1-1-2002

CBCA: balance cache checkpointing for superscalar processors

Adeel Israr
Iowa State University

CBCA: Balance Cache Checkpointing for Superscalar processors

by

Adeel Israr

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2002

Graduate College
Iowa State University

This is to certify that the master's thesis of

Adeel Israr

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

DEDICATION

To my kind and loving parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The reliability of future general-purpose processors (GPPs) is threatened by a combination of factors like shrinking transistor size, higher clock rates, and reduced supply voltages. It is predicted that the occurrence of soft errors will dramatically increase as these trends continue. It is necessary that the processors be covered from the effects of transient faults with predictable minimal impact on performance. Such a predictable impact would allow the processor to be employed in real-time applications. In this thesis, we propose a superscalar architecture based on checkpointing buddy cache architecture. This architecture keeps the execution of tasks free from soft errors by detecting the fault in the execution state and resetting the state of the processor to last known fault free state.

Our scheme minimizes the overhead of checkpointing by eliminating the unnecessary checkpoints. We added *checkpointing buddy cache* (CBC), a small fully associative cache, that works in parallel with the data cache. Whenever there is a need for replacing a dirty line from the data cache, then instead of writing it to main memory and taking a checkpoint, it is written to CBC. The checkpoint is taken when the CBC is filled with dirty lines. The results show that the number of checkpoints required for the *checkpointing buddy cache architecture* (CBCA) is 10% to 80% less than the conventional cache checkpointing architecture. The cache miss rate for our scheme is also reduced up to 26% as compared to base processor. The number of checkpoints and miss rate were found to decrease with the increase in the size of CBC. However, the access time for CBC of size 32 lines was found to be greater than the data cache, thus making CBC of size greater than 32 impractical. CBCA provides fault tolerant mechanism to protect the processor against soft errors, while providing task execution time similar to base processor.

CHAPTER 1. Introduction

1.1 Background

Throughout history, the human beings have made tools and devices that assist them to survive and to improve their performance. The computer is one of their latest inventions. The computers have found their use in a large varity of systems that assist humans. Their applications range from small toys to space stations. As humans are becoming more and more dependent on computer-based systems, increasing attention is being paid not only to the processors' computing speed, but also to their dependability as well as predictability of execution times.

1.1.1 Processor Speed

Architectural and technological innovations have enabled the development of powerful microprocessors that can execute several instructions concurrently at a very high clock-rate. With the availability of ample on-chip hardware resources, researchers are proposing aggressive micro-architectures that can execute large number of instructions every clock-cycle. Both density and cycle time of memory is improving every year, thus improving the overall speed of the system. This development is keeping pace with Moore's law, which states that computation speed and memory capacity double every 18 months. Technological advancements in general purpose processor (GPPs) and onchip memories have significantly increased the speed and density of these components. This tremendous improvement in the performance of computer is mainly due to decrease in feature size. This progress in computer performance is made while maintaining, or even increasing, the reliability of the individual components.

1.1.2 Dependability

Dependability is the quality of service provided by a particular system. Reliability, availability, safety, maintainability, performablity, and testability are examples of measures used to quantify the dependability of a system. Among these, reliability and availability are most frequently used dependability measures. *Reliability* is the probability of continuous operation over an interval of time, where as *availability* is the probability of the system being operational at any given instant.

Unstable environmental conditions can generate temporary hardware failures in computers, even if they are shipped with no design errors or manufacturing defects. These failures are termed as transient faults or soft errors. According to Siewiorek and Robert[11], transient faults are much more frequent, than any other kind of faults. An error is simply defined as an inappropriate change in the value of a signal (i.e. from high to low, or from low to high).

1.1.2.1 Sources of Soft error

In [1], it is shown that particles such as alpha and gamma, cosmic rays, and neutrons are the main causes of soft errors. These particles cause electric disturbances in the silicon and thus might change the state of transistors affected by them. The radioactive impurities present in the packaging material or the silicon are the major sources of these radiations. In spite of improved manufacturing techniques, these impurities cannot be removed 100%, so they still cause problem. The three major causes are described as follows [1]:

Alpha particles: Upon impact on the chip, they strike out the electrons from the silicon or other p-type or n-type impurity atoms.

Cosmic rays: The Sun being their primary source, they react with earth's atmosphere to produce many particles having high energy. These rays produce strong electrical disturbances at their point of impact on the chip.

High-energy neutron particles: They are primarily created by reaction of the cosmic rays with the atmosphere. These partials strike the molecular structure of the chip and produce high-energy alpha particles.

Transistor size is continuously shrinking and so is the voltage difference between voltage levels of logic "0" and "1" [2], but the flux of the above mentioned radiations remains unchanged. As a result these particles can inject errors on chip. The intensity of the radiations might be different at different points on the chip. An influx of these rays on the chip surface that in the past didn't affect the state of a chip would now cause the same effects that was formerly caused by the higher intensity radiation. High intensity radiation would now cause many more errors in the chip.

Soft errors do not necessarily translate into system failures, making it hard to measure their frequency of occurrence. The location and timing of an error determine the final effect. A given soft error may be inconsequential or may propagate for a certain time without affecting a computation. Since memories have always been more susceptible to errors than processor pipelines, earlier research has focused on estimating the soft error rate in the context of memory technology [3, 4, 5]. Error-correcting codes (ECCs) have been used to ensure memory reliability [4]. ECCs and memory scrubbing techniques can drastically reduce the probability of system failure. Currently, a given single-bit in a RAM is expected to flip only once in many billion hours of operation[25]. However, this error frequency will increase in the future because of the shrinking trends in the size of transistors.

The bit error rate of the processor is expected to increase even higher than in a memory chip due to higher complexity of the processor. At current operating speed, a processor experiences a bit-flip every 10 hours. Although not all soft errors result in a failure, the fact that even a single-bit error may cause fatal damage generates a need for error detection and recovery techniques for the system.

1.1.3 Predictability

Real time systems like flight control and health care, are hard real-time or mission critical systems. Failure to meet the deadlines of the tasks in these systems might result in loss of life and/or property. These systems require that the execution of the task must be fault free

[6], because any fault in the system during the execution of the task might lead to loss of life and/or property. To avoid this scenario these systems should be provided with timely response [6]. A hard, real time system that is not fault tolerant or doesn't provide timely execution of the tasks may lead to loss of life and/or property. While being fault tolerant, such systems should also provide a minimum performance (in terms of execution time) that is acceptable for the overall working of the application. Since the performance degradation in fault tolerant system is inevitable, the execution time of the task on a given processor should be predictable. This allows a real time scheduling algorithms [7, 8] to be able to schedule the task on the processor with better accuracy.

