usage, one should consider taking the time to ensure that such non-useful information is not recorded.

At this point, it might be a good idea to consider the creation of another checkpoint. A variety of heuristics can be used to determine when a checkpoint should be made. This is discussed in greater detail in Chapter 4. For now, it would be instructive to examine what would happen if indeed a checkpoint were created at this point in the example programs execution.

First, however, aside from actually deciding if a checkpoint should be created, a minor issue is whether this decision should take place at the end of a cycle or at the beginning. That is, when a cycle begins, should we decide on creating a checkpoint (and then do so) before executing the instruction or afterwards? From an abstract point of view, there should be no difference. This is strictly an implementation detail.

For JIMS, it was decided that all checkpoint handling would take place near the beginning of a cycle (before the execution of the instruction). The reason is because of the following logic: if it is decided that a checkpoint should be created, the checkpoint is created immediately and state history recording is disabled for that cycle. Otherwise, if a checkpoint is not created, a state history buffer is created for the duration of the cycle, and then added to the state history records after execution (near the end of the cycle). The idea is that disabling history recording when it is not necessary can (during execution) conserve space and improves performance. If a checkpoint were created after the instruction was executed, then the state changes recorded during the cycle are redundant since the creation of the checkpoint contains the same information.

Returning to the example, Figure 2.10 shows the representation of the checkpoint that would be created at the beginning of cycle ten of the sample program.

```
CHKPNT @ 10
PC = 0x0118
0x0040 [li] [sw] [li] [j]
0x0100 [addi] [or] [beq] [li] [sw] [lw] […]
0x8040 00 00 00 C8 00 00 00 04
$t0 = 504
$t1 = 4
$t3 = 200
```

Figure 2.10: Second Checkpoint for Sample Program

This checkpoint represents the accumulation of state changes made during the last nine cycles. As such, the current collection of state history records is discarded, since they can be reconstructed by re-executing from the previous (in this case, initial state) checkpoint. If execution continues, this sequence of events is repeated: the state change records reach a certain space threshold, then are replaced with a checkpoint.

Alternatively, one could maintain a dedicated buffer for state change records. Then, rather than deleting all state change records after a checkpoint is created, any new state change record can instead replace the oldest state change record still in the buffer. That is, the state change record buffer is circular. This allows small undo operations issued just after the creation of a checkpoint to be performed using state change records, rather than re-execution from a previous checkpoint.

There are two important points to examine regarding this new checkpoint: (1) It is slightly larger than the previous checkpoint. This should be expected, since more state variables have been modified since the last checkpoint was created. The size of a checkpoint, or rather the potential size of a checkpoint, should be a factor in deciding when to create checkpoints. (2) The new checkpoint contains some identical information already available in the last checkpoint. It is obvious that it would be more optimal to store only the modified state values that differ from the previous checkpoint.

If only the state changes that differ from the last checkpoint had been recorded, then the new checkpoint would not contain the modified word values at addresses `0x0040` and `0x0100` (since it would be redundant to do so). The problem, however, is it is computationally expensive to detect redundant checkpoint data. A significant amount of scanning and comparing values would be necessary.

Of course, one could apply simple techniques to remove obvious redundancy. For instance, as mentioned earlier, self modify code is not the norm. Therefore, it is probably not necessary to store program text in the state history information, or at least not in every checkpoint. However, suppose it were in fact easily possible to identify and record only those state changes whose value differs from that stored in the previous checkpoint (a means for doing so is briefly discussed in the next chapter). This would change the requirement for performing the undo operation. If an undo request required the use of a checkpoint, then the process would need to start with the initial state checkpoint, and apply each checkpoint thereafter in sequence (until a checkpoint with a suitable time index was encountered), then re-execute from that point towards the desired time index.

There are two arguments for simply recording all modified state values during the creation of each checkpoint: (1) To maintains consistency, which benefits design and implementations. (2) When running a program with a long run time, one can cull out old checkpoints as necessary. Not because these checkpoints are not useful, but because the state recorded in these checkpoints can be re-constructed by re-execution from earlier checkpoints. The culling process, which is discussed in Chapter 5, is intended to prevent the size of the recorded history information from becoming exceedingly large.

Having examined the example described earlier in this discussion, we can formally specify a language that describes the state history. Using the notation of regular expressions, Figure 2.11 shows a context-free grammar that could be used to describe the syntax of recorded state history information.

```
StateHistory      -> Checkpoint* StateChangeRecord*
Checkpoint        -> ‘c’ TimeIndex ProgramCounter State*
                  ->
StateChangeRecord -> ‘s’ TimeIndex? ProgramCounterSCR State*
                  ->
TimeIndex         -> ‘<word>’
ProgramCounter    -> ‘<word>’
ProgramCounterSCR -> ‘i’       // Indicates PC is incremented
                  -> ‘a’ ‘<word>’  // Indicates new PC address
State             -> ‘r’ WhichReg RegValue
                  -> ‘m’ Address Value* ‘e’
                  -> ‘f’ WhichFlag FlagValue
                  ->
WhichReg          -> ‘<byte>’
RegValue          -> ‘<word>’
Address           -> ‘<word>’
Value             -> ‘<byte>’
WhichFlag         -> ‘<byte>’
FlagValue         -> ‘<byte>’
```

Figure 2.11: Sample Context Free Grammar for State History Recording

By using the grammar described in Figure 2.11, the following sequence would be used to record the initial state checkpoint used in the earlier example (this sequence would be stored in some pre-allocated byte array or data stream):

```
CHECKPOINT 1:     c 0 0x0040
                  m 0x0040 [li] [sw] [li] [j] e
                  m 0x0100 [addi] [or] [beq] [li] [sw] [lw] e
                  m 0x8040 53 54 88 91 34 73 65 34 e
```

Assuming that all of the integer tokens are 32-bit (requiring four bytes each), and each non-integer token (i.e. `c`, `m`, `e`) and memory value consumes only one byte, then this sequence requires 75 bytes of storage. The nine state change records that follow would be encoded as follows:

```
CYCLE 2:            s 1 i m 0x8044 34 73 65 34 e
CYCLE 3:            s 2 i r $t3 0
CYCLE 4:            s 3 i
CYCLE 5:            s 4 a 0x0100 r $t0 0
CYCLE 6:            s 5 i r $t1 0
CYCLE 7:            s 6 i
CYCLE 8:            s 7 i r $t3 8
CYCLE 9:            s 8 i m 0x8040 53 54 88 91 e
CYCLE 10:           s 9 i r $t1 12
```

Using the same assumptions used with the checkpoint sequence, and also assuming that the register index (`WhichReg`) is encoded using only one byte, these state change records altogether use 108 bytes of storage (averaging 12 bytes per cycle). Notice that the recorded time index is simply incremented during each cycle. As mentioned earlier, some optimization can be done by not storing the time index for each state change record (though it is necessary to store a time index with each checkpoint). Doing so would reduce this sequence to 72 bytes (an average of eight bytes per cycle, a 33% improvement). Also, the use of the `s` token appears redundant, however it is necessary in order to mark the beginning of each state change record.

For completeness, the second checkpoint created would be recorded by the following sequence:

```
CHECKPOINT 2:       c 10 0x0118
                    m 0x0040 [li] [sw] [li] [j] e
                    m 0x0100 [addi] [or] [beq] [li] [sw] [lw] e
                    m 0x8040 0 0 0 200 0 0 0 4 e
                    r $t0 504
                    r $t1 4
                    r $t3 500
```

With the same assumptions as before, this would use 93 bytes of storage. Notice that this is smaller than the combined size of the state change records (i.e. `93 < 108`). Since this checkpoint replaces the current set of state change records, creating this checkpoint conserves some space. Also notice that the redundant data in this checkpoint has not been removed. In particular, the redundant data is the text of the loaded program.

The final issue is where the state history information should be stored. If the CPU itself recorded state history, the information could be stored in a cache dedicated to this purpose. This would facilitate the debugging of embedded system level software and operating systems. If a critical or unexpected error was encountered, an exception would incur and state history recording could be stopped. Within the exception handler, the developer could not only examine the current value of registers and other state variables, but also revert to previous instances of the state (to pinpoint the cause of the problem).

At the operating system level or application level, state history could be buffered at a designated region of main memory or a disk. Storing the entire state history to disk would probably not be practical, for performance reasons. However, it might be a reasonable compromise to store checkpoints to disks rather than in main memory, and keep state change records in memory. One interesting prospect is that the operating system could record a separate state history for each process. This would certainly lead the way to more effective debugging of multithreaded programs, distributed systems, or parallel processing environment.

CHAPTER 3
CHECKPOINT CREATION PROCESS

Each cycle of execution most likely results in some change to the state of the environment. At the very least, for example, the Program Counter is changed during each execution cycle. Recording these changes to the state as efficiently as possible is highly important, and a technique for doing so is discussed in the previous chapter.

Modern memory systems are finite, and so it is inevitable that storing state history changes every cycle will consume all (or a large portion) of the available memory space. Checkpoints are used to reduce the rate of growth of state history records. However, creating checkpoints introduces computational overhead in two ways: (1) Deciding when to create a checkpoint. This issue is discussed in the next chapter. (2) The actual process of recording all of the modified state variables, which is discussed in this chapter.

For a modern CPU, examples of state variables include register values, contents of main memory, internal flags, contents of cache memory, etc. If the environment has no cache or main memory, it might be suitable for each checkpoint to simply record all of the register values and any necessary flags. At the operating system level, state variables might include the state of all CPUs, as well as the state of each process (opened files, ports, and so on).

Checkpoints do not need to simply record the value of *every* state variable. Doing so would result in enormous sized checkpoints. Instead, a checkpoint records only state variables that do not have their default reset state value (typically, this means a non-zero value). If such a strategy is used, then when a checkpoint is applied (that is, when one desired to set the current state to the state recorded in some checkpoint), it is necessary to first reset all state variables to their default value (such as zero), and then apply all the state variable values stored in the checkpoint.

The primary problem, however, is how to rapidly determine which state variables have a non-default value. The brute force approach is to compare the current value of every state variable to that of its default value. If the values differ, then the state variable has been modified (at least since the start of the program) and should be recorded in the checkpoint. Generally, all state variables have a default value of zero. Although a minor issue, it is worth mentioning that this is not always the case. Consider a status register, which also uses some of its bits to hold a non-zero revision number. One cannot simply compare the value of this status register to zero to determine if it has been modified.

For the sake of good performance, this brute force approach is not practical. Consider an environment with a 32-bit addressable memory system. Each time a checkpoint is created, this approach would require 2^{32} comparisons. That is, it would need to compare each memory address to its default value. As 64-bit memory systems become more pervasive, the problem is compounded further. Likewise, comparisons would need to be made for all other state variables (register values, flags, etc.). However, there are generally only a few such variables with respect to the number of comparisons needed for any memory system.

The advantage of the brute force approach is that it is easy to implement. The major drawback is its performance, since (typically) comparisons are computationally expensive. If checkpoints are created infrequently, and the number of state variables is sufficiently small enough, this approach might in fact be practical. However, another issue with this approach is that it stores more information than necessary. An alternative approach is derived by the following idea: a checkpoint need only record the value of those state variables that have been modified since the last checkpoint.

There are a number of reasons why it would be more practical to have a checkpoint store all state variables whose values differ from their default value, rather than storing only those state variables modified since the last checkpoint. The primary reason is consistency and allowing checkpoints to be independent: any checkpoint by itself can be used to resolve the state of the environment at the cycle during which that checkpoint was created. If a checkpoint stores only state changes since the last checkpoint, the state for that checkpoint can only truly be resolved by using earlier checkpoints or the current set of state history records. This can cause problems if, for instance, earlier checkpoints are removed or corrupted.

If a checkpoint only stores the value of state variables modified since the last checkpoint, the implementation of the undo-process becomes more difficult (as mentioned in the previous chapter). While a significant amount of space can be conserved, the space saved might be less significant if an efficient checkpoint culling method is implemented (which is discussed in Chapter 5). However, this notion at least introduces a technique that can be used to speed up the process of determining which state variables have been modified. This technique can be realized by the following suggestion: in order to determine which state variables have been modified, we need only to maintain a roster of which variables have been modified. The following describes this idea more formally.

All of the state variables can be considered to be part of the set **S**. Each element of **S** itself contains a pair of two elements: a name and a value associated with that name. Each name in the set corresponds to some state variable (such as a register or memory address). When a checkpoint is created, we need only to record those state variables that do not have their default reset state value. Rather than scan the entire set **S**, looking for modified state variables (by performing comparisons), we can instead define another set **M** as follows: whenever a state variable is modified, the name of that variable is inserted into set **M**, so long as that name is not already present in **M** (i.e. the elements of **M** must be distinct, which can be enforced in constant time by using a hash table). It might be helpful to divide **M** into several sets, such as $\mathbf{M}_\mathbf{R}$ (for modified registers) and $\mathbf{M}_\mathbf{M}$ (modified memory).

By maintaining a set of modified state variables, creating a checkpoint becomes much easier and no comparisons are necessary. The checkpoint simply records the value of all of the state variables identified in **M**. If one desires to have checkpoints record only those state variables modified since the last checkpoint, this can be easily achieved by clearing the contents of **M** following the creation of each checkpoint.

There are three primary drawbacks to this approach: (1) Maintaining a list of modified state variables consumes memory space. The amount of space used depends on the nature of program. If this issue becomes a significant problem, one could revert to using the brute-force approach, which has no memory overhead. (2) The time required to modify any state variable is no longer constant. When a state variable is modified, its name is added to **M** only if that name is not already present in **M**. This requires a non-constant amount of time. It is important to enforce this condition, in order to bound the space complexity of **M**. (3) When performing an undo-operation, it might be necessary to remove an entry from **M**. Suppose an instruction results in register `R3` being modified for the first time, and so the name "`R3`" is added to **M**. Then, suppose that instruction is undone, meaning that register `R3` has no longer been modified. As a result, "`R3`" should be removed from **M**.

A reasonable solution to resolve this last issue would be as follows: tag each name in **M** with the number of times the state variable of that name has been modified. When the name is first entered into **M**, it is tagged with the value 1. If that name continued to be modified, simply increment its tag count. When performing an undo, the tag associated with a state variable (if present in **M**) is decremented. If the tag becomes zero, the associated state variable is removed from **M**. Due to the computational and space overhead of this solution, it may also be acceptable to simply ignore this issue. The disadvantage of doing so would be that when earlier checkpoints are re-created, such as during re-execution, they might include state variables that have not actually been modified at that point in time.

Two techniques for creating checkpoints have been discussed: by brute-force (BF) or by using a modified state variable set (MSVS). There are advantages and disadvantages to either approach. The advantage of BF is its simplicity and that it does not affect the space complexity. The primary disadvantage is its slow linear time complexity. Although linear, `O(n)`, the value of `n` (the number of state variables in the environment) is very large. Using a MSVS reduces the time complexity to `O(m)`, where `m` is only the number of modified state variables (since the last checkpoint). Commonly, `m << n`, however `m` can potentially be as large as `n`. The disadvantage of using a MSVS is that the space complexity also increases by `O(m)`. An adaptive environment would have implementations of either approach available.

For the implementation of JIMS, a combination of both approaches is used: BF is used for register and internal flags, MSVS is used only for memory addresses. The justification is that the number of registers and internal flags is small (approximately 100 variables), while the number of memory addresses is enormous (2^{32} variables). JIMS is implemented such that the use of a MSVS can be readily replaced by the BF approach.

While the use of a MSVS is appealing, some optimizations might be made to reduce the number of comparisons necessary in the BF approach. For example, a program might modify only a specific region of memory. If this range can be predetermined [10, 12], then only the memory values in that range need be compared, not the entire address space.

As a more general approach, the main memory address space can be divided into logical segments (for example, a 32-bit address space can be divided into 2^{16} segments each of size 2^{16}). A “modified flag” is associated with each segment, and has a default value of false. When any memory address is modified, the corresponding modified flag of that segment is set to true. Then, when a creating a checkpoint, only those segments whose modified flag is set are considered. This approach can be used by operating systems that create checkpoints, where the read/write bit of each memory page is used to determine the set of modified memory pages [5, 12].

The implementation of the memory model used by JIMS makes this optimization highly practical. The address space, which is 32-bits, is divided into segments of size 2^8. A separate modified flag is not necessary, as the modified flag is implied by the use of a null-pointer. That is, suppose no memory address has currently been modified (each address, or memory cell, is then assumed to have a default value of "0"). Then suppose that an instruction is executed which results in address 0xAABBCCDD being set to the value "78."

Figure 3.1 shows how JIMS represents this fact in its memory model. The Main Memory Buckets represent the high byte of memory addresses (i.e. 0xFF000000), Alpha Memory Buckets represent the next byte (0x00FF0000), followed by Beta Memory Buckets (0x0000FF00). The Gamma Memory Buckets represent a segment of 2^8 memory cells (from 0xXXXXXX00 to 0xXXXXXXFF).

Notice that most of the buckets point to null. This indicates that no address in that region has been initialized (or in other words, modified). For instance, in the Main Memory Bucket, entry "01" points to null. This indicates that no address from 0x01000000 to 0x01FFFFFF has been modified. In the Alpha Memory Bucket, entry "02" points to null. In this case, this indicates that no address from 0xAA020000 to 0xAA02FFFF has been modified. However, by following the pointers in Figure 3.1, notice that address 0xAABBCCDD leads to a cell whose value is 78.

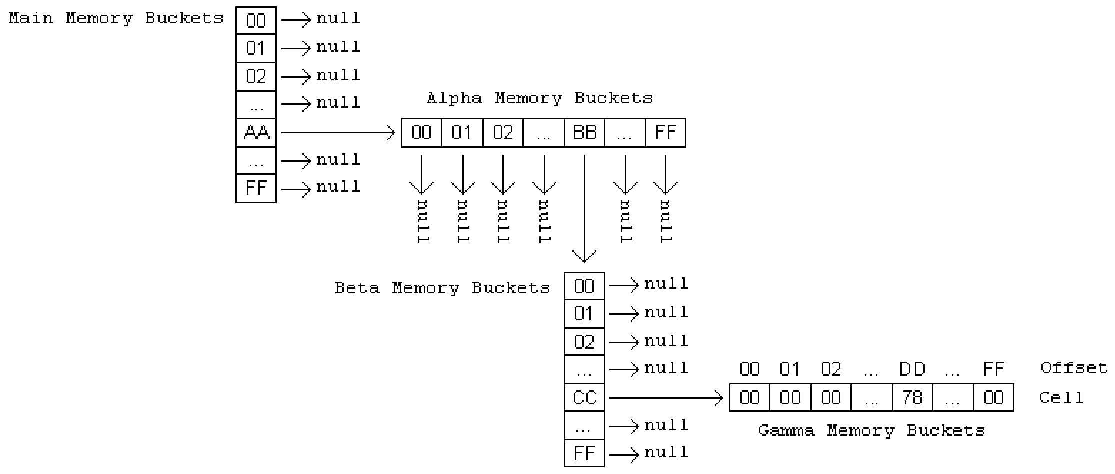


Figure 3.1: JIMS Memory Model

Using such a memory model, one can quickly determine which regions of memory have been modified. When a bucket that points to null is encountered, it is simply ignored and the next bucket is considered. This significantly reduces the number of comparisons necessary in the BF approach. Although many memory systems are not modeled in this fashion, a technique similar to this may be a reasonable substitute for using a MSVS.

CHAPTER 4
WHEN TO CREATE CHECKPOINTS

The previous chapter discusses how to create checkpoints, using two different approaches. For this chapter, the issue of deciding when to create a checkpoint is addressed. Recall that checkpoints are not created during every cycle. Only state change records are constructed every cycle, which hold incremental changes to the state of the environment over time.

Checkpoints are used to essentially compress a set of state change records into one entity. More directly, a checkpoint represents the accumulation of state change records. This reduces the growth rate of state history information, while retaining sufficient information to allow a program to be undone indefinitely. However, one must decide when it is appropriate for a checkpoint to be created. If checkpoints are created too frequently, then they will increase the space requirement rather than decrease it. On the other hand, if checkpoints are not created often enough, the performance of the undo process can be severely degraded. The following discussion describes several strategies for determining when to create checkpoints.

The first strategy, called `NAÏVE`, is to simply wait a constant number of cycles. For example, the system might dictate that checkpoints be created every 1000 cycles of execution. Obviously this strategy is easy to implement and consistent. However, it is not very adaptive. Waiting n cycles might be suitable for some programs, but not others. Most likely the program would need to be run several times in order to determine an appropriate value of n, which is probably not desirable.

The second alternative is identified as `WUMF` (wait until memory full). As the acronym implies, this approach waits until memory becomes full (or nearly full) due to the creation of many state change records. Once this condition is detected, a checkpoint is created and the system proceeds. This allows the maximum number of cycles to be recorded, before it becomes necessary to create a checkpoint. This approach may be difficult to implement. For instance, at the hardware level, most hardware architectures do not have any notion of memory being full or not. This approach might be suitable for a compiler, where exceptions and other means can be used to indicate when memory for the current process is full.

For lack of a better acronym, the third approach is identified as `WUA` (wait until appropriate) and is described as follows: a checkpoint is created when it is determined that the size of the created checkpoint would be less than or equal to, by some constant, the size of the set of state change records. This approach guarantees that creating a checkpoint would reduce the space consumed by state history information, and also ensures that checkpoints are created frequently enough. The primary drawback is that it may be difficult to determine the size of a potential checkpoint without actually creating the checkpoint.

The three approaches described above assume the use of both state change records and checkpoints. An alternative approach might only use checkpoints, then augment an existing program source code to specify when checkpoints should be created. For instance, with a high level programming language, one could manually specify when to create a checkpoint [12]. As an example, the program might call a `makeCheckpoint()`

function call shortly before calling some other major function, or entering some significant program loop. A compiler might be made to do this automatically.

Each of these approaches (NAÏVE, WUMF, WUA, or manually specified) is a viable solutions for determining when checkpoints should be created. The approach that is used would depend on the nature of the environment. There may also be an advantage to using a combination of these three approaches within the same environment. The JIMS project uses the WUA approach, since it appears to be the most general solution (given the use of both state change records and checkpoints within the same system). The remainder of this section discusses the primary issues in implementing this approach.

The Wait Until Appropriate (WUA) approach relies on being able to determine the size of a checkpoint. Furthermore, it must also be able to determine the size of the current set of state records. For example, suppose the current cycle of execution is 100, and the size of the state records is 800 bytes (representing the state changes made during the last 100 cycles). Suppose it could be determined that if a checkpoint were created during the next cycle of execution, it would result in a checkpoint of size 300 bytes. Therefore, if the checkpoint were created, then 500 bytes of space would be conserved (since the 800 bytes used for state records would be compacted to one checkpoint of 300 bytes). Notice that the size of the checkpoint is less than the size of the state records by a constant factor of 2.66. (= 800/300).

In general, the WUA approach can be described by the following conditional statement:

```
if ( (CheckpointCreationConstant * NextCheckpointSize) <=
     StateHistorySize) then CreateCheckpoint()
```

The *StateHistorySize* variable is the size of the current set of state change records (such as in number of bytes). Similarly, *NextCheckpointSize* is the size of the checkpoint that would be created if a checkpoint were in fact created during the current cycle. The *CheckpointCreationConstant* is simply a constant that offers control over when a checkpoint is actually created. This constant factor is important and will be discussed later. For now, it is important to consider how the size of a checkpoint and the state record set can be determined quickly (since this information must be available during each cycle of execution).

For the size of state change records, a separate integer variable is used to represent this information. The value of this variable is incremented and decremented accordingly as state records are added (during step) and removed (during undo), and set to zero following the creation of any checkpoint. The size of a checkpoint is approximated in a similar fashion, by knowing the number of state variables (and of what kind) that have been modified. For example, if five memory addresses have been modified and two register values, the size of the checkpoint could be estimated as `5*`*`BytesPerModifiedMemory`*` + 2*`*`BytesPerModifiedRegister`*.

As mentioned earlier, the *CheckpointCreationConstant* is a significant part of the `WUA` approach. This constant controls how often checkpoints are created. In order to better understand the role of this constant, three different MIPS assembly programs were experimented with. Each program was executed (using JIMS Release 1) until completion 22 times, where each execution used a different value of the constant.

The three programs are briefly described as follows: (1) `SQRT` determines the square root of a large double precision number using Newton's Method. This program has a short runtime and performs only one memory reference. (2) `SORT` uses the Swap Sort technique to sort a given sequence of numbers stored in memory. (3) `FACTOR` uses a Naïve Factoring Algorithm to determine all factors of a given integer.

Figure 4.1, at the end of this chapter, show the effect of different constant values for these three programs. When the *`CheckpointCreationConstant`* is set to zero, this implies that checkpoint creation is disabled (i.e. no checkpoints are created). Each graph shows the size of the state history (including both state change records and checkpoints), which is recorded in bytes. In addition, each graph shows the average distance between checkpoints, in terms of number of cycles. These two statistics were recorded once the program terminated. As per the JIMS Release 1 implementation, checkpoints record all modified state variables at the time they are created (not just the modified state variables since the previous checkpoint). For this experiment, checkpoint culling (which is discussed in the next chapter) was not enabled.

From these three graphs, the following trends are observed regarding the value of the *`CheckpointCreationConstant`*: (1) A value of `1.0` offers no practical benefit, except to remove the small overhead of maintaining a large set of state change records. (2) A value of `2.0` apparently reduces the size of the state history by the fastest rate of improvement. Larger values typically offer greater reductions in the size of the state history, but at a much slower rate of improvement. (3) As the value of the constant increases, so does the distance between checkpoints. (4) A large constant value is not necessarily better. For example, in the `SQRT` program, a value of 21 is so large that the program terminates before any checkpoint is ever made.

As a conclusion, the value of the *CheckpointCreationConstant* should probably be between 2 and 5. Such a range ensures that checkpoints will help in reducing the growth rate of the state history, while keeping the average distance between checkpoints reasonably small. Allowing checkpoints to be excessively far apart neglects the performance of large undo operations. The reason for this is because it is increases the probability that re-execution will be necessary, and moreover increases the amount of re-execution required.

The WUA approach easily reduces the space used for state history by 50 percent, even with the use of small values for the *CheckpointCreationConstant*. With the three programs used for the experiment discussed above, constant values ranging from 6 to 20 were found to reduce the state history size by 75 to 90 percent (depending on the nature of the program and the particular constant used). Larger constant values can offer further reductions, but risk increasing the average distance between checkpoints too greatly. As a final note, the *CheckpointCreationConstant* does not need to be limited to integer values only. The general trends that were observed apply also to any real number greater or equal to 1.0.

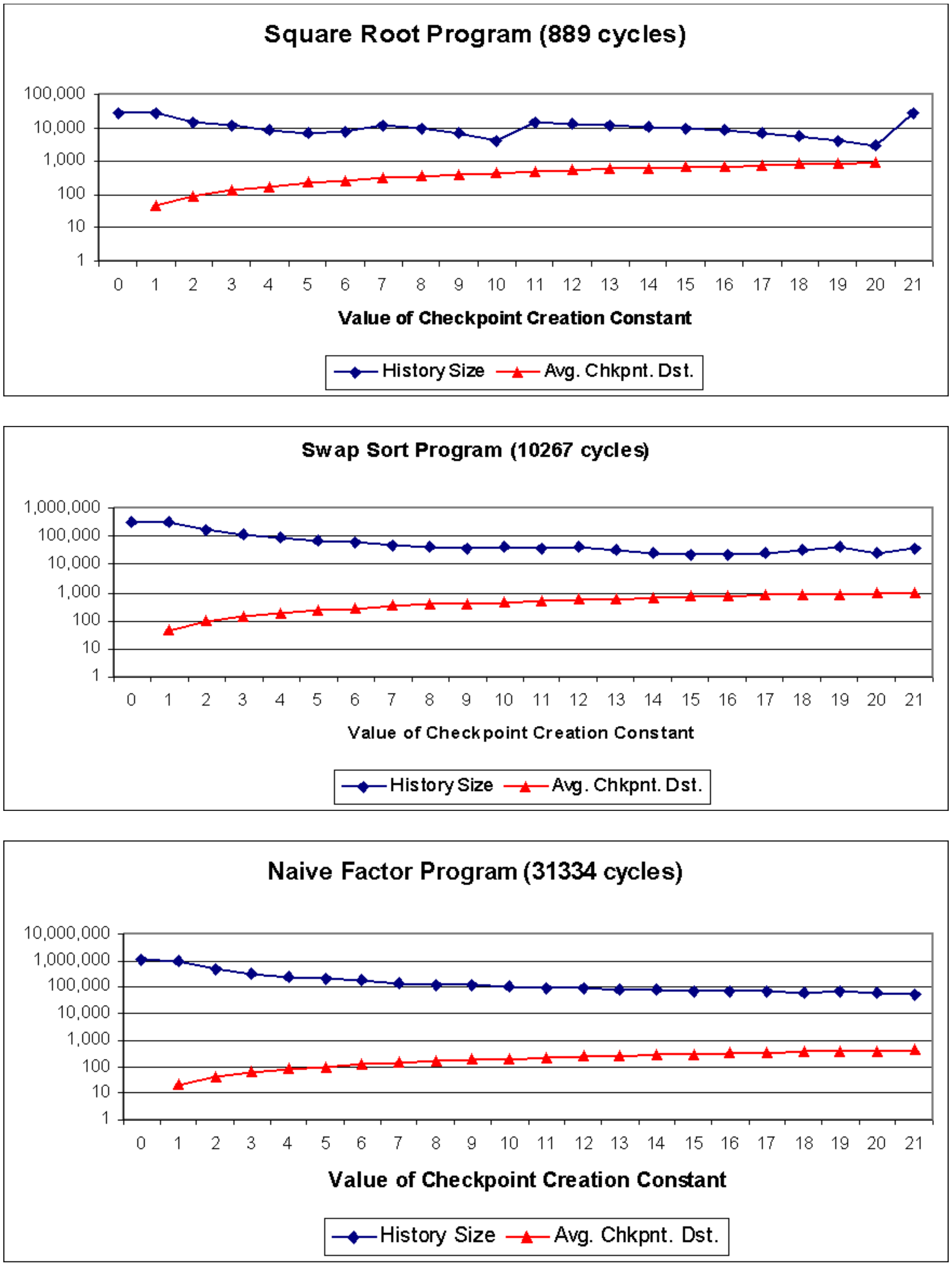


Figure 4.1: Effect of Checkpoint Creation Constant

CHAPTER 5
CHECKPOINT CULLING PROCESS

The techniques and issues regarding how and when to create checkpoints are discussed in earlier chapters. While these techniques are sufficient to support reversible debugging, the growth of the state history information remains linearly unbounded. This chapter discusses the Checkpoint Culling Process (CCP), which is a technique used to maintain logarithmic growth of state history information.