1.2 Proposed Solution

The number of the soft errors occurring during computation is much more than the permanent faults[11]. Whenever an error is detected during a program execution, it has to be restarted. This solution may very expensive in certain cases, as there is a large performance penalties associated with any re-execution. The penalties might be even worse for mission-critical real time systems where the delay of service might translate into loss of life and/or money. If a very long program is being executed on a computer system that takes many months to execute, then a single fault might waste the entire computation that was done over a long time. *Checkpointing* can be used to decrease the loss incurred due to soft errors. The processor achieves fault tolerance by periodically verifying execution state of the processor and upon finding it fault free, the state of the execution is saved in safe storage (presumably fault free). A program may restart from the saved state upon finding a fault. The process of saving the state is called checkpointing and the saved state is referred to as *checkpoint.* The process of restarting the process state to an earlier state is called *rollback-recovery.*

However, the process of checkpointing is not without any cost. The processor has to stop its execution while the state of the processor is being verified. If there are many checkpoints, then the amount of time the processor is doing useful work decreases. On the other hand, if the number of checkpoints is too small, then the amount of computation wasted, in case of fault, will be large. The time taken by the processor during checkpointing cannot be determined exactly before the execution. So if the number of checkpoints is very large, then the predictability of the execution time of the program decreases, making the processor not suitable for real time applications. Thus the number of checkpoints during a program execution should be *balanced.*

The purpose of our research is to develop and evaluate a new cache checkpointing architecture called *Checkpointing Buddy Cache architecture* (CBCA). This is a step towards the balanced checkpointing with predictability and reliability requirements. We add a small fully-associative cache, called *Checkpointing Buddy Cache* to work in parallel with the data-cache. The checkpointing buddy cache assists the processor in balancing the number of checkpoints.

The main memory is assumed to be a stable storage device for the processor's state. Therefore, the write operation must not be arbitrarily performed on the memory. Every time there is a requirement to replace a dirty line in conventional cache checkpointing architecture, a checkpoint has to be established. Checkpointing buddy cache architecture decreases the number of checkpoint by copying the replaced dirty line to a fully associative buffer called *checkpointing buddy cache*(CBC), instead of writing it to the memory. A checkpoint is only established when CBC is completely filled with dirty lines. The result shows that our scheme reduces the number of checkpoints by 10% to 80% for different benchmarks as compared to conventional cache checkpointing architecture. When the number of checkpoints is large, the predictability of the execution time of the tasks greatly decreases. Our scheme reduces the number of checkpoints, and therefore the execution time of the tasks executing on the processor with CBCA can be predicted with better accuracy.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present some previous work on cache checkpointing architecture. In Chapter 3 we develop and discuss checkpointing buddy cache architecture. Chapter 4 is devoted for the discussion of the basic operation of checkpointing buddy cache based superscalar architecture. In Chapter 5 we present the results of our simulation and in Chapter 6 concludes the thesis.

CHAPTER 2. Cache for Fault Tolerance

A variety of checkpointing schemes have been proposed in the literature. These schemes can be broadly classified into the following two categories on the basis of their assumed stable storage for checkpoint information.

1. Memory Checkpointing
2. Cache Checkpointing

Memory checkpointing schemes assume the hard disk to be a stable storage for checkpoint information. The execution state is assumed to be in the memory hierarchy (main memory, cache and registers) above the hard disk. On every checkpointing event the execution state is stored on the hard disk. In these schemes, the emphasis is on managing the state variables in the main memory, cache and registers, so that they are efficiently moved to and recovered from the stable storage (hard disk system) with minimum overhead.

Cache based checkpointing schemes assume that the main memory can be used as a stable storage for checkpoint information. Memory hierarchy (cache and registers) above the main memory is assumed to be the part of execution state. On every checkpointing event this information is saved in the main memory. In these schemes, the emphasis is on managing state variables in the cache and registers, so that they are efficiently moved to and recovered from the stable storage (main memory) with minimum overhead.

2.1 Recovery Cache

The use of cache for recovering from an error condition was first proposed by Lee et al. [21]. They used cache to automate a small portion of software rollback block scheme. The scheme that they proposed is invisible to the programmer. However the hardware (only cache), operating system and compiler have to be modified to incorporate it. They call the cache used in their system as *recovery cache.* The main purpose of the recovery cache is to record the recovery data so that error recovery can be provided for the variables of a program containing recovery blocks. Whenever the processor writes to memory variables, their old values as well as their addresses are stored. Subsequently, if the application detects a bug in the computation, then these stored variables are restored and another algorithm is applied to the information.

The cache checkpointing schemes can be further classified into two main categories on the basis of the underlying architecture.

1. Uniprocessor cache checkpointing

2. Share memory processor cache checkpointing

2.2 Uniprocessor Cache Checkpointing

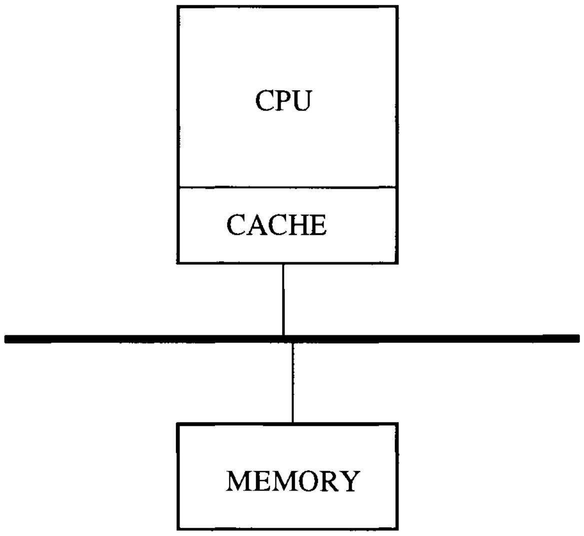


Figure 2.1 General block diagram of a Uni-processor

2.2.1 CARER

In [15], Hunt and Marions presented cache-aided rollback recovery (CARER) scheme and proposed a cache unit to assist checkpointing process. Their technique is invisible to the operating system, complier and programmer. In their work, each cache block is associated with one of the four status: invalid, clean, dirty and unchangeable. They keep checkpoint state in the main memory and use write-back algorithm for write operation. Checkpointing is primarily initiated when a dirty block has to be written back to the main memory. To keep the processor from staying idle at every checkpoint, the dirty lines, instead of flushing to main memory, are marked unchangeable. These unchangeable lines are flushed to the memory as the computation progresses after the checkpointing. Rollback to the correct system state is achieved by invalidating all the dirty lines and restoring the register file. This scheme is discussed in detail in Section 3.2.