Should the size of the state history information consume all available space, or at least a significant portion thereof, there are a number of obvious action that can be taken: (1) Signal an error to the user and indicate that additional storage capacity is needed. The user would then need to increase the capacity of the storage medium, such as main memory, and restart the program. (2) Erase all current state history information and start recording over from the current program position. (3) Erase some of the state history information (such as oldest first), allowing for additional state changes to be recorded.

The third action describe above is essentially what the CCP approach does, except it does not wait until the available space has been exhausted. The idea is analogous to what people tend to remember. That is, people tend to forget details of events that took place long ago, while remembering more details of recent events. With regards to checkpoints, it is more likely that only recent checkpoints will actually ever be used.

The Checkpoint Culling Process is applied following the creation of each checkpoint. This processes uses an algorithm that maintains recent checkpoints, but gradually removes old checkpoints. The Checkpoint Culling Algorithm[1] is described in Figure 5.1.

```
Algorithm CullCheckpoints(C, t, z) {
  // C is the current set of checkpoints
  // t is the current cycle index
  // z is the cull rate (2, 3, 4, ...)
  if ( $t \leq 0$ ) then
    return
  for ( $0 \leq k \leq \lfloor \log_z t \rfloor$ ) {
    $low = \max[0, (t - z^{k+1}) + 1]$
    $high = t - z^k$
    Mark all checkpoints in C whose time index is between
      low and high (inclusive).
    Remove all marked checkpoints from C, except the one
      that has the earliest time index.
    Unmark the checkpoint that was not removed.
  }
}
```

Figure 5.1: Checkpoint Culling Algorithm

The `CullCheckpoints` algorithm provides a logarithmic bound on the size of the state history. The Table 5.1 shows which checkpoints would be culled out over time, using the `CullCheckpoints` algorithm with a *z* value of `2.0`. Larger values of *z* result in a slower rate of state history growth, though increases the average distance between culled checkpoints.

[1] Dr. Michael P. Frank and I formulated this algorithm in personal discussions during the development of this thesis.

Table 5.1: Checkpoint Culling Algorithm for $z = 2$

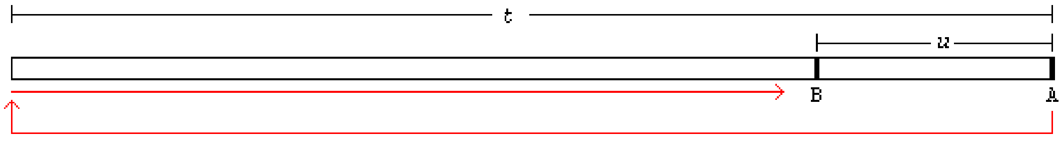
t	COLUMN I	COLUMN II	COLUMN III					
0	X	X	*initial state checkpoint*					
1	XX	XX	0 to 0					
2	XXX	XXX	0 to 0	1 to 1				
3	X-XX	XXX	0 to 1	2 to 2				
4	X-XXX	XXXX	0 to 0	1 to 2	3 to 3			
5	X-X-XX	XXXX	0 to 1	2 to 3	4 to 4			
6	X---XXX	XXXX	0 to 2	3 to 4	5 to 5			
7	X---X-XX	XXXX	0 to 3	4 to 5	6 to 6			
8	X---X-XXX	XXXXX	0 to 0	1 to 4	5 to 6	7 to 7		
9	X---X-X-XX	XXXXX	0 to 1	2 to 5	6 to 7	8 to 8		
10	X---X---XXX	XXXXX	0 to 2	3 to 6	7 to 8	9 to 9		
11	X---X---X-XX	XXXXX	0 to 3	4 to 7	8 to 9	10 to 10		
12	X-------X-XXX	XXXXX	0 to 4	5 to 8	9 to 10	11 to 11		
13	X-------X-X-XX	XXXXX	0 to 5	6 to 9	10 to 11	12 to 12		
14	X-------X---XXX	XXXXX	0 to 6	7 to 10	11 to 12	13 to 13		
15	X-------X---X-XX	XXXXX	0 to 7	8 to 11	12 to 13	14 to 14		
16	X-------X---X-XXX	XXXXXX	0 to 0	1 to 8	9 to 12	13 to 14	15 to 15	
17	X-------X---X-X-XX	XXXXXX	0 to 1	2 to 9	10 to 13	14 to 15	16 to 16	
18	X-------X---X---XXX	XXXXXX	0 to 2	3 to 10	11 to 14	15 to 16	17 to 17	
19	X-------X---X---X-XX	XXXXXX	0 to 3	4 to 11	12 to 15	16 to 17	18 to 18	
20	X-------X-------X-XXX	XXXXXX	0 to 4	5 to 12	13 to 16	17 to 18	19 to 19	
21	X-------X-------X-X-XX	XXXXXX	0 to 5	6 to 13	14 to 17	18 to 19	20 to 20	
22	X-------X-------X---XXX	XXXXXX	0 to 6	7 to 14	15 to 18	19 to 20	21 to 21	
23	X-------X-------X---X-XX	XXXXXX	0 to 7	8 to 15	16 to 19	20 to 21	22 to 22	
24	X---------------X---X-XXX	XXXXXX	0 to 8	9 to 16	17 to 20	21 to 22	23 to 23	
25	X---------------X---X-X-XX	XXXXXX	0 to 9	10 to 17	18 to 21	22 to 23	24 to 24	
26	X---------------X---X---XXX	XXXXXX	0 to 10	11 to 18	19 to 22	23 to 24	25 to 25	
27	X---------------X---X---X-XX	XXXXXX	0 to 11	12 to 19	20 to 23	24 to 25	26 to 26	
28	X---------------X-------X-XXX	XXXXXX	0 to 12	13 to 20	21 to 24	25 to 26	27 to 27	
29	X---------------X-------X-X-XX	XXXXXX	0 to 13	14 to 21	22 to 25	26 to 27	28 to 28	
30	X---------------X-------X---XXX	XXXXXX	0 to 14	15 to 22	23 to 26	27 to 28	29 to 29	
31	X---------------X-------X---X-XX	XXXXXX	0 to 15	16 to 23	24 to 27	28 to 29	30 to 30	
32	X---------------X-------X---X-XXX	XXXXXXX	0 to 0	1 to 16	17 to 24	25 to 28	29 to 30	31 to 31
33	X---------------X-------X---X-X-XX	XXXXXXX	0 to 1	2 to 17	18 to 25	26 to 29	30 to 31	32 to 32
34	...							

`COLUMN I` of Table 5.1 shows a diagram that represents the current set of checkpoints created over time. A dash (`-`) represents a checkpoint that was removed, while an `X` represents a checkpoint that has been retained. Notice that over time (increasing values of `t`), new `X` marks are added while at certain times earlier `X` marks are removed. Furthermore, in each row, the initial state checkpoint and the two previously constructed checkpoints are always present. This suggests that some optimization can be made by not checking to see if those checkpoints should be removed during the culling process. **`COLUMN II`** simply shows a graph representing the size of the state history information over time. That is, it is the same as the previous column, except the dash entries are removed since they do not occupy any memory. Notice that the graph grows logarithmically. **`COLUMN III`** shows the set of *low* and *high* values calculated by the algorithm. As specified by the algorithm, all existing checkpoints within each of these regions are removed, except the checkpoint that occurs the earliest.

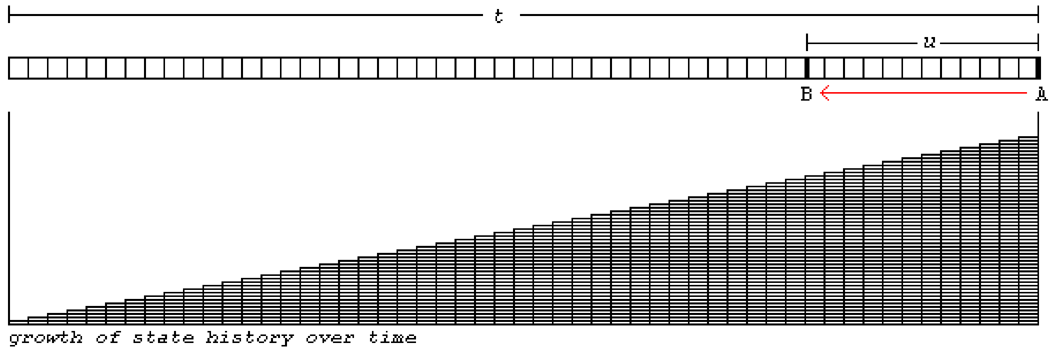
Typically, checkpoints are not created during every cycle (as suggested by Table 5.1). If the average distance between each checkpoint remains constant, then the value of `t` can instead be the cycle counter. Alternatively, each checkpoint can be marked with a static index value. For example, the first checkpoint created is marked with index 0, the second checkpoint created is marked with index 1, the third checkpoint created is marked with index 2, and so on. As checkpoints are removed, these index values do not changed. With this approach, then the value of `t` is not a cycle index, but instead a checkpoint index.

The remainder of this discussion describes the motivation behind the Checkpoint Culling Algorithm. Let t be the current cycle index, u be the number of steps to be undone, and $f(t,u)$ represents the worst-case number of steps required to go backwards u steps from cycle t. Keep in mind that, most often, $u << t$. That is, the number of desired undo cycles is much smaller than the total number of previously executed cycles. The definition of $f(t,u)$ depends on what state history information is available, as described by the following four cases.

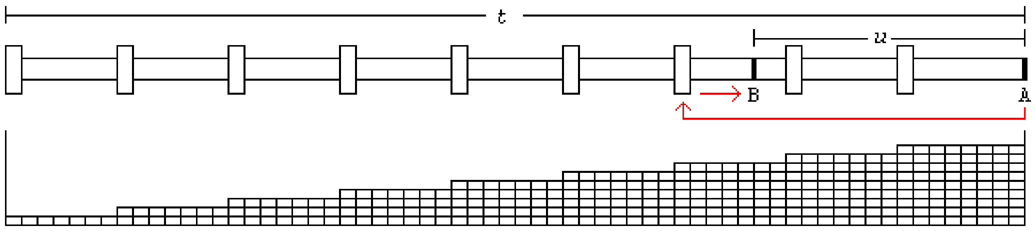
- **CASE 1: No state history recorded.** Since there is no state history information available, we would need to perform re-execution in order to undo u steps. We would start from the initial state (perhaps by resetting the environment and reloading the program), then re-execute $t-u$ cycles forward. Therefore, $f(t,u)$ is $\Theta(t-u)$. Since $u << t$ may mean that $u = o(t)$, then $f(t,u) = \Omega(t)$, which is a poor time complexity since t is often very large. On the other hand, the space complexity is not affected in this case.



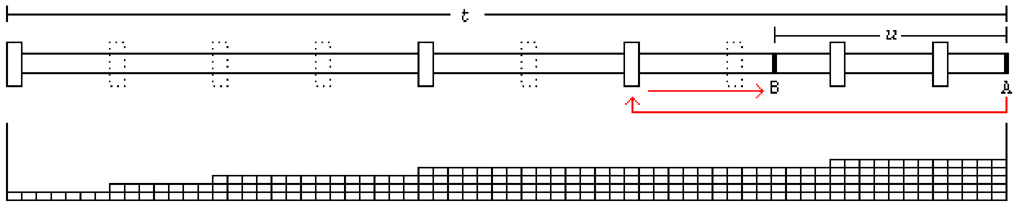
- **CASE 2: Complete incremental history recording.** The state changes for each and every cycle of execution is recorded, thus we can readily reverse u steps by applying the available state history in reverse. Therefore, $f(t,u)$ is $O(u)$, which is reasonable since often $u << t$. However, this case requires $\Omega(t)$ space to store the incremental state changes (see Chapter 2).



- **CASE 3: Complete checkpoints recorded at constant intervals.** Checkpoints are created periodically, at some relatively constant time interval. In this case, $f(t,u)$ is $\Theta(1)$. Undoing u steps involves first finding the checkpoint whose cycle index is closest to (but does not exceed) $t-u$. The state is set to said checkpoint, then we perform (at most) a constant number of steps to reach the actual desired cycle time. While a constant time complexity is very desirable, this technique requires $\Omega(ts)$ space (where s is any checkpoint size).



- **CASE 4: Checkpoints culling.** Checkpoints are created as in the previous case, but the Checkpoint Culling Algorithm is applied following the creation of each checkpoint. This acts as a compromise between the time-inefficiency of case 1 (re-execution) and the space-inefficiency of case 2 (complete history recording). The result is that $f(t,u)$ is $O(u)$ with a space complexity of $O(\log t)$.