2.2.2 Comparison-Based Cache-Aided Rollback Error Recovery

In [16], Lin presents another cache checkpoint architecture. He proposed to divide the cache into two parts and duplicate the resulting architecture. In each processor one part of the cache performs normal cache operations, while other cache stores the old checkpoint state that has to be flushed to the main memory. Like CARER [15], the checkpointing is primarily initiated when a dirty block has to be written back to the main memory. On every checkpoint the dirty lines in active caches of the two processors are compared. On a successful comparison the active cache becomes inactive cache and the inactive cache becomes active cache and the computation continues. The system uses the same memory bus to flush the dirty blocks from the inactive cache and read the data lines to the active cache. However, if there is a need for another checkpoint before the completion of the flush of inactive cache, then the normal checkpointing operations will have to wait until all the dirty lines from the inactive cache are flushed. Rollback to the correct system state is achieved by invalidating all the dirty lines in the active cache and restoring the register file.

2.2.3 Write-through cache scheme

Unlike CARER [15] and Lin [16], Rana E. Ahmed [22] has proposed to use write-through algorithm for write operation. First outcome of this assumption is that a checkpoint primarily has to be established every time a store operation is performed. On every checkpoint the word being stored is validated and register file is copied to the backup register file. Whenever an error is detected, the rollback is achieved by coping the backup register file to the active or working register file. Rana's cache checkpointing scheme [22] decreases the time required for checkpointing and the time required for the recovery. However, writing through the memory hierarchy every time a store is executed heavily affects system's performance.

2.3 Shared memory, cache checkpointing

In this section we investigate some work done on shared memory processors cache checkpointing. Shared memory processors (Figure 2.2) consists of multiple processors connected to each other through the shared memory by either a shared bus or an interconnection network. Each processor has its own private cache. Unlike a uniprocessor, shared memory processor can simultaneously execute many processes of the same program. Inter-process communication is achieved by accessing shared variables. In such a system, a processor can directly read from a modified line of another processor's cache. Several scheme using this paradigm have been proposed as discussed below.

2.3.1 Recovery Stack based cache checkpointing

The cache checkpointing scheme proposed by Wu et. al. [26], is based upon the CARER scheme for a uniprocessor. There are two main instances for which a process has to establish a checkpoint. The first is similar to uniprocessor checkpointing, i.e. whenever a dirty line is being replaced back to the shared memory. The second instance occurs when another processor tries to read a dirty line from a processor's cache.

To avoid serial marking (unchangeable) of the cache on every checkpoint, Wu et. al., use "checkpoint identifiers". These are k-bit memory elements present with every cache line. A

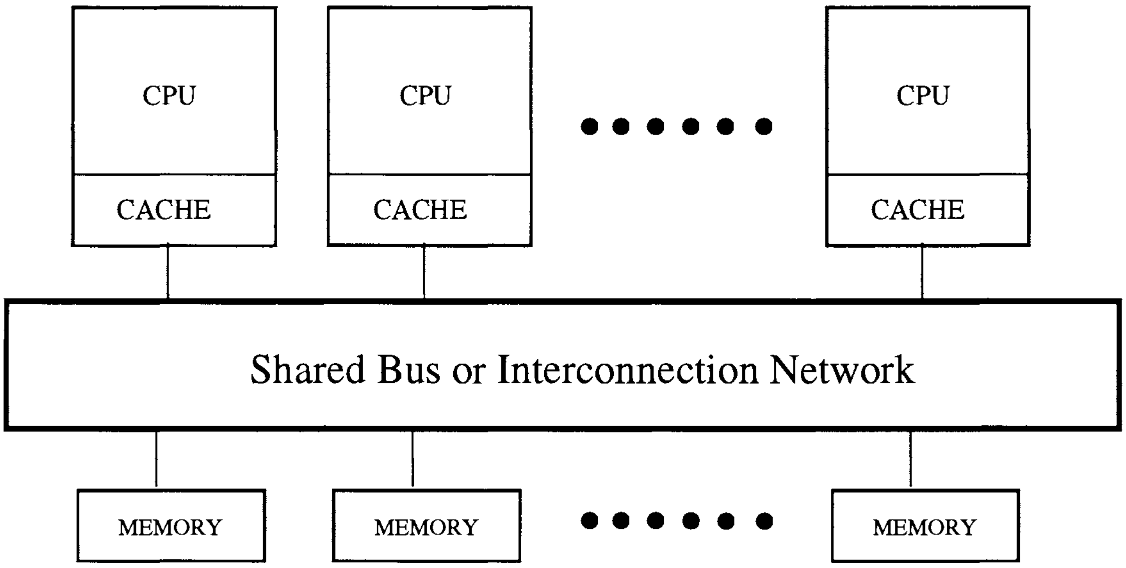


Figure 2.2 General block diagram of shared memory processor

k-bit counter is present in the cache controller that is incremented on every checkpoint and its current value is written to checkpoint identifier on every write access to the block. A cache line is considered as unwriteable if its checkpoint identifier is less then the counter.

To reduce the memory bandwidth while computation is taking place the replaced unwriteable line are put on to the "recovery stack," however, they have to be written back to the main memory before the subsequent checkpoint.

When an error is detected, all cache blocks except the unwriteable blocks are invalidated. The entries in the recovery stack are written back to the cache. The processor's registers are reloaded and the computation is restarted. The scheme is pretty much programmer transparent, except that the programmer has to properly synchronize the shared variables. The computation might be wrong if an error occurs while executing software with poor synchronization.

2.3.2 Shared Memory CARER

In [23], Ahmed et. al. did some modification to the CARER technique to make it workable for shared memory systems. They assume a shared memory bus as a connection network between the processors and the shared memory. They introduce three control lines to the base bus architecture.

1. Shared Line

2. Establish Rollback Point Line

3. Rollback Line

A checkpoint primarily is initiated whenever there is a need for replacing a dirty line on any processor. A checkpoint is also initiated when a processor accesses a modified variable present in the cache of some other processor. If a processor submits a read request for a modified line present in some other processor then only the supplying processor has to be checkpointed. Then, when a write request for a modified line is submitted, to avoid checkpoint propagation, both the serving as well as the requesting processors has to be checkpointed.

The establishment of checkpoint is similar to [15], i.e. all the dirty lines are validated and marked as unchangeable. Once an error is detected in a particular processor, then only that processor is rolled back, i.e., all the dirty lines in its cache are invalidated and its backup register file is copied to the active register file.