The Checkpoint Culling Algorithm enforces the following invariant: there is always at least one checkpoint between each $low = (t - z^{k+1}) + 1$ and $high = t - z^{k}$ range for all values of k (i.e. $0 \le k \le \lfloor \log_z t \rfloor$). This is clear by inspecting the culling algorithm or observing Table 5.1 (which models the execution of the algorithm).

Suppose the user desires to undo u steps ($u \le t$) and that this invariant holds. If u is less than the current number of incremental state change records, if any, then the undo operation is $O(u)$ as described in case 2 above. Otherwise, the undo operation requires re-execution, which (as will be shown) also has a time complexity of $O(u)$.

For any given target index $r = t - u$, there exist a pair of checkpoints whose time index surrounds *r*. We can call these checkpoints C_{low} and C_{high}, such that $t_{C_{low}} \leq r \leq t_{C_{high}}$ (where t_X represents the time index of *X*). The number of cycles to be re-executed would be $e = r - t_{C_{low}} \leq k \cdot u$ (for some *k*). That is, the distance from C_{low} to the target cycle *r* is always less than some constant times *u*, or $O(u)$. Figure 5.2 demonstrates these arguments.

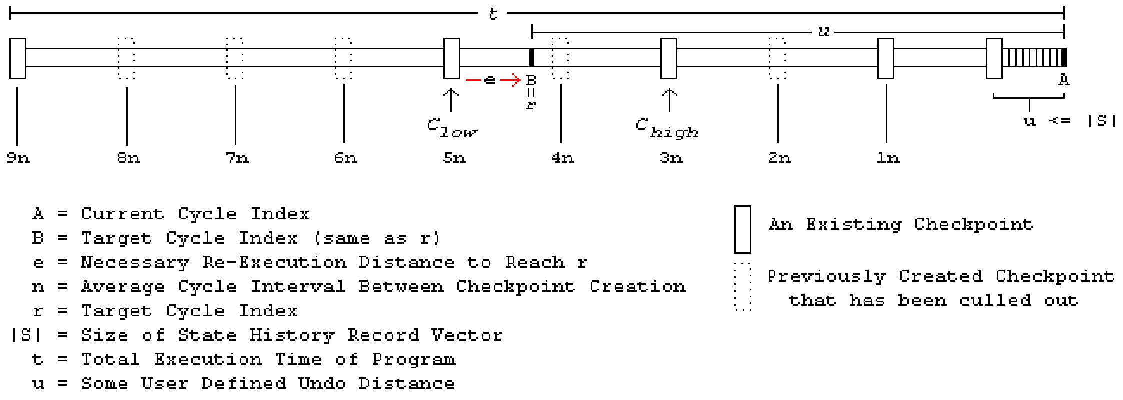

t
u
e
B
=
r
A
u <= |S|
C_low
C_high
9n
8n
7n
6n
5n
4n
3n
2n
1n
A = Current Cycle Index
B = Target Cycle Index (same as r)
e = Necessary Re-Execution Distance to Reach r
n = Average Cycle Interval Between Checkpoint Creation
r = Target Cycle Index
|S| = Size of State History Record Vector
t = Total Execution Time of Program
u = Some User Defined Undo Distance
An Existing Checkpoint
Previously Created Checkpoint that has been culled out

Figure 5.2: State History Model With Checkpoint Culling

Clearly, the advantage of the Checkpoint Culling Algorithm is to reduce the state history growth rate from a linear function to a logarithmic function while also keeping the undo time linear. The runtime of the algorithm itself should not pose any significant overhead. In the JIMS project, the algorithm was implemented such that the vector of checkpoint needed to be scanned only once, thus a time complexity of $O(n)$, where n is the number of checkpoints.

State history recording generally requires a massive amount of readily available storage space. Therefore, an asymptotic reduction in space usage generally outweighs any corresponding sub-optimal runtime performance overhead.

CHAPTER 6
RESULTS AND ISSUES

The previous chapters of this thesis discuss a number of concepts: state history records that record incremental changes to the state of the environment, periodic checkpoints that record the entire state of the environment, strategies for how and when to create checkpoints, and an algorithm that reduces the size of recorded state history. All of these concepts can be applied together to provide an environment that efficiently supports reversible execution, and thus reversible debugging.

The use of state history records allows programs to be executed in reverse at nearly the same speed that they are executed forward. However, state history records rapidly consume memory space. In addition, to be of any use, incremental state history records must be generated for each executed cycle, which thus reduces normal forward execution performance (due to the overhead of creating and storing state history records).

For the most part, state change records and checkpoints are independent concepts. That is, one might choose to use either strategy (one or the other) or both strategies together to implement reversible execution. If only checkpoints are used, then re-execution will always be necessary in order to perform reversible execution. If only state change records are used, then the growth rate of state history information will likely be too rapid to support programs for any useful amount of runtime.

The use of both state change records and checkpoints in the same environment combines the advantages and disadvantages of both concepts. That is, the growth rate of state history is reduced, and relatively small undo operations do not require re-execution.

However, the growth rate remains linear, and relatively large undo operations will require re-execution. The checkpoint culling process can greatly reduce the space usage of state history information. But the issues related to re-execution require special attention, which is discussed later in this chapter.

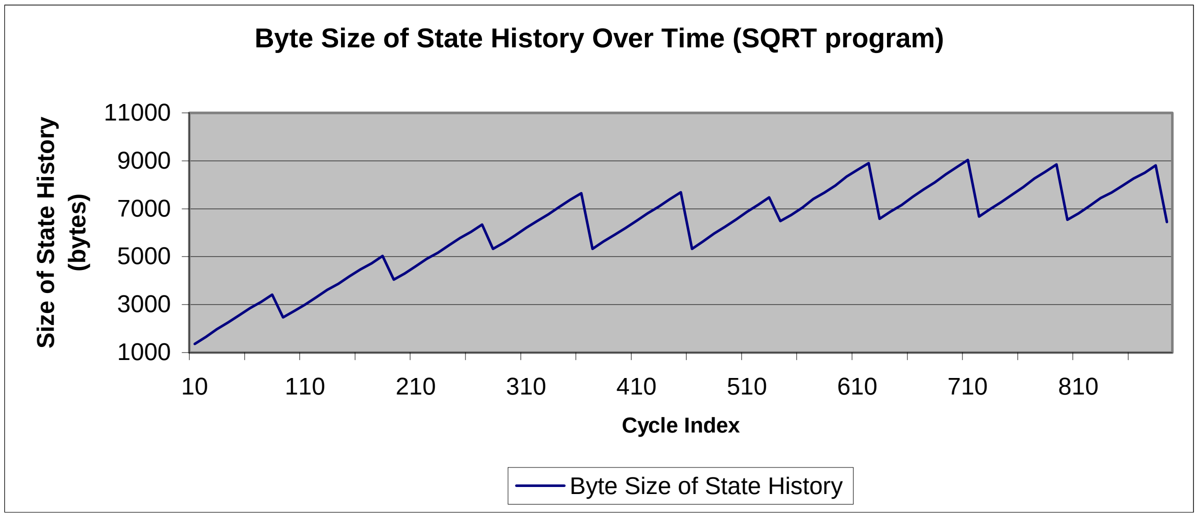
State History Growth Statistics

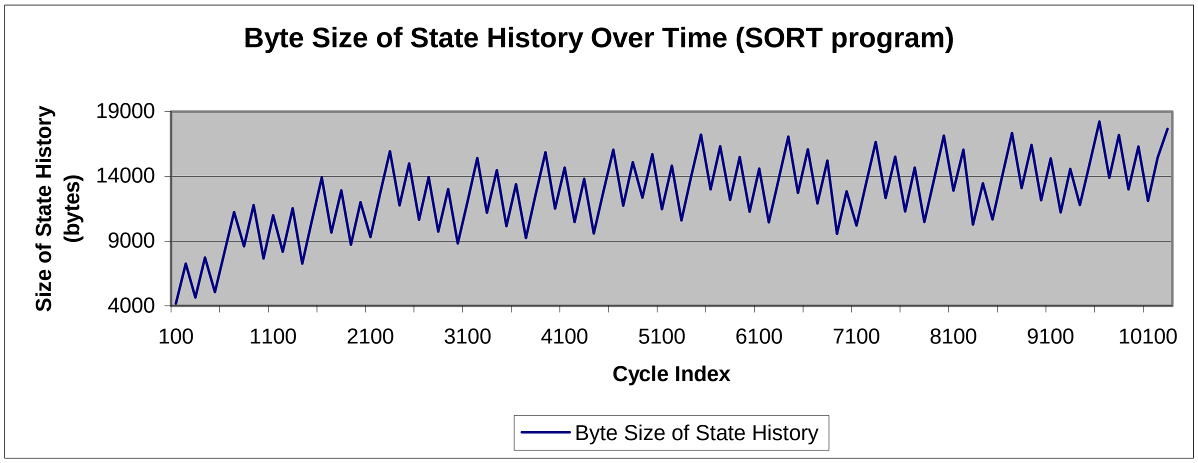
In order to demonstrate the growth of state history, using the techniques described in this thesis, the three sample programs described in Chapter 4 were used to construct Figure 6.1. Each program was executed forward some constant number of steps. At the end of each step, the current size of the state history was recorded. This allows us to examine the growth of the state history over time.

For all three programs, both checkpoints and state change records were recorded, and the Checkpoint Culling Algorithm was applied. The `SORT` program performs mostly memory references, unlike the other two programs. As a result, it has a larger state history. For the `SQRT` program, a Checkpoint Creation Constant of `2.0` was used (see Chapter 4), while a constant of `5.0` was used for `SORT` and `FACTOR`.

The use of checkpoints causes the jagged growth of state history. That is, a sudden reduction in the size of the state history indicates when a checkpoint was created. However, notice that the peaks in each graph indicate a general logarithmic growth pattern. This is due to the Checkpoint Culling Algorithm, which gradually removes older checkpoints.

The data for these graphs was produced using JIMS Release 1, which stores state history as a sequence of ASCII characters. Using binary encoded sequences instead, the general state history byte size would be easily reduced by as much as 50 percent.





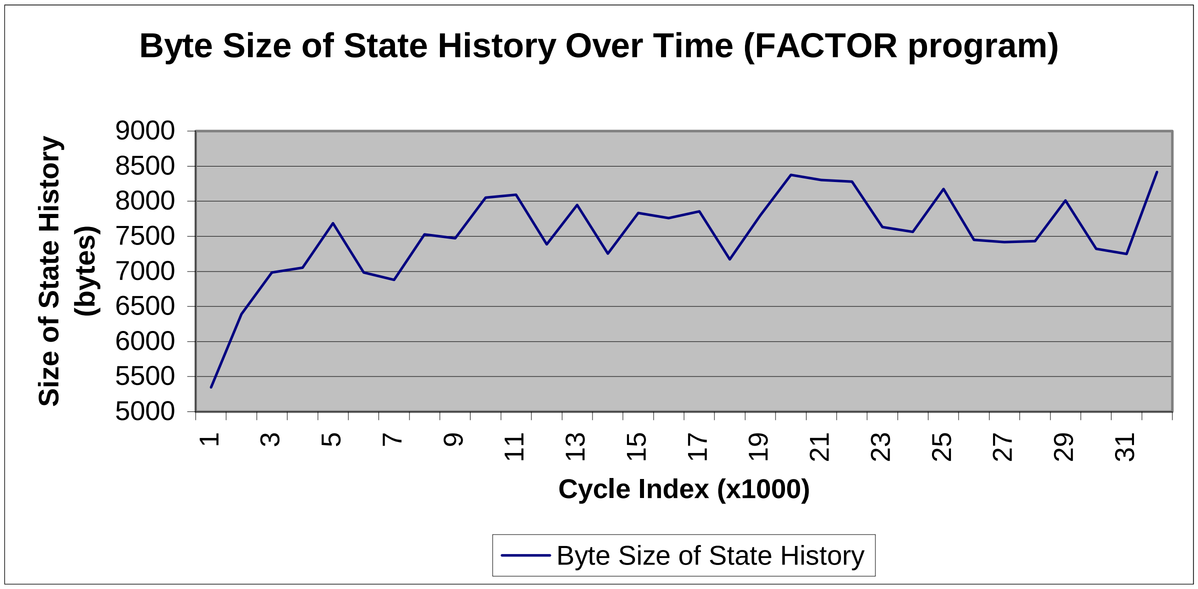


Figure 6.1: State History Growth Statistics Using JIMS

Re-Execution Issues

As mentioned earlier, there are several issues to be considered when performing re-execution [13]. During re-execution, the user would typically expect the program to behave as it did the last time the program text was executed (all things being the same). However, as an example of a common problem, suppose that the current state history is as shown in Figure 6.2.

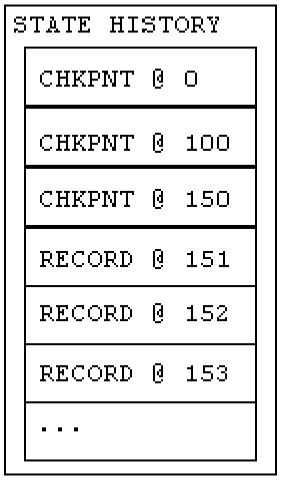


Figure 6.2: Sample State History

That is, three checkpoints were created during cycle 0, 100, and 150. Three state change records were created following the last checkpoint, during cycle 151, 152, and 153. The current cycle is 154. Suppose the user desired to revert back to cycle 120 (or, in other words, requested reverse execution of 34 cycles). Using the information available from state history, one would first apply the state recorded in the second checkpoint (created during cycle 100), and then perform 19 cycles of forward execution (i.e. re-execution) such that the current cycle becomes 120 as desired by the user.

So long as the program is deterministic, re-execution is acceptable. Moreover, in the example described above, only 19 cycles had to be re-executed forward instead of 34 cycles in reverse. The problem of re-execution arise when re-executing non-deterministic program text. Suppose, for example, that during cycle 110 the program requested some input whose value affected the control flow of the program. In order to properly return to cycle 120 (as the user had last experienced it), the process of re-execution must ensure that (at least) all input is the same as during the original execution of those cycles (in this case, cycle 101 through 119).

One way to alleviate this problem is to use state change records exclusively, without checkpoints. In doing so, the user can then truly perform reverse execution, and re-execution is never necessary. Of course, if only state change records are used, then the memory system must have enormous capacity (refer to Figure 2.6).

There are several other feasible solutions to the problem of re-execution. One could ignore the problem altogether, and simply require that the user manually re-specify any input. While acceptable for some user-controlled input (such as from a keyboard), some input is not within the control of the user (for instance, a temperature probe that is used to periodically polled for the current temperature). Another solution might be to use a separate vector to record all input into the system. When the program is re-executed, it retrieves input from the input vector, rather than the environment. The primary drawback to this solution is that it is proprietary and requires knowledge of what the inputs of the system are. For debugging purposes, however, one might be able to define a limited set of inputs to be recorded during that particular debugging sessions.

One other solution might be to give the user some control, or full manual control, over when state change records and checkpoints are used. Once the user has identified a region where reversible execution would be useful, the user could specify that only state change records be used during that region of the programs execution. This might require that the entire program (or portions thereof) to be executed multiple times. In addition, this approach might only be feasible at the application level, such as within a debugging application.

Irreversible Effects

For the purpose of this thesis, the term *state variable* has been used in reference to internal state variables (such as CPU registers, memory cells, internal flags, etc.). State variables might also include the contents of a disk, cache memory, or state variables of associated coprocessors. At the operating system level, state variables might consist of various tables (process table, open file table, etc.) and numerous internal global variables.

By recording the changes to the value of these variables, it is then assumed that reversible execution can be performed by re-applying past values to state variables in the reverse order that the values were recorded. However, this assumes that the state variables can be both read and written, which is not always the case. For example, certain regions of main memory might be read-only some of the time, or a certain address can only be written under certain circumstances. As a result, some sequences of instructions cannot be reversed. There is no practical and general solution to this problem. However, there are several ways to elude the problem. Three potential solutions are mentioned in the following discussion.

Reverting to an earlier checkpoint, and therefore bypassing the sequence of irreversible instructions, is one possible solution. Obviously the state of the environment is not truly reversed. Moreover, re-executing from a checkpoint may cause stuttering [14] (i.e. same output repeated multiple times). This solution can probably only be implemented at the application level, which can predict upcoming events that cannot be reversed (and thus create checkpoints as necessary).

A second solution might be to use a virtual environment. A software simulation of an environment can ignore or alter certain aspects of its real world counterpart. For example, a virtual printer could be used, which could allow the contents of a printed page to be undone. Or, as a more practical example, a CPU simulator could allow normally read-only state variables to be both read and write. This solution might be viable for some situations, but obviously the development cost of a simulator can be expensive (in terms of time, manpower, etc.) such as to negate their benefit.

The final solution mentioned here is, as described earlier in this chapter, to record all input. During re-execution, all input is read from the recorded input buffer, and all output is only pretended to be sent (that is, during re-execution, any output is not actually sent since it is assumed to have already been sent). The amount of input can be enormous, especially when considering common serial input devices (e.g. a mouse) and network communication. Some compromise must be made as to what input is recorded, since it does not appear to be practical to record all the input of the environment. As with re-execution discussed earlier, this solution is proprietary since both the inputs and outputs must be known beforehand (or somehow determined otherwise).

CHAPTER 7
CONCLUSION AND FUTURE WORK

The content of this thesis has described some techniques for, and issues related to, state history recording. These techniques are intended to facilitate reversible debugging, and to be applied at the hardware level or either the kernel or user level of the operating system, or in hardware simulators.

Reversible debugging has numerous applications and is an idea that has been considered for some time [15]. While inherent reversibility of instructions seems plausible, such as by using a reversible instruction set, unraveling the input of a system is more difficult. As such, state history recording (or history logging) is currently a much more popular means of supporting reversible execution.

A primitive example of state history recording is the use of tape-back up systems. Should the current data somehow become corrupted, one can restore the data from a copy made the day before. Most tape-back up systems copy the entire contents of a large data storage device, similar to a checkpoint recording the entire state of the environment. Some back up systems are clever enough to record only incremental changes to the data, by recording only those files whose date and/or file size differ from the previous copy made of that file.

Many concepts of the tape-back up process can be applied to state history recording. However, there is at least one fundamental difference: state history recording must be performed in real time, such as not to hinder the normal performance of the system. I surmise that reversible computing has not been extensively applied in the past for the following reason: history recording is both computationally and spatially expensive, and computers of the past could not afford to spend processing and storage resources for this purpose..

Modern computers have much greater computational ability, much lower memory access latency, and much greater memory capacity than those of even one decade ago. This trend seems likely to continue for some time. As a result, it does appear that state history recording can be applied on modern computer systems such that its benefit outweighs the associated performance and space-usage penalty.

The techniques described in the previous chapters of this thesis are sufficient for supporting reversible debugging using state history recording. However, due to time constraints, there were several additional ideas considered that were not fully explored for this project. The rest of this chapter briefly describes some of these additional ideas that were considered.

Checkpoint Deltas

One idea that was mentioned was that checkpoints could record only the state variables modified since the previous checkpoint (rather than having each checkpoint record the entire state). We could call these *checkpoint deltas*. While this should significantly reduce the size of checkpoints, this approach seems to preclude the use of the Checkpoint Culling Algorithm (since if checkpoint N is removed, then checkpoint N+1 might not correctly represent all the deltas since checkpoint N-1).

However, suppose checkpoint deltas were used. Rather than culling out old checkpoints, old checkpoints could instead be merged with newer checkpoints. That is, the same Checkpoint Culling Algorithm could be used. But instead of deleting an old checkpoint, it merges the state values recorded in the culled checkpoint with the subsequent checkpoint. That is, all state values of checkpoint N are copied into checkpoint N+1, except for the state values already recorded in checkpoint N+1. This should save some space by allowing both checkpoint deltas and the Checkpoint Culling Algorithm to be used. However, when setting the state to that of a checkpoint, it is still necessary to apply each checkpoint in sequence from the initial state checkpoint.

Reverse Checkpoint Deltas

An extension to the idea of checkpoint deltas is *reverse checkpoint deltas*. That is, when a checkpoint is created, it is initially empty. Then, when any state variable is modified whose old value is not already recorded in the checkpoint, that state variable and its value are then recorded in the checkpoint. Eventually a new checkpoint (N+1) is created and then the earlier checkpoint (N) is no longer modified. However, if checkpoint N+1 is culled, all its changes are propagated backwards to checkpoint N. Then, to go back to an earlier checkpoint state, one applies checkpoints in reverse from the current state rather than from the initial state.

Circular State Change Record Buffer

Another idea, which was briefly mentioned in Chapter 2, is the use of a circular state change record buffer (rather than a stack). It is worth mentioning here again, since it was not implemented in JIMS Release 1. The idea is that when a checkpoint is created, the current set of state change records is not deleted. Instead, any new state change record simply replaces the oldest state change record. This idea helps in greatly improving the run time of some undo operations. Suppose that just after the creation of a checkpoint, the user desired to undo only one or two instructions (i.e. to return to the state just before the creation of the checkpoint). If all the state change records had been deleted following the creation of the checkpoint, then we would need to revert to some earlier checkpoint, and re-executed forward. If we had instead used a circular buffer for the state change records, then any small undo request could be performed immediately without re-execution.

Alternative Checkpoint Creation Decision

The `WUA` approach described in Chapter 4 seems adaptive to different types of programs, since it considers the size of the state change record buffer and next checkpoint of the currently running program. However, this decision algorithm might benefit from more careful consideration. An alternative way of deciding when to create checkpoint is suggested as follows: create a checkpoint when the size of the state change records becomes greater than or equal the total size of all existing checkpoints (by some constant factor). The motivation is that it is only necessary to *compress* state change records when they become a significant fraction of total state history space usage. In addition, this would eliminate the need to estimate the size of the next checkpoint (as currently required by the `WUA` approach).

Case Studies

The final suggestion for future work would be to perform case studies that investigate the use of the undo operation. For example, obtaining statistics showing the average distance of undo requested, or observing in what situations the undo feature is most often used. This would help us to make a better decision of when to create checkpoints, as well as encourage future developers (of operating systems, compilers, debuggers) or hardware designers to consider implementing reversible execution.

APPENDIX
JAVA IMPLEMENTATION OF MIPS SIMULATOR (J.I.M.S.)

The JIMS project includes a MIPS assembler and simulator, and is primarily intended for academic use. JIMS was developed with the following criteria in mind:

- Implemented in Java, for modern portability reasons.
- Model the instruction set of an R2000/R3000 MIPS microprocessor, which is very well known in both academia and industry.
- Use state history recording to support reversible execution, for academic research purposes.

The content of this appendix includes various notes regarding the implementation and functionality of JIMS Release 1. At the time of this writing, JIMS is a work-in-progress project. However, the Release 1 version of JIMS does satisfy the criteria listed above.

Three sources were instrumental during the development of JIMS [16,17]:

- "Computer Organization & Design: The Hardware/Software Interface" by David A. Patterson and John L. Hennessy (Morgan Kaufmann Publishers, Inc., 1998)
- "See MIPS Run" by Dominic Sweetman (Morgan Kaufmann Publishers, Inc., 1999)
- The pcSPIM simulator by James R. Larus (archive available at ftp://ftp.cs.wisc.edu/pub/spim)

The MIPS Assembler

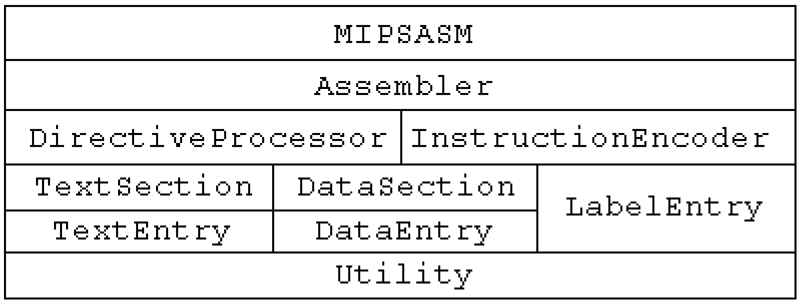


Figure A.1: Layout of JIMS Assembler Source Code

Figure A.1 shows the abstract layout of the JIMS Assembler source, which is organized into five layers. The top layer, which includes only `MIPSASM`, is the user interface layer. The second layer, consisting of the `Assembler` module, only coordinates the assembly process using the lower layers. The third layer includes the primary processing modules, `DirectiveProcessor` and `InstructionEncoder`, which perform the actual work of translating the assembly code into MIPS machine code.

Supported Directives

The follow is a list of assembly directives supported by JIMS Release 1:

NAME AND PARAMTERS	**DESCRIPTION**
.ASCII “string1”(,”string2”,…,”stringN”)	Define a sequence of character strings, starting at the current data address.
.ASCIIZ “string1”(,”string2”,…,”stringN”)	Define a sequence of null terminated character strings, starting at the current data address.
.BYTE b1(,b2,…,bN)	Define sequence of bytes (8-bit sequence), starting at the current data address.
.DATA ([address])	Set the current mode to Program Data. Use the address if specified, otherwise resume from the last or default data address.
.DOUBLE d1(,d2,…,dN)	Define sequence of doubles (64-bit sequence), starting at the current data address.
.FLOAT f1(,f2,…,fN)	Define sequence of floats (32-bit sequence), starting at the current data address.
.HALF h1(,h2,…,hN)	Define sequence of halfs (16-bit sequence), starting at the current data address.
.KDATA ([address])	Set the current mode to Kernel Data. Use the address if specified, otherwise resume from the last or default kdata address.
.KTEXT ([address])	Set the current mode to Kernel Text. Use the address if specified, otherwise resume from the last or default ktext address.
.SET [at] | [noat]	Toggle warning about the use of the $at (assembler temporary) register during assembly.
.SPACE [n]	Define a sequence of n nulls, starting from the current data address.
.TEXT ([address])	Set the current mode to Program Text. Use the address if specified, otherwise resume from the last or default text address.
.WORD w1(,w2,…,wN)	Define a sequence of words (32-bit sequence), starting at the current data address.

The `.ALIGN` directive has not been implemented for Release 1. However, at the same time, misaligned memory access is not yet considered as an exception by the simulator. In addition, the `.EXTERN` and `.GLOBL` directives are not supported and are ignored when encountered.

Supported Instructions

Below is a list of the instructions supported by the JIMS assembler and simulator, which includes most of the standard set of MIPS instructions. The semantics of some instructions is included in the associated description. Supported pseudo instructions are also listed:

INSTRUCTION AND PARAMETERS		**DESCRIPTION (semantic)**
RFE		Return From Exception
SYSCALL	n	Perform System Call n (see below for values of n)
BREAK		Cause Break Exception
MFHI	rd	Move from HI (reg[rd] = $hi)
MFLO	rd	Move from LO (reg[rd] = $lo)
MTHI	rs	Move to HI ($hi = reg[rs])
MTLO	rs	Move to LO ($lo = reg[rs])
J	target	Jump ($pc = addr[target])
JAL	target	Jump and Link ($ra = $pc + 4, $pc = addr[target])
JALR	rs,rd	Jump and Link Register (reg[rd] = $pc + 4, $pc = reg[rs])
JR	rs	Jump Register ($pc = reg[rs])
DIV	rs,rt	Divide ($lo = reg[rs]/reg[rt], $hi = reg[rs]%reg[rt])
DIVU	rs,rt	Divide Unsigned
MULT	rs,rt	Multiply (x = reg[rs]*reg[rt], $hi = high[x], $lo = low[x])
MULTU	rs,rt	Multiply Unsigned
ADDI	rt,rs,imm	Add Immediate (reg[rt] = reg[rs] + imm)
ADDIU	rt,rs,imm	Add Immediate Unsigned
ANDI	rt,rs,imm	AND Immediate (reg[rt] = reg[rs] & imm)
ORI	rt,rs,imm	OR Immediate (reg[rt] = reg[rt] | imm)
XORI	rs,rt,imm	XOR Immediate (reg[rt] = reg[rt] ^ imm)
LUI	rt,imm	Load Upper Immediate (upper[reg[rt]] = lower[imm])
SLTI	rt,rs,imm	Set Less Than Immediate (reg[rt] = reg[rs] < imm)
SLTIU	rt,rs,imm	Set Less Than Immediate Unsigned
BC0T	label	Branch Coprocessor 0 True
BC1T	label	Branch Coprocessor 1 True
BC0F	label	Branch Coprocessor 0 False
BC1F	label	Branch Coprocessor 1 False
BEQ	rs,rt,label	Branch on Equal (if reg[rs] == reg[rt] goto label)
BGEZ	rs,label	Branch on Greater Than or Equal Zero
BGEZAL	rs,label	Branch Greater Than or Equal and Link
BGTZ	rs,label	Branch Greater Than Zero
BLEZ	rs,label	Branch Less Than or Equal Zero
BLTZAL	rs,label	Branch Less Than Zero and Link
BLTZ	rs,label	Branch Less Than Zero
BNE	rs,rt,label	Branch Not Equal Zero (if reg[rs] != reg[rt] goto label)
LB	rt,address	Load Byte (reg[rt] = byte[address])
LBU	rt,address	Load Byte Unsigned
LH	rt,address	Load Halfword (reg[rt] = halfword[address])
LHU	rt,address	Load Halfword Unsigned
LW	rt,address	Load Word (reg[rt] = word[address])
LWC0	rt,address	Load Word Coprocessor 0
LWC1	rt,address	Load Word Coprocessor 1