2.4 Cache checkpointing system in practice

Sequoia [27] is one of many practical fault tolerant systems. It consists of fault tolerant shared memory multiprocessor system and a fault tolerant operating system. Each of its processing nodes consists of duplex processors that are clock synchronized. Each processor has its own cache that is connected to the shared memory by a common bus. Many fault-tolerating techniques are used in sequoia to make it fault tolerant. Some of them are as follows:

- SEC/DEC code for covering data storage and transfer paths.

- Comparison of duplicated operations: multiprocessors, address generation logic, and cache management functions.

- Protocol monitors: for inter-element communication.
- Other software fault tolerant techniques.

Sequoia also employs cache checkpointing, i.e. it uses main memory as a safe storage for the checkpoint data. However, cache protocols implemented in sequoia are different then cache protocols for the share memory processors discussed in last section. Like other shared memory processors, the inter-processor communication takes place by accessing the shared variable, however these variables cannot be accessed directly from other's processor's cache. A processor can request the use of these shared variables. If no other processor is accessing these variables then these variables are fetched to the cache of that processor. After the processor completes the access to the shared variable, it releases it (writes it back to the memory).

However, data from the cache cannot be written to the memory without validating it, therefore a checkpoint has to be established. In Sequoia, a checkpoint is established whenever a dirty line or a shared variable is to be written to the main memory. The cache checkpointing scheme employed in Sequoia is more robust than other techniques discussed above. This is because it uses two memories to save its checkpoint (Figure 2.3). Thus the state can be recovered even if error occurs while saving the checkpoint.

While establishing a checkpoint, the duplex processor flushes its dirty lines and they are compared and saved in a backup memory. After that these lines are again flushed to the main memory. If the lines that are being flushed to the main memory don't compare to the lines present in the backup memory, then the entire procedure is repeated.

If the dirty lines from the two caches of the duplex processor don't match, then rollback process is initiated. To rollback the duplex processor, the dirty lines in the caches are invalidated and register file is loaded with the checkpointed register file data (which is present in the memory). After that the processor start processing again. Thought Sequoia provides practical implementation of cache checkpointing, the system does not provide for minimizing the overhead in checkpointing.

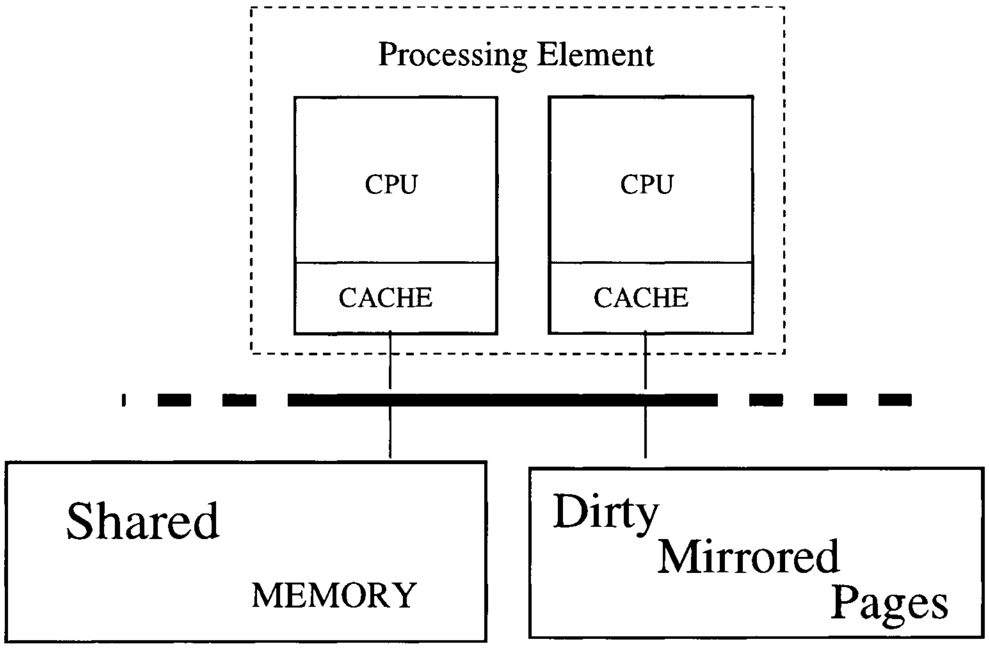


Figure 2.3 Processing element of Sequoia and it's partially duplex memory

CHAPTER 3. Towards Balanced Cache Checkpointing

The goal of checkpointing is to return the state of the system to the last correct execution in case of occurrence of a fault. Thus, the amount of computation wasted is minimized. Checkpointing involves validating and saving the execution state of the system at well-defined intervals. A program reloads the saved state when an error is detected. While developing a checkpointing scheme the main assumption is that the soft error rate is much higher than the permanent error rate [11]. If there were permanent faults in the system, then reexecution would not help the program to reach a fault free state.

3.1 Basic Considerations and Definitions

In order to facilitate further discussion on cache checkpointing we first describe the techniques that are common in all the cache checkpointing schemes.

Execution State

In cache checkpointing schemes, the *register file* as well as the entire memory hierarchy is the part of the execution state. The register file constitutes the *CPU state variables*, and the modified memory locations make *memory state variables.* To conserve memory bandwidth some checkpointing schemes propose to save only the modified portions of the execution state. Such schemes are called *incremental checkpointing* schemes.

Stable storage

One of the fundamental assumptions in any checkpointing scheme is that the storage structure (i.e., main memory or both cache and main memory or hard disk) used to store the checkpoint is either protected by its own error detection and recovery mechanism or it is inherently much more reliable than the system that is to be covered by the checkpointing scheme.

In the cache checkpointing schemes the memory state variables are stored in the main memory, and the CPU state variables (*the operating register file*) are stored in a separate *backup register file*, which is present along with operating register file on the CPU [15, 16, 22]. These storage devices are protected by error correcting codes, and/or memory scrubbing techniques, so the probability of their failure is very small [3, 4, 5]. Other cache checkpointing schemes like Sequoia doesn't use a backup register file, and the main memory is used to store the contents of the register file.

Cache-write policy

While saving the execution state at the time of checkpointing, it is preferred to save only the modified information. Incremental checkpointing reduces the time required for establishing a checkpoint. To adapt to the incremental checkpointing, the memory area must be classified into modified memory and non-modified memory areas. In case of cache checkpointing, the memory data in the cache is classified as dirty (modified) or clean (non-modified). A write policy must be given (assumed) to govern the writing to cache.

The cache-write policy that is used in most of the cache checkpointing schemes is write-back cache [9] with *least recently used* (LRU)[14] replacement policy (if the cache is set-associative). The line that is selected by LRU policy for replacement is the least recently used among a set of cache lines. Write-through policy requires checkpoint to be established on every store operation. However, for write-back caches a checkpoint is established when a dirty line is replaced back to the main memory.