LWL	rt,address	Load Word Left
LWR	rt,address	Load Word Right
SB	rt,address	Store Byte (byte[address] = reg[rt])
SH	rt,address	Store Halfword (halfword[address] = reg[rt])
SW	rt,address	Store Word (word[address] = reg[rt])
SWC0	rt,address	Store Word Coprocessor 0
SWC1	rt,address	Store Word Coprocessor 1
SWL	rt,address	Store Word Left
SWR	rt,address	Store Word Right
MFC0	rt,rd	Move From Coprocessor 0
MFC1	rt,rd	Move From Coprocessor 1
MTC0	rd,rt	Move To Coprocessor 0
MTC1	rd,rt	Move To Coprocessor 1
C.EQ.D	fs,ft	Compare Equal Double
C.EQ.S	fs,ft	Compare Equal Single
C.LE.D	fs,ft	Compare Less Than or Equal Double
C.LE.S	fs,ft	Compare Less Than or Equal Single
C.LT.D	fs,ft	Compare Less Than Double
C.LT.S	fs,ft	Compare Less Than Single
ADD	rd,rs,rt	Addition (reg[rd] = reg[rs] + reg[rt])
ADDU	rd,rs,rt	Addition Unsigned
AND	rd,rs,rt	Logical AND (reg[rd] = reg[rs] & reg[rt])
NOR	rd,rs,rt	Logical NOR (reg[rd] = ~(reg[rs] | reg[rt]))
OR	rd,rs,rt	Logical OR (reg[rd] = reg[rs] | reg[rt])
SLL	rd,rt,shamt	Shift Left Logical (reg[rd] = reg[rt] << shamt)
SLLV	rd,rt,rs	Shift Left Logical Variable (reg[rd] = reg[rt] << reg[rs])
SRA	rd,rt,shamt	Shift Right Arithmetic (reg[rd] = reg[rt] >>> shamt)
SRAV	rd,rt,rs	Shift Right Arithmetic Variable
SRL	rd,rt,shamt	Shift Right Logical (reg[rd] = reg[rt] >> shamt)
SRLV	rd,rt,rs	Shift Right Logical Variable
SUB	rd,rs,rt	Subtract (reg[rd] = reg[rs] – reg[rt])
SUBU	rd,rs,rt	Subtract Unsigned
XOR	rd,rs,rt	Logical Exclusive OR (reg[rd] = reg[rs] ^ reg[rt])
SLT	rd,rs,rt	Set Less Than (reg[rd] = (reg[rs] < reg[rt]))
SLTU	rd,rs,rt	Set Less Than Unsigned
ABS.D	fd,fs	Absolute Value Double (reg[fd] = |reg[fs]|)
ABS.S	fd,fs	Absolute Value Single
ADD.D	fd,fs,ft	Addition Double (reg[fd] = reg[fs] + reg[rt])
ADD.S	fd,fs,ft	Addition Single
CVT.D.S	fd,fs	Convert Single to Double
CVT.D.W	fd,rs	Convert Word to Double
CVT.S.D	fd,fs	Convert Double to Single
CVT.S.W	fd,rs	Convert Word to Single
CVT.W.D	rd,fs	Convert Double to Word
CVT.W.S	rd,fs	Convert Single to Word
DIV.D	fd,fs,ft	Divide Double
DIV.S	fd,fs,ft	Divide Single
MOV.D	fd,fs	Move Double (reg[rd] = reg[fs])
MOV.S	fd,fs	Move Single
MUL.D	fd,fs,ft	Multiply Double (reg[rd] = reg[rs] * reg[rt])
MUL.S	fd,fs,ft	Multiply Single
NEG.D	fd,fs	Negate Double (reg[rd] = ~reg[rs])
NEG.S	fd,fs	Negate Single
SUB.D	fd,fs,ft	Subtract Double (reg[fd] = reg[fs] – reg[ft])
SUB.S	fd,fs,ft	Subtract Single

PSEUDO INSTRUCTIONS

NOP		No Operation
DIV	rdest,rsrc1,rsrc2	Divide (reg[rdest] = reg[rsrc1] / reg[rsrc2])
DIVU	rdest,rsrc1,rsrc2	Divide Unsigned
ABS	rdest,rsrc	Absolute Value (reg[rdest] = |reg[rsrc]|)
MUL	rdest,rsrc1,rsrc2	Multiply (reg[rdest] = reg[rsrc1] * reg[rsrc2])
MULO	rdest,rsrc1,rsrc2	Multiply Overload
MULOU	rdest,rsrc1,rsrc2	Multiply Overload Unsigned
NEG	rdest,rsrc	Negate (reg[rdest] = -reg[rsrc])
NEGU	rdest,rsrc	Negate Unsigned
NOT	rdest,rsrc	Logical NOT (reg[rdest] = ~reg[rsrc])
ROL	rdest,rsrc1,rsrc2	Rotate Left
ROR	rdest,rsrc1,rsrc2	Rotate Right
LI	rdest,imm	Load Immediate (reg[rdest] = imm)
SEQ	rdest,rsrc1,rsrc2	Set If Equal

SEQI	rdest,rsrc1,imm	Set If Equal Immediate
SGE	rdest,rsrc1,rsrc2	Set If Greater Than or Equal
SGEU	rdest,rsrc1,rsrc2	Set If Greater Than or Equal Unsigned
SGT	rdest,rsrc1,rsrc2	Set If Greater Than
SGTU	rdest,rsrc1,rsrc2	Set If Greater Than Unsigned
SLE	rdest,rsrc1,rsrc2	Set If Less Than or Equal
SLEU	rdest,rsrc1,rsrc2	Set If Less Than or Equal Unsigned
SNE	rdest,rsrc1,rsrc2	Set If Not Equal
SNEI	rdest,rsrc1,imm	Set If Not Equal Immediate
B	label	Branch
BEQZ	rsrc,label	Branch If Equal Zero
BGE	rsrc1,rsrc2,label	Branch If Greater Than or Equal
BGEU	rsrc1,rsrc2,label	Branch If Greater Than or Equal Unsigned
BGT	rsrc1,rsrc2,label	Branch If Greater Than
BGTU	rsrc1,rsrc2,label	Branch If Greater Than Unsigned
BLE	rsrc1,rsrc2,label	Branch If Less Than or Equal
BLEU	rsrc1,rsrc2,label	Branch If Less Than or Equal Unsigned
BLT	rsrc1,rsrc2,label	Branch If Less Than
BLTU	rsrc1,rsrc2,label	Branch if Less Than Unsigned
BNEZ	rsrc,label	Branch Not Equal Zero
LA	rdest,address	Load Address
MOVE	rdest,rsrc	Move
L.D	fdest,address	Load Double
L.S	fdest,address	Load Single
S.D	fdest,address	Store Double
S.S	fdest,address	Store Single

For reference, System Call Parameter are provided below (system call parameter specified in register $v0):

n	**DESCRIPTION**
1	Print Integer Stored in Register $a0
2	Print Float Stored in Register $f12
3	Print Double Stored in Register $f12
4	Print String Starting at Memory Address Specified by Register $a0
5	Read Integer, Store in Register $v0
6	Read Float, Store in Register $f0
7	Read Double, Store in Register $f1
8	Read String, Store in Buffer Starting at Address $a0 of Length $a1
9	System Break (not supported in Release1)
10	Exit Program (used to specify the end of the current program)

The Assembly Process

The assembly process typically requires multiple passes. If any labels are defined, then two passes are required. If any pseudo or synthetic instructions are used, then two passes are also required. Most programs use both labels and pseudo/synthetic instructions. Among other benefits, labels are used to give assembly programs much greater clarity. The pseudo instructions are as listed above, and are essentially macros that represent a sequence of more fundamental instructions. Synthetic instructions, however, are more difficult to explain.

Synthetic instructions are inserted into the program during the assembly process. They are intended to correct assembly code that is otherwise incorrect or meaningless. A typical example is the use of the LW (Load Word) instruction. Often, assembly code includes an instruction such as "LW $t0, x", where it is intended to load register $t0 with the value at the memory address designated by label x. However, used in this form, the LW instruction can only load from the first 16-bit region of the address space (that is, $0000 to $FFFF). Typically, program data does not reside in this region. More than likely, the label x refers to some 32-bit address. To resolve the problem, the assembler can use a synthetic instruction and modify the specified LW instruction as follows:

```
LUI $at, hi16(x)
LW $t0, lo16(x)($at)
```

While both pseudo and synthetic instructions provide tremendous convenience for programmers, inserting instructions increases the distance between labels. As a result, jump targets and branch distances can be difficult to pre-determine. This is another motivation for the use of labels, which allow the assembler to automatically determine this information using two passes. The first pass translates any pseudo instructions and inserts any necessary synthetic instructions. At the same time, symbolic assembly instruction are converted into their machine code form. Literal label and target addresses are determined during the second pass, and instructions that use these labels are translated into their final machine code form. The final machine code form is then output to a file, to be used by the simulator.

The JIMS assembler uses a proprietary object code format, which is easier for debugging purposes. One can readily view the object code and verify manually if it is correct or not. This format is described in a separate document associated with JIMS.

Sample MIPS Programs

This section includes two sample MIPS programs: `SQRT.S` (square root finding program) and `FACTOR.S` (naïve factoring program). These programs, among many others, were used as test programs during the development of JIMS.

SQRT.S

```
# Square root finding program.

# double DELTA = 1.0E-8;
# double n = number to find square root of;
# double guess = initial guess;  (x / 2.0)
# while (true) {
#   if ( abs(n-guess^2) <= DELTA) )
#     break;
#   output guess
#   guess = 0.5 * (n / guess + guess);
# }
# output sqrt(n) == guess

        .data   0x10000000
# n == the number you want to find the square root of
n:      .double 999435678123443.5
# DELTA == the amount of desired precision (larger value
#          is less precise, but faster)
DELTA:  .double 0.00000001

# guess == the initial guess, then holds the final guess
guess:  .double 0.0

# two == constant used during the initial guess
two:    .double 2.0

# half == constant used when calculating the next guess
half:   .double 0.5

# some user friendly output strings
nline:  .asciiz "\n"
final:  .asciiz "\nThe sqrt is "

        .text   0x00400000
main:
        l.d     $f0, n          # $f0 = n
        l.d     $f2, two        # $f2 = 2.0
        l.d     $f6, DELTA      # $f6 = DELTA
        l.d     $f10,half       # $f10 = 0.5

# prepare the initial guess (store at MEM[guess])
        div.d   $f8, $f0, $f2   # $f8 = $f0 / $f2 == (n / 2.0)
        s.d     $f8, guess      # $f8 = the initial guess

        l.d     $f14, guess     # $f14 = previous guess
        l.d     $f4, guess      # let $f4 = current guess
loop:


# Perform the guard of the while loop:  abs(n-guess^2) <= DELTA

#   1) square the guess
        mul.d   $f4, $f4, $f4   # $f4 = $f4 * $f4 == guess^2

#   2) n - guess^2
        sub.d   $f4, $f0, $f4   # $f4 = $f0 - $f4 == (n - guess^2)
```

```
#   3) abs result
        abs.d   $f4, $f4        # $f4 = abs($f4) == abs(n-guess^2)

#   4) result > DELTA
        c.le.d  $f4, $f6        # if ( abs(n-guess^2) <= DELTA ) )
        bc1t    done

# Output the guess
        l.d     $f12, guess     # reg[$f12] == double to print
        li      $v0, 3          # 3 == print double
        syscall

# adjust the value of guess
#   1) (n/guess)
        div.d   $f4, $f0, $f8   # $f4 = $f0 / $f8 == (n/guess)
#   2) (n/guess) + guess
        add.d   $f4, $f4, $f8   # $f4 = $f4 + $f8 == (n/guess) + guess
#   3) (n/guess)+guess * 0.5
        mul.d   $f4, $f4, $f10  # $f4 = $f4 * $f10 == (n/guess)+guess * 0.5
#   4) store the guess
        s.d     $f4, guess      # guess = $f4 (the new guess)
        l.d     $f8, guess      # let $f8 == copy of new guess

        c.eq.d  $f8, $f14       # if new guess == old guess, exit
        bc1t    done

        l.d     $f14, guess     # set the old guess to the new guess


        j       loop

done:
# output the final guess
        li      $v0, 4          # 4 == print string
        la      $a0, final
        syscall

        l.d     $f12, guess     # reg[$f12] == double to print
        li      $v0, 3          # 3 == print double
        syscall

        mul.d   $f12, $f12, $f12 # $f12 = $f12 * $f12 == answer ^ 2

        li      $v0, 3          # 3 == print double
        syscall

        li      $v0, 10
        syscall
```

FACTOR.S

```
# Very-Highly-Naive Factoring Algorithm
# int n = integer;
# for (int x = 0; x <= (n/2); x++) {
#   for (int y = 0; y <= n; y++) {
#     if (x * y == n)
#       System.out.println(x + " " + y);
#     }
#   }
# }

        .data
n:      .word   100
space:  .asciiz " "
nline:  .asciiz "\n"

        .text   0x00400000
main:
        lw      $t7, n          # let reg[$t7] == n (number to factor)
```

```
# prepare outter-loop
        li      $t0, 0          # let reg[$t0] == x (outter loop variable)
        move    $t1, $t7        # $t1 = $t7 == n
        div     $t1, $t1, 2     # let $t1 == (n/2)

outter:
        bgt     $t0, $t1, done

# prepare inner-loop
        li      $t2, 0          # let reg[$t2] == y (inner loop variable)
        move    $t3, $t7        # let $t3 = $t7 == n

inner:
        mul     $t4, $t0, $t2   # let $t4 = $t0 * $t2 == x * y
        bne     $t4, $t7, skip

# print x and y
        move    $a0, $t0        # print $t0 (==x)
        li      $v0, 1
        syscall

        la      $a0, space      # print a space
        li      $v0, 4
        syscall

        move    $a0, $t2        # print $t2 (==y)
        li      $v0, 1
        syscall

        la      $a0, nline      # print a nline
        li      $v0, 4
        syscall

skip:
        addi    $t2, $t2, 1     # y++
        ble     $t2, $t3, inner

        addi    $t0, $t0, 1     # x++
        j       outter

done:
        li      $v0, 10         # exit
        syscall
```