Causes for checkpointing

After a checkpoint is established the cache and the main memory are identical, and the execution continues. A checkpoint is initiated under the following three circumstances.

1. When there is a need for replacement of a dirty cache line

2. When an external interrupt occurs

3. When an I/O instruction is executed

When an external interrupt occurs the normal flow of instruction execution is altered. A checkpoint must be established as soon as the interrupt is recognized, otherwise, if an error occurs during the interrupt service routine the computation is rolled back to a point in the main program before the interrupt occurred. The program does not re-enter the interrupt service routine since the external event that caused the interrupt might not re-occur. Another way of looking at this is to say that the computation gets rolled back, but the external world cannot be rolled back [15].

To establish a checkpoint the following step must be done.

- Verify the execution state (i.e. original register file and dirty cache lines) of the system

- Save the execution state to the safe storage.

Rollback and Recovery

When an error is detected, then all of the cache-lines are marked as invalid, and the backup register file is copied to the operational register file [15, 16, 22]. In this way the processor state is reset to the previously known fault free state. The processor then continues to execute from that point onwards.

3.2 Cache-aided rollback error recovery (CARER)

In [15], Douglas et. al. showed that the processor stall time at the point of checkpoint can be decreased if instead of copying all dirty lines to main memory the dirty cache lines are marked unchangeable. These lines are written back to the main memory as the execution continues. Writing an unchangeable line back to main memory does not affect the memory state, because the data in the line marked unchangeable is part of the verified state. They assumed that the cache memory is protected against errors just as the main memory and the register files.

3.2.1 Tasks performed

The processor system based on CARER technique performs the following main tasks.

1. Normal operation.

2. Checkpointing .

3. Rollback/Recovery .

3.2.1.1 Normal Operation

During the normal operation (i.e. when the processor is not performing a checkpoint or rollback) the processor operates in normal state, and executes the program. All memory references are served by cache, as in a conventional processor. However, for every write request to an unchangeable line, the line is first written back to the main memory (after which it is clean), and then it is written into. The "normal operation" continuous until there is a need to establish a checkpoint.

A checkpoint is established if an I/O operation is performed or an interrupt is recognized. Checkpoint is also established whenever the cache memory needs to write a dirty line back to the main memory. Checkpointing is necessary at that point because writing of a dirty line will change (or corrupt if faulty) the system state saved in memory.

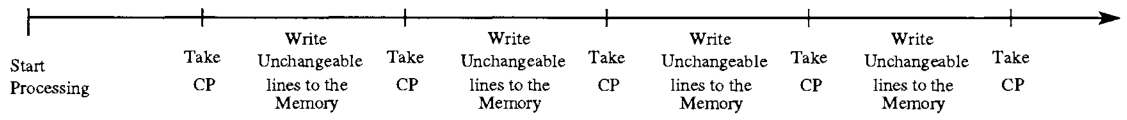

Start Processing
Take CP
Write Unchangeable lines to the Memory
Take CP
Write Unchangeable lines to the Memory
Take CP
Write Unchangeable lines to the Memory
Take CP
Write Unchangeable lines to the Memory
Take CP

Figure 3.1 Timeline for the execution of a program in CARER based processor.

3.2.1.2 Checkpointing operation

When the execution reaches a checkpoint the control unit generates a checkpoint interrupt to the processor to halt the program execution. The content of backup registers and the dirty cache lines are verified. The CPU state (the original register file) is copied to backup register file and all dirty lines in the cache are marked as unchangeable. Execution then resumes but from that point on the entries in the cache that are marked unchangeable is written back to main memory as the execution continues.

When there is a need to establish another checkpoint the entire cache is scanned for lines marked as unchangeable that are still present in the cache. These lines are written back to main memory. The processor then follows the normal checkpointing process.

The time taken to complete a checkpoint is a function of the number of unchangeable lines that are still present in the cache at the time of next checkpoint. This number cannot be predicted before the execution of the program with high accuracy. Therefore, the time taken to complete a checkpoint cannot be determined with high certainty before the execution of the task. Thus the predictability of the execution time of the task running on a checkpointing processor is low.

3.2.1.3 Rollback/Recovery Operation

Rollback/Recovery operation is initiated, whenever an error is detected during the process of verification of execution state of the processor (i.e. the original register file and the dirty cache lines). All of the dirty lines in the cache are invalidated, the contents of backup register file are copied to original register file, and temporary buffers in the processor are flushed [15]. The processor thus rolls-backs the program execution to the previous correct state. The program execution restarts from this point, and the system returns to the normal operation.

3.2.2 Behavior of Cache

The cache unit of a processor based upon the CARER technique uses a write back cache. The replacement policy for the cache is LRU policy [10]. The load/store operation is similar to that of a conventional cache. When the processor executes a load/store operation it sends data request to memory system. If there is a hit in the cache the data request is served by the cache.

A hit on read operation proceeds uninterrupted. If there is a hit in the data-cache for a write request, and the requested data is in a clean or dirty lines, then the memory system proceeds as in the conventional system. However, if the data is to be written to a unchangeable line, then the line has to be written back to main memory, and then the data is written to the line.

If there is a cache-miss for the data request by the processor, then a cache line is replaced. The line to be replaced is selected using the LRU algorithm. Based on the status of line (that is being replaced), different operations are performed.

- If the line to be replaced is clean/invalid, then it can be simply replaced.
- If the line to be replaced is unchangeable, then it is first copied to the main memory, and then replaced.
- If the line to be replaced is dirty, then a checkpoint has to be established.

Let H_{time} be the access time for the cache unit, M_R be the miss rate for the cache, and M_P be the time penalty for every cache-miss, then the average cache access time is expressed as:

$$t_{avg} = H_{time} + M_R * M_P$$

3.3 CBCA based processor

Figure 3.2 shows the block diagram of a processing system based on Checkpointing Buddy Cache Architecture (CBCA). The data requests from the processor are served by the data-cache, which operates in parallel with a small fully associative cache called as Checkpointing Buddy Cache (CBC). Both caches together serve the memory requests, but the data request is served by one of the caches. The CBC works in parallel with the data-cache. Therefore, the access-time for the entire cache system does increase (this hypothesis is evaluated in Section 5.3).

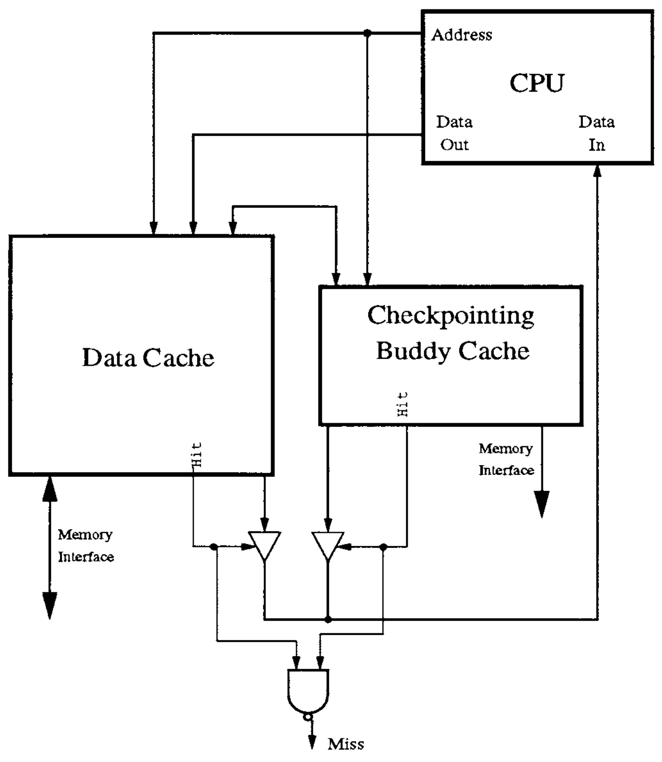


Figure 3.2 Logical connection between the data-cache and the checkpointing buddy cache.

3.3.1 Operation of CBCA-based processor

The operation of a processor using CBCA is similar to that of CARER-based scheme except for an extra operation (*CBC transfer*) performed by CBCA based microprocessor. The following different operations are performed by CBCA based microprocessor.

1. Normal operation

2. CBC transfer

3. Checkpointing

4. Rollback/Recovery

3.3.1.1 Normal operation

Normal operation proceeds as in Section 3.2.1.1, except for the cache access. Whenever the memory operation is executed, the memory request is simultaneously sent to the data-cache as well as to the CBC. The processor continues the normal operation until an I/O operation is to be performed, or an interrupt is recognized, or a replacement of a dirty line is needed.

3.3.1.2 CBC transfer

During the execution of program, if there is a need to replace a dirty line from the data-cache, then instead of writing it back to main memory, and establishing a checkpoint the line is written to the CBC. As the CBC is fully associative cache, therefore, the dirty line from data-cache can be written into any available line in the CBC. The memory system then brings data from the main memory, and writes it in the data-cache in the replaced dirty line. The processor then continues with normal operation. When the CBC is completely filled (by the dirty lines), and there is a need to replace another dirty line from the data-cache, a checkpoint has to be established.

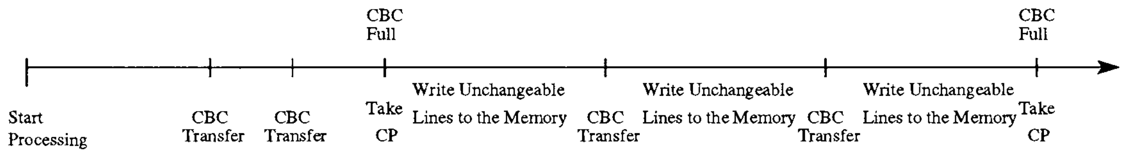

CBC Full
CBC Full
Start Processing
CBC Transfer
CBC Transfer
Take CP
Write Unchangeable Lines to the Memory
CBC Transfer
Write Unchangeable Lines to the Memory
CBC Transfer
Write Unchangeable Lines to the Memory
Take CP

Figure 3.3 Timeline for the execution of a program in CBCA based processor.

3.3.1.3 Checkpointing operation

The process of checkpointing for CBCA based processor is similar to that discussed in Section 3.2.1.2, except the unchangeable lines (if any) in both data-cache as well as the CBC (there might be some unchangeable lines in the CBC at in case of *interrupt checkpointing*) are copied to the main memory. Next, the CPU state and the contents of all dirty line of both data-cache and CBC are verified. After the verification the CPU state (the original register file) is copied to backup register file, and all dirty lines in the data-cache as well as in CBC are marked as unchangeable. These lines are written back to the main memory one at a time as when they are replaced in the normal operation. However, all unchangeable lines have to be written back before the next checkpoint is taken.

After the process of checkpointing is complete the memory system will have a saved correct processor state. The processor continues with normal operation. A decrease in the number of checkpoints increases the time between two successive checkpoints, causing greater number of unchangeable lines to get flushed to main memory before the next checkpoint. Therefore, the time that the processor stalls at the checkpoint deceases. The number of the unchangeable lines flushed at the checkpoint cannot be determined before execution. Therefore, the execution time still cannot be exactly predicted. However, our scheme decreases the average number of unchangeable cache lines present at the time of checkpoint, thus decreasing the average time to establish the checkpoint. Also, there is a reduction in the number of checkpoints. Therefore, overall our scheme reduces the execution times as compared to other cache checkpointing schemes.

3.3.1.4 Rollback/Recovery Operation

After copying the unchangeable lines (at the time of checkpointing) from the data-cache and the CBC a validation process is initiated. If an error is found during the validation process, then Rollback/Recovery operation is initiated. During the rollback/recovery operation all of the dirty lines in the data-cache as well as the dirty lines of the CBC are invalidated, and the contents of *backup register file* are copied to original register file and temporary buffers in the processor are flushed. The processor thus rolls-back the program execution to the previous correct state. The program execution restarts from this point, and the system returns to the normal operation.

3.3.2 Behaviour of the cache unit

The cache unit of a processor based upon the CBCA technique is a write back cache. If there is a cache-miss for the data request by the processor, then a cache line from the data-cache is selected according to LRU policy (if the data-cache is multi-way set associative). Based on the status of the line (that is being replaced), different operations are performed.

- If the line to be replaced is clean/invalid, then it can be simply replaced.
- If the line to be replaced is unchangeable, then it is first copied to the main memory, and then replaced.
- If the line to be replaced is dirty, then it is first copied to the CBC and then replaced.

The dirty line from the data-cache replaces a line in the CBC. At the checkpoint all the dirty lines in CBC are marked unchangeable, and copied into main memory as need arises. A CBC line needs to be written back into the main memory only to make room for replacement of a dirty line. The line to be replaced is selected by *least recently used unchangeable* (LRUU) algorithm. If the status of the line is clear, invalid or unchangeable then the operation performed it similar to that of data-cache. If the LRUU algorithm selects a dirty line to be replaced (i.e. the CBC is filled with dirty lines) then a checkpoint is established. Dirty lines in CBC are not replaced, as they are part of the current checkpoint state.