The MIPS Simulator

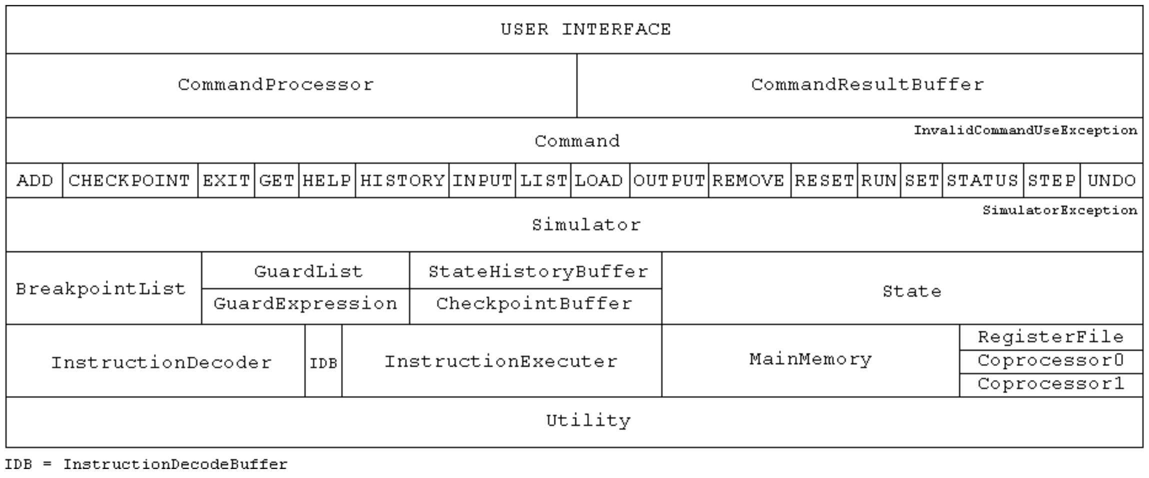


Figure A.2: Layout of JIMS Simulator Source Code

The abstract layout of the JIMS Simulator source is shown in Figure A.2. The simulator is organized into eight total layers, however these layers can be described as three sets of domains: (1) The User Interface Domain, which is represented only by the top layer of the JIMS Simulator source. (2) The Command Interface Domain, represented by layers 2, 3, and 4 of the simulator source. (3) the Simulator Domain, represented by all of the remaining layers (5 through 8) of the simulator source.

The Command Interface Domain provides a standard interface between the simulator and the user interface. The command interface provides a well-defined protocol that allows the user interface to treat the simulator as a server. The benefit of this organization is that multiple user interfaces can be developed, without requiring any changes to the Simulator Domain. Alternatively, one can ignore the Command Interface Domain completely, and develop a user interface that manages the simulator directly. Generally this would be done for performance reasons. However, the user interface and simulator core source would be tightly coupled together. The User Interface and Command Interface Domain are both described in a separate document. For our purposes, we focus on the Simulator Domain.

The Simulator Core

The simulator core consists of the bottom four layers of Figure A.2. The bottom layer, `Utility`, simply contains various useful functions that do not need to be discussed. The more interesting portions of the simulator are discussed as follows.

The `InstructionDecoder` and `InstructionExecuter` are, as should be obvious, used to decode and execute instructions. The instruction decoder simply retrieves the instruction at the current program counter, then extracts all the possible data contained in the instruction (opcode, function code, source register, destination register, shift amount, etc.). This data is stored in an `InstructionDecodeBuffer` which is later sent to the `InstructionExecuter` during the execution stage.

For the most part, the `Simulator` layer is responsible for managing state change records and checkpoints (such as determining when to create a checkpoint, etc.). The state history data itself is actually stored in the `StateHistoryBuffer` and `CheckpointBuffer` modules. However, the `Simulator` does not modify state values directly.

All state values are guarded by the `State` sub-layer. As a result, the `Simulator` actually makes a request to change a state value by using the `State` layer. The request is always granted, however this policy allows the `State` layer itself to record all changes in state values, which facilitates the creation of state records (used for state history recording). For JIMS, state values include main memory, registers, and the registers of coprocessor 0 and coprocessor 1 (which are each represented accordingly as separate modules below the `State` layer).

Highlights of Simulator Source Regarding State History Recording

The remainder of this appendix includes highlights of the main `Simulator` source code. These highlights demonstrate the implementation of the reversible execution feature of JIMS. The functions highlighted, followed by a brief description, are as follows:

- `performCycle()`: Used to perform one cycle of execution for the JIMS simulator. Notice that the check that determines if a checkpoint should be created is near the beginning of this function. Furthermore, notice how the state change record is handled at the end of the function.
- `bTimeToMakeCheckpoint()`: Uses the WUA approach described in Chapter 4 to determine if a checkpoint should be created during the current cycle.
- `makeCheckpoint()`: A wrapper for the doCreateCheckpoint(), though notice the steps that are done following the creation of the checkpoint.

- `iGetNextCheckpointByteSize()`: Estimates the size of the next checkpoint, as described in Chapter 3.
- `doCreateCheckpoint()`: Performs the actual work of creating a checkpoint. Notice how the code is very similar to the iGetNextCheckpointByteSize() function.
- `performCheckpointCulling()`: Apples the Checkpoint Culling Algorithm to the current set of checkpoints.
- `iPerformUndo(long lUndoDistance)`: Shows how state change records, checkpoints, and re-execution are all used together to provide reversible execution.

```
private void performCycle() throws SimulatorException {
// This is the primary method responsible for simulating
//   the CPU cycles.  This method should only be called
//   from the iPerformStep(int) method.

  StateHistoryBuffer stateHistoryBuffer = null;
  // Keep stateHistoryBuffer == null to disable state change recording.

  if (bHistoryRecordingEnabled) {
    if (bTimeToMakeCheckpoint()) {
      makeCheckpoint();
      // Since we just performed a checkpoint, we do not need to
      //   record state changes for this cycle.  This means
      //   that stateHistoryBuffer should remain null.
      performCheckpointCulling();

    } else {
      if (lCycleIndex > lLastCheckPointCycleIndex) {
        // We only want to record state changes if the current
        //   cycle is past (greater than) that of the last checkpoint.
        // NOTE:
        //   If (lCycleIndex == lLastCheckpointCycleIndex then
        //     a checkpoint was created on the last cycle.
        //   If (lCycleIndex < lLastCheckpointCycleIndex then
        //     there is a bug in the system.

        stateHistoryBuffer = new StateHistoryBuffer(lCycleIndex);
      }
    }
  }

  state.setStateHistoryBuffer(stateHistoryBuffer);

  // -- BEGIN CYCLE ----------------------------------------

  // INSTRUCTION FETCH
  IF();
  // INSTRUCTION DECODE
  ID();
  // EXECUTE
  EX();

  // EXCEPTION-HANDLER
  if (iExceptionCode != EXCEPTION_NONE) {

    --- PERFORM INTERNAL EXCEPTION HANDLING IF NECESSARY (code not shown) ---

  }

  // Increment the cycle counter.
  incrementCycleIndex();

  // -- END CYCLE ------------------------------------------

  if (bHistoryRecordingEnabled && (stateHistoryBuffer != null) ) {
    // Push the state history buffer into the state history vector,
    //   then disable the recording of state changes.
    addToStateHistoryBuffer(stateHistoryBuffer);
    state.setStateHistoryBuffer(null);
  }

}  // end method performCycle()

private boolean bTimeToMakeCheckpoint() {
  // This method implements the heuristic that determines
  //   when a checkpoint is to be created.
  // If it is determined that a checkpoint should be created,
  //   then this method returns TRUE.  Otherwise, it returns FALSE.

  boolean bResult = false;
  int iCheckpointByteSize = iGetNextCheckpointByteSize();
  if ( (iCheckpointCreationFactor * iCheckpointByteSize) <= iStateHistoryByteSize ) {
    bResult = true;
  }
  return bResult;
}  // end method bTimeToMakeCheckpoint()

public void makeCheckpoint() {
  // Create a checkpoint that records the current state
  //   of the simulator.
  cpb = doCreateCheckpoint();
  vCheckpoint.add(cpb);

  // Clear the current state history vector.
  clearStateHistoryBuffer();

  // Mark which cycle the checkpoint was created.  This is
  //   used to determine when state history recording
  //   should start again.
  lLastCheckPointCycleIndex = lCycleIndex;

}  // end method makeCheckpoint()

private int iGetNextCheckpointByteSize() {
  // This method returns an approximation to the size of the
  //   next checkpoint (in number of bytes).

  int iResult = 0;

  // Add all GPR registers that have a non-zero value.
  iResult += 2;  // == length("R ")
  for (int i = 0; i < RegisterFile.NUM_REGS; i++) {
    int iRegValue = state.iGetRegister(i);
    if (iRegValue != 0) {
      iResult += 12;  // == length(i+"="+HexPadded(iRegValue,8)+" ")
    }
  }

  // Add all CP0 registers that have a non-zero value.
  iResult += 2;  // == length("0 ")
  for (int i = 0; i < Coprocessor0.NUM_REGS; i++) {
    int iRegValue = state.iGetCP0Register(i);
    if (iRegValue != 0) {
      iResult += 12;  // == length(i+"="+HexPadded(iRegValue, 8)+" ")
    }
  }

  // Add all CP1 registers that have a non-zero value.
  iResult += 2;  // == length("1 ");
  for (int i = 0; i < Coprocessor1.NUM_REGS; i++) {
    int iRegValue = state.iGetCP1Register(i);
    if (iRegValue != 0) {
      iResult += 12;  // == length(i+"="+HexPadded(iRegValue,8)+" ")
    }
  }

  // Add modified memory values.
  Vector v = state.vGetModifiedAddresses();
  iResult += 11;  // == length("M AABBCCDD "), address
  iResult += v.size() * 3;  // 3 == length("XX "), hex values

  return iResult;
}  // end method iGetNextCheckpointSize()

private CheckpointBuffer doCreateCheckpoint() {

  CheckpointBuffer cpb = new CheckpointBuffer(lCycleIndex);

  StringBuffer sb = null;

  // Add all GPR registers that have a non-zero value.
  sb = new StringBuffer(CHECKPOINT_GPR + " ");
  for (int i = 0; i < RegisterFile.NUM_REGS; i++) {
    int iRegValue = state.iGetRegister(i);
    if (iRegValue != 0) {
      sb.append(i + "=" + Utility.sAsHexPadded(iRegValue, 8) + " ");
    }
  }
  cpb.addStateValue(sb.toString());

  // Add all CP0 registers that have a non-zero value.
  sb = new StringBuffer(CHECKPOINT_CP0 + " ");
  for (int i = 0; i < Coprocessor0.NUM_REGS; i++) {
    int iRegValue = state.iGetCP0Register(i);
    if (iRegValue != 0) {
      sb.append(i + "=" + Utility.sAsHexPadded(iRegValue, 8) + " ");
    }
  }
  cpb.addStateValue(sb.toString());

  // Add all CP1 registers that have a non-zero value.
  sb = new StringBuffer(CHECKPOINT_CP1 + " ");
  for (int i = 0; i < Coprocessor1.NUM_REGS; i++) {
    int iRegValue = state.iGetCP1Register(i);
    if (iRegValue != 0) {
      sb.append(i + "=" + Utility.sAsHexPadded(iRegValue, 8) + " ");
    }
  }
  cpb.addStateValue(sb.toString());

  // Add modified memory values
  Vector v = state.vGetModifiedAddresses();
  if ((v != null) && (v.size() > 0)) {
    // There is at least one modified address.

    Enumeration e = v.elements();

    // Get the first element, so we know which address we
    //   are starting at.
    int i = ((Integer)e.nextElement()).intValue();
    int iAddress = i;

    sb = new StringBuffer(CHECKPOINT_MEM + " " +
      Utility.sAsHexPadded(iAddress, 8) + " ");

    do {

      // Get the memory value at the current address, and add
      //   it to the string buffer.
      byte value = state.loadByte(iAddress);
      sb.append(Utility.sAsHexPadded(value, 2) + " ");

      if (!e.hasMoreElements()) {
        // No more addresses to process.
        break;
      }

      // Increment the address counter.  We expect the
      //   next address to match this incremented value.
      iAddress++;

      // Get the next address.
      i = ((Integer)e.nextElement()).intValue();

      if (i != iAddress) {
        // The new address does not match the address that
        //   we expected (i.e. we have a disjoint set).
        //   Add the current string buffer value to the
        //   command response, and prepare a new string buffer.
        cpb.addStateValue(sb.toString());

        iAddress = i;
        sb = new StringBuffer(CHECKPOINT_MEM + " " +
          Utility.sAsHexPadded(iAddress, 8) + " ");
      }

    } while (true);

    cpb.addStateValue(sb.toString());

  }  // end if-modified memory values
  return cpb;
}  // end method doCreateCheckpoint()

private void performCheckpointCulling() {

// t == lCycleIndex (current cycle index)
// C == vCheckpoint (current set of checkpoints)

  if (lCycleIndex <= 0)
    return;

  int iChkIndex = 0;  // Index into vCheckpoint vector.

  // Some constant values to experiment with:
  //   LN_1.5 = 0.17609
  //   LN_2 = 0.6931471805599453
  //   LN_3 = 1.0986122886681096
  //   LN_4 = 1.3862943611198906
  //
  // NOTE:
  //   logn(n,x) = log(x) / log(n), where log(k) == log base e of k

  // Let k == floor( log base 2 of t )
  int K = (int)( StrictMath.log((double)lCycleIndex) / 0.6931471805599453 );

  for (int k = K; k > 0; k--) {

    int z = (int)StrictMath.pow(2, k);
    int iUpper = (int)lCycleIndex - z;               // Upper = t - 2^k
    int iLower = (int)lCycleIndex - (z << 2) + 1;    // Lower = t - 2^(k+1) + 1

    if (iLower < 0)
      iLower = 0;

    // Since z was already declared, but is no longer used,
    // we re-use it also as a flag to indicate when we first
    // encounter a checkpoint in this region (rather than using
    // one more variable, like a boolean).
    z = -1;

    do {

      // Get the time index of the checkpoint specified
      //   by the current checkpoint index.
      CheckpointBuffer cp = (CheckpointBuffer)vCheckpoint.elementAt(iChkIndex);
      int iTimeIndex = (int)cp.lGetCycleIndex();

      if ( (iTimeIndex >= iLower) && (iTimeIndex <= iUpper) ) {
        if (z == -1) {
          // This is the first checkpoint encountered in this
          //   region.  Keep it and increment the checkpoint
          //   index to the next checkpoint.
          iChkIndex++;
          z = 0;  // Clear the "first encountered" flag

        } else {
          // We already encountered the first checkpoint in ths
          //   region, therefore this checkpoint is marked
          //   to be deleted.  Go ahead and do so.
          vCheckpoint.removeElementAt(iChkIndex);
        }
      } else {
        // We have encountered a checkpoint whose time index
        //   is outside this region.  This means there are no
        //   more checkpoints that can be in this region, so
        //   move to the next region.
        break;
      }
    } while(true);
  }  // end for k == K down to 1
} // end method performCheckpointCulling()

public int iPerformUndo(long lUndoDistance) {
/*

  Reverse lUndoDistance number of instructions.

  RETURNS
    0 = SUCCESS
   -1 = NO MORE STATE HISTORY CHANGE
   -2 = ERROR APPLYING STATE HISTORY CHANGE
   -3 = ERROR APPLYING CHECKPOINT STATE
   -4 = ERROR RE-EXECUTING FROM LAST CHECKPOINT
   -5 = EXCEPTION ENCOUNTED WHILE UNDO FROM CHECKPOINT
   -6 = OUTPUT ERROR WHILE RE-EXECUTING INSTRUCTIONS

*/

  int iResult = 0;

  if (!bHistoryRecordingEnabled) {
    return -1;
  }

  int iStateHistorySize = vStateHistory.size();
  int iCheckpointSize = vCheckpoint.size();

  sbOutputFromUndo = null;  // Clear any undo output from any previous undo.

  if ( (iStateHistorySize == 0) && (iCheckpointSize == 0) ) {
    // There is no history information.  Either we are in
    //   the initial state, or all the history information
    //   was cleared.  Eitherway, we have no information
    //   to perform the undo with.
    return -1;
  }

  long lTargetCycle = lCycleIndex - lUndoDistance;
  // The target cycle is what execution cycle we are trying
  //   to get to, which is simply the current cycle
  //   minus the number of cycles we want to undo.

  if (lTargetCycle < 0) {
    // We can not go to a state that is earlier than
    //   the initial state.
    lTargetCycle = 0;
  }

  if ( (iStateHistorySize == 0) || (lUndoDistance > iStateHistorySize) ) {
```

```
// There is no state history information.  Or the distance
//   of the undo is larger than the size of the state history
//   buffer.  Regardless, perform the undo using the last checkpoint.

if (lUndoDistance > iStateHistorySize) {
  // Clear the state history, since all of the state changes
  //   recorded in it will not be used.  An earlier
  //   checkpoint will be used instead.
  clearStateHistoryBuffer();
}

if (iCheckpointSize > 0) {
  // Set the state to that of the last useful checkpoint
  //   that was created.

  CheckpointBuffer cpb = null;
  long lCheckpointCycleIndex = -1;
  // Find a checkpoint buffer to start from.
  try {
    do {
      cpb = (CheckpointBuffer)vCheckpoint.elementAt(iCheckpointSize-1);
      lCheckpointCycleIndex = cpb.lGetCycleIndex();
      if (lCheckpointCycleIndex > lTargetCycle) {
        // The checkpoint represents a state at a point in time
        //   AFTER the desired target cycle.  It would be better
        //   to examine the next checkpoint.

        vCheckpoint.removeElementAt(iCheckpointSize-1);
        iCheckpointSize--;
        if (iCheckpointSize <= 0) {
          // This is a safety check.  If no more checkpoints
          //   exists, then go back to the reset state.
          state.reset();
          cpb = null;
          break;
        }
      } else {
        break;
      }
    } while (true);

    if (lCheckpointCycleIndex == lTargetCycle) {
      // The previous cycle (iTargetCycle) was when the
      //   checkpoint was created.  We can remove the checkpoint,
      //   since if the user goes back further, the top checkpoint
      //   is no longer useful.  If the user steps forward,
      //   the checkpoint will be recreated.
      vCheckpoint.removeElementAt(iCheckpointSize-1);
      iCheckpointSize--;
    }

  } catch (Exception e) {
    cpb = null;
  }

  // We found no suitable checkpoint buffer.  Either an error occurred,
  //   or there is no more suitable undo information available.
  if (cpb == null) {
    return -1;  // NO MORE STATE HISTORY CHANGE
  }

  // Set the state back to the reset state.  All memory and
  //   register values should be defaulted to have the value 0.
  state.reset();

  lCycleIndex = lCheckpointCycleIndex;
  // Set the current cycle count to the time index stored
  //   in the checkpoint buffer.

  // Apply all of the state settings stored in the checkpoint.
  Vector v = cpb.vGetStateRecord();
```

```
      Enumeration e = v.elements();
      while (e.hasMoreElements()) {
        String s = (String)e.nextElement();
        int x = iApplyCheckpointSetting(s);
        if (x != 0) {
          // This is most likely the result of an
          //   internal bug in the simulator.
          return -3;
        }
      }
    } else {
      // Set the state to the reset state.
      state.reset();
    }

    // Disable any exceptions that triggered from the previously
    //   executed instruction.
    iExceptionCode = EXCEPTION_NONE;

    // *** RE-EXECUTION PHASE **************************************
    // Perform cycles until the cycle count reaches lTargetCycle,
    //   or an exception triggers.
    while (lCycleIndex < lTargetCycle) {

      try {
        int i = iPerformStep(lTargetCycle - lCycleIndex);
        switch (i) {
          case 0:
            // No Error
            break;
          case -1:  // = BREAKPOINT ENCOUNTERED (which one is set in iBreakpointIndex)
            // Ignore breakpoint?
            break;
          case -2:  // = GUARD ENCOUNTED (which one is set in iGuardIndex)
            // Ignore guard?
            break;
          default:  // = CRITICAL ERROR (out of memory?)
            throw new Exception();
        }
      } catch (SimulatorException e) {

        // SimulatorException triggered (invalid opcode, etc).
        //   Refer to EXCEPTION_CODE value.

        if (iExceptionCode == EXCEPTION_OUTPUT_WAITING) {
          // Some output was requested.  Store the output in a buffer for now.
          StringBuffer sbOutput = new StringBuffer();
          if (iReadOutput(sbOutput) == 0) {
            if (sbOutputFromUndo == null) {
              sbOutputFromUndo = new StringBuffer();
            }
            sbOutputFromUndo.append(sbOutput);
          } else {
            return -6;
          }
        } else {
          return -5;
        }

      } catch (Exception e) {

        // CRITICAL_ERROR (index out of range, null pointer --
        //   indicates either a bug in the simulator, or
        //   perhaps an out of memory condition).
        return -4;

      }
    }  // end while (performing cycles until reach iTargetCycle)
    // ************************************************************

  } else {
    // There is undo information in the state history buffer.
    //   These are performed first, before checkpoints.

    do {

      StateHistoryBuffer shb = null;
      // Get the last state history buffer instance.
      try {
        shb = (StateHistoryBuffer)vStateHistory.elementAt(iStateHistorySize-1);
        vStateHistory.removeElementAt(iStateHistorySize-1);
        iStateHistorySize--;
      } catch (Exception e) {
        shb = null;
      }
      if (shb == null) {
        // Either an error occurred, or there was no state history
        //   buffer information (meaning there is no undo information).
        return -1;  // NO MORE STATE HISTORY CHANGE
      }

      // Apply the state values stored in the change buffer.
      //   These must be applied in reverse, so that the
      //   state is returned to its earliest state per this
      //   change buffer.
      Vector v = shb.vGetStateHistory();
      for (int i = v.size()-1; i >= 0; i--) {
        String s = (String)v.elementAt(i);
        int x = iApplyStateSetting(s);
        if (x != 0) {
          // This is most likely the result of an
          //   internal bug in the simulator.
          return -2;
        }
      }
      // It is assumed that each state history element represents
      //   one cycle of execution (because at least $PC will change
      //   on every cycle).  Therefore, undoing this state history
      //   change decreases the cycle count by one.  But to be certain,
      //   use the cycle index stored in the SHB instead.
      lCycleIndex = shb.lGetCycleIndex();

      // Decrease the size of the state history by the size of the
      //   state history buffer that was just applied.  This is for
      //   performance reasons, so that the size of the history buffer
      //   does not need to be directly re-calculated each cycle.
      iStateHistoryByteSize -= shb.iGetSize();

    } while (lCycleIndex > lTargetCycle);

    // Disable any exception that was caused by the previous command.
    //   If we don't, then the instruction will be undone, but a pending
    //   exception will be waiting (e.g. for input) when it shouldn't be.
    iExceptionCode = EXCEPTION_NONE;

  }  // end state-history undo
  return iResult;
}  // end method iPerformUndo()
```