The hit time for the cache system ($H[CBCA]_{time}$) is same as the hit time for data-cache as long as the hit time for the CBC remains less than or equal to that of the data-cache. Let $H[data]_{Rate}$ and $H[CBC]_{Rate}$ be the hit rate for the data-cache and the CBC, respectively. Then

$$M[data]_{Rate} = 1 - H[data]_{Rate}$$
$$M[CBC]_{Rate} = 1 - H[CBC]_{Rate}$$

where $M[data]_{Rate}$ is miss rate for the data-cache and $M[CBC]_{Rate}$ is the miss rate for the CBC. Then the miss rate for the entire cache system is given by:

$$M[CBCA]_{Rate} = H[data]_{Rate} * M[CBC]_{Rate} + H[CBC]_{Rate} * M[data]_{Rate}$$

If the time penalty for a cache miss is determined by $M[CBCA]_P$, then average access time for CBCA is expressed as:

$$t_{avg} = H[CBCA]_{time} + M[CBCA]_{Rate} * M[CBCA]_P$$

CHAPTER 4. Cache Checkpointing for Superscalar Processor

In this chapter, the attributes of modern superscalar processors and a description of how the proposed cache checkpointing scheme can used to cover them against faults is described.

4.1 Superscalar Processor

Modern processors have multiple functional units to execute multiple instructions. These processors fetch and issue multiple instructions every clock cycle that don't have dependencies on each other. To further improve the throughput of the processor the instructions that have their dependencies resolved are allowed to execute before the instructions that don't have their dependencies resolved yet. This is called out-of-order execution [17]. However, executing the instructions out-of-order makes the exception handling imprecise. The exceptions are handled precisely by adding an extra stage to the pipeline. This stage is called the commit stage [18]. When an instruction is dispatched, a reservation is made for it in a circular queue called reorder buffer (ROB). The instructions at the top of the ROB whose execution has been completed are committed to the processor state. If the instruction is an ALU operation, then the result of instruction is committed to the *register file*, and a store instruction is committed to the memory hierarchy.

4.2 Checkpointing

Figure 4.1 show different pipeline stages the instruction has to pass through for complete execution. The process of checkpointing for superscalar processors is similar to the one described in Chapter 3, except that the read and write requests by the superscalar processor are sent to cache system in different pipeline stages. The load request is sent to the memory hierarchy in *Execute/Memory* stage, whereas the store request has to wait till the instruction is in *Commit* stage.

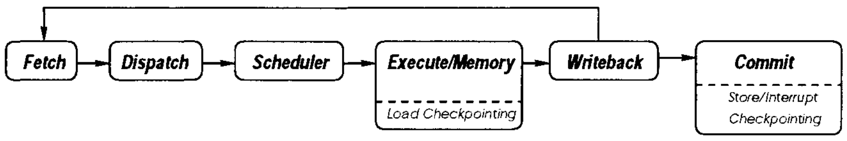


Figure 4.1 Pipelined Superscalar Processor

4.2.1 Exception/Interrupt

Whenever an exception occurs or an interrupt is recognized the processor pipeline stops further instruction fetching and dispatching. The ROB entry of the instruction that caused the exception is marked. When the marked entry reaches top of the ROB the exception handling procedure is invoked, thus making the exception handling precise. In our scheme, checkpoint is established (Section 3.3) when such an instruction reaches the top of the ROB (Figure 4.1). After the checkpointing processes is completed, the processor invokes the Exception/Interrupt handling routines. The checkpointings triggered by exception/interrupt are collectively called *interrupt checkpointings.*

4.2.2 Store Operation

Access to memory hierarchy is initiated by load/store instructions. This access might initiate a checkpoint (i.e., there is no room in the CBC for more dirty lines). In superscalar processor the actual store request is carried out when the instruction is about to commit (Figure 4.1). If there is a need to establish a checkpoint, the cache proceeds with the process of checkpointing, and the CPU is not stopped. However, if there is a need to process another load or store request, then the processor is stopped. If the load/store instructions following the "checkpointing store" are far from the store instruction that caused checkpointing, the ROB quickly fills up, and there would be no space for new instructions, and future dispatching of the instruction is stopped until the completion of the checkpointing process. This would eventually stop further fetching, and thus stopping the entire processor.

4.2.3 Load Operation

The actual load operation in a load instruction is performed in the execute/memory stage of a superscalar pipeline (Figure 4.1). If a load instruction initiates the establishment of checkpoint, the ROB entry reserved by that load is not released until the completion of checkpointing process. The instructions following the "checkpointing load" (load instruction that triggered checkpointing process) might be dependent upon its result, and hence will be stalled. Eventually the entire ROB fills up, and the processor stalls until the competition of the checkpointing operation.

Both load and store checkpoints are initiated because of the replacement of dirty lines. Therefore, these checkpoints are called *replacement checkpoints.*

When there is a need to establish a checkpoint, for any of the above-mentioned reasons, the following operations are carried out by the processor.

1. Flush all the unchangeable lines in data-cache or CBC (if any), to the main memory.
2. Validate entries in the *original register file* and all the dirty cache lines.
3. Mark all the dirty lines to unchangeable.

4.3 Recovering form a fault situation

Whenever a fault is detected, during the verification stage of a checkpoint, the following steps are taken to return the processor to the fault free state.

1. All the unchangeable cache lines either in data-cache or CBC (here the cache and memory are assumed to be reliable), if any, are written back to the memory.
2. The entire ROB is flushed.
3. All the cache lines in data-cache and CBC are marked invalid.
4. The backup register file is copied to original register file.

At this point, the processing state is rolled back to the last fault free state. All memory locations, as seen by the processor, are the same as they were when the last checkpoint was established.

CHAPTER 5. Evaluation and Results

5.1 Simulation

Previous works on checkpointing cache architecture only use program traces of memory accesses [12, 13, 15]. Their results don't take into account the effect of checkpointing on the instruction execution. In our research, we have used simplescalar tool set 3.0 [19] that simulates out of order superscalar processor, using spec95 and spec2000 [20] binaries as the input program to the simplescalar "processor." Therefore, the results generated by our experiments are representative of the overall affect of checkpointing on the performance due to various parameters of the processor.

The original simplescalar simulator assumes a totally fault free environment, and doesn't simulate any fault detection or correction mechanism. Changes were made to the base simulator to incorporate checkpointing mechanisms in it. To achieve this, *cache.c*, *cache.h* and *sim-outorder.c* were modified, and the modified simulator is called *sim-check*.