Simulated Checkpoint Culling

The following is a short Java Application that demonstrates the checkpoint culling process, described in Chapter 5. This program was used to generate Table 5.1.

```
public class SimulateCheckpointCulling {

  private static String sCheckpointVector = "X";
  private static final int MAX_TIME_INDEX = 50;
  private static final int LOG_BASE = 2;

  public static void main(String[] args) {

    for (int i = 0; i <= MAX_TIME_INDEX; i++) {
      sCheckpointVector += "X";  // Create new checkpoint.
      doCullCheckpoints(i);
    }

  }

  private static void doCullCheckpoints(int t) {

    if (t <= 0)
      return;

    int K = (int)(Math.log(t) / Math.log(LOG_BASE));  // log base b of t = ln(t) / ln(b)

    for (int k = K; k >= 0; k--) {

      int low = t - (int)Math.pow(LOG_BASE, k+1) + 1;
      int high = t - (int)Math.pow(LOG_BASE, k);

      if (low < 0)
        low = 0;

      boolean foundFirst = false;
      for (int i = low; i <= high; i++) {
        if (sCheckpointVector.charAt(i) == 'X') {
          if (foundFirst) {
            // Delete checkpoint i
            sCheckpointVector = sCheckpointVector.substring(0, i) +
              "-" + sCheckpointVector.substring(i+1);
          } else {
            foundFirst = true;
          }
        }
      }

      System.out.print(low + " to " + high + "\t");
    }
    System.out.println("\t" + sCheckpointVector);

  }  // end method doCullCheckpoints(int t)

}  // end class SimulateCheckpointCulling
```

LIST OF REFERENCES

[1] J. S. Plank, “An Overview of Checkpointing in Uniprocessor and Distributed Systems Focusing on Implementation and Performance,” Tech. Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, Tenn., 1997. http://citeseer.nj.nec.com/plank97overview.html (accessed 05/04/2001)

[2] R. Sosic, “History Cache: Hardware Support for Reverse Execution,” *Computer Architecture News*, 22(5):11-18, 1994. http://citeseer.nj.nec.com/sosic94history.html (accessed 05/04/2001)

[3] B. P. Miller and J-D Choi, “A Mechanism for Efficient Debugging of Parallel Programs,” In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, published in SIGPLAN Notices, pages 141-150, January 1989.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford, “An Execution-Backtracking Approach to Debugging,” *IEEE Software*, 8(3):21-26, May 1991.

[5] S. Feldman and C. B. Brown, “IGOR: A System for Program Debugging via Reversible Execution,” In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, SIGPLAN Notices, pages 112-123, January 1989.

[6] C. Demetrescu and I. Finocchi, “LEONARDO: A C Programming Environment for Reversible Execution and Software Visualization,” 1999. http://www.dis.uniroma1.it/~demetres/Leonardo (accessed 06/12/2001)

[7] B. Boothe, “Algorithms for Bidirectional Debugging,” Tech. Report USM/CS-98-2-23, Dept. of Computer Science, University of Southern Maine, Portland, ME, 1998.

[8] M. P. Frank, “Reversibility for Efficient Computing,” Ph.D. dissertation, University of Florida, 1999. http://www.cise.ufl.edu/~mpf/manuscript (accessed 07/10/2001)

[9] B. J. Ross, “Running Programs Backwards: The Logical Inversion of Imperative Computation,” *Formal Aspects of Computing*, 9:331-348, 1997. http://citeseer.nj.nec.com/ross98running.html (accessed 05/04/2001)

[10] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala, “Checkpointing and its Applications,” In *25th International Symposium on Fault-Tolerant Computing*, pages 22-31, Pasadena, CA, June 1995.
`http://citeseer.nj.nec.com/wang95checkpointing.html` (accessed 05/04/2001)

[11] D. Z. Pan and M. A. Linton, “Supporting Reverse Execution of Parallel Programs,” In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, published in SIGPLAN Notices, pages 124-129, January 1989.

[12] M. Beck, J. S. Plank, and G. Kingsley, “Compiler-Assisted Checkpointing,” Technical Report CS-94-269, University of Tennessee at Knoxville, December 1994.
`http://citeseer.nj.nec.com/173887.html` (accessed 05/04/2001)

[13] M. Ronsse, K. Bosschere, and J. C. Kergommeaux, “Execution Replay and Debugging,” Ghent University, 1999.

[14] R. Chow and T. Johnson, *Distributed Operating Systems and Algorithms*, Addison Wesley Longman, Inc., Berkeley, California, 1997.

[15] R.M. Balzer, “EXDAMS: EXtendable Debugging and Monitoring System,” In *Proc. Spring Joint Computer Conf.*, pages 567-589. AFIPS Press, Reston, VA, 1969.

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994.

[17] D. Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.

BIOGRAPHICAL SKETCH

Steve A. Lewis II was born in Lafayette, Indiana, in May 1978. In the following year, his parents moved the family to Gainesville, Florida. He attended P.K. Yonge Developmental Research School from August 1984 until June 1996 (K-12). Steve began his college career in August 1994 by a dual enrollment program, in which he attended both P.K Yonge and Santa Fe Community College simultaneously. He transferred to the University of Florida in August 1997, later receiving his Bachelor of Science degree in computer science from the College of Liberal Arts and Sciences in December 1999. During his undergraduate studies, he worked part time as a software programmer for Jenmar International and MindSolve Technologies. In January 2000, he began his graduate study in the Department of Computer and Information Science and Engineering at the University of Florida, for the Master of Science degree in computer engineering. He will receive the degree in August 2001. Soon thereafter, he will begin work with Lockheed Martin in Fort Worth, Texas.