Three types of processors, based on simplescalar tool set, are used for experiments. The first processor assumes a fault-free environment. Therefore, this simulator (named sim-fault-free) has no fault tolerant features built into it. The second processor uses CARER cache architecture, and perform checkpointing as described in the Section 3.2 to cover the faults. The sim-check simulator has been modified to develop *sim-CARER* . The procedure of checkpointing is changed so that the dirty cache lines would only be marked unchangeable, instead of flushing on every checkpoint. The third processor uses CBCA cache architecture (Section 3.3) to reduce the number of checkpoints. This simulator is developed based on the sim-CARER. Modifying the *cache* files so that the simulator copies the replaced the dirty lines to the CBC, instead of taking a checkpoint on every replacement. This simulator is called *sim-CBCA*.

Table 5.1 shows the simulator parameters that are common to all the architectures (i.e. sim-fault-free, sim-CARER, and sim-CBCA).

Table 5.1 Input parameters to the simplescalar simulator

Parameter	Value
Instruction Fetch Queue Size	4
Decode width	4
Issue width	4
Commit width	4
RUU size	16
LSQ size	8
D1-Cache Sets	128
D1-Cache line size	32 Bytes
D1-Cache Associatively	4-way
D1-Latency	1
I1-Cache Sets	512
I1-Cache line size	32
I1-Cache Associatively	1-way
I1-Latency	1
D2-Cache	None
I2-Cache	None

SPEC95 and SPEC2000 programs were used. PISA little-end binaries obtained from [19]. Simulations are run on a Linux workstation. Each benchmark on the simulator is run for 200 million instructions [24] with parameters shown in Table 5.1.

5.2 Results and Discussions

Checkpointing comparison

Figures 5.1, 5.2, and 5.3 show the variation in the number of checkpoints for various CBC sizes while executing various benchmarks. For each benchmark the number of checkpoints obtained has been normalized with the number of checkpoints obtained for CARER architecture. The figures show that the number of checkpoints required for programs executing on a processor based on *checkpointing buddy cache architecture* is less than number of

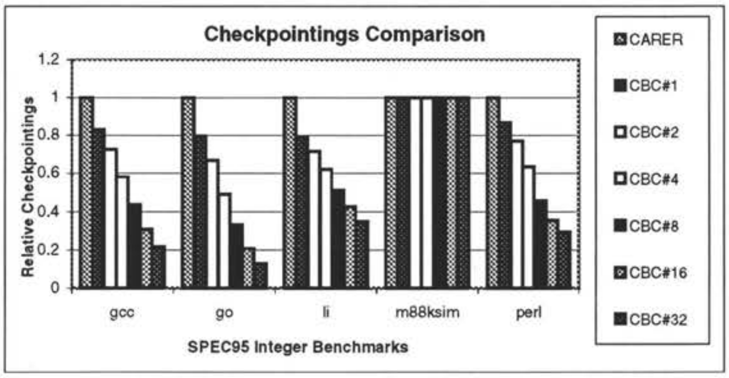


Figure 5.1 Comparison of number of Checkpoints between different cache architectures using spec95 Integer benchmarks

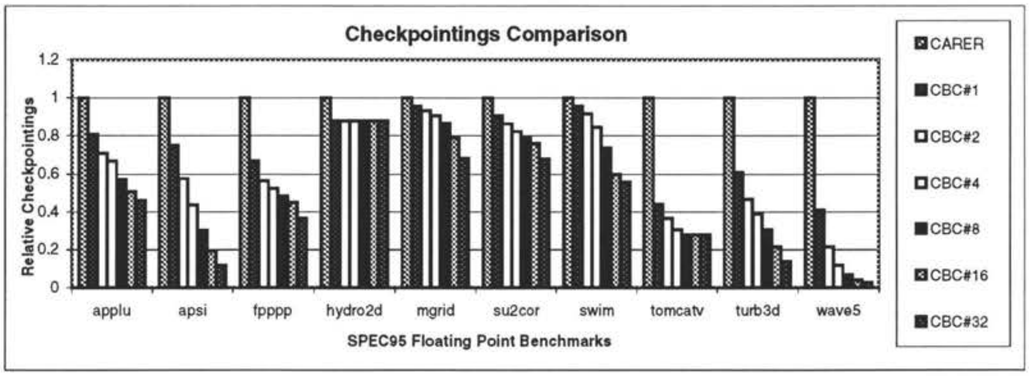


Figure 5.2 Comparison of number of Checkpoints between different cache architectures using spec95 floating point benchmarks

checkpoints required for the programs executing on a processor based on CARER. There is a clear decrease in number of checkpoints as the CBC size is increased from 0 (CARER) to 32. This decrease is linear in most of the benchmarks. For some floating-point benchmarks like fpppp, tomcatv, turb3d and wave5, the decrease is considerable, but saturates with the higher CBC size. The benchmark Hydro2d requires a smaller number of *interrupt checkpoints*, and much smaller number of *replacement checkpoints*. Therefore, there is no decrease in the number of checkpoints as the size of CBC is increased beyond one. The benchmark m88ksim has very large number of interrupt checkpoints whereas its replacement checkpoints are very small. Therefore, there is almost no decrease in its number of checkpoints, with the increase in the CBC size.

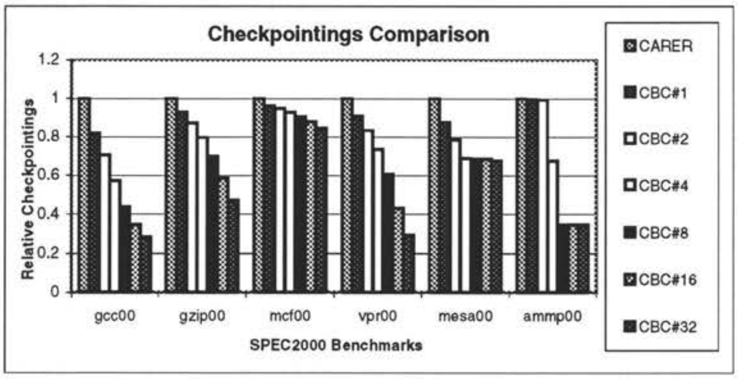


Figure 5.3 Comparison of number of Checkpoints between different cache architectures using spec2000 benchmarks

Research herein shows that the number of checkpoints needed for programs running on the CBCA based processor is less than the number of checkpoints needed for programs running on CARER. In Section 3.2.1.2, it was discussed that the predictability of execution time decreases with the increase in the number of checkpoints because of the unpredictable stall time of the checkpoint. The average stall time in case of CBCA based processor is less than the conventional cache checkpointing processor. Therefore, the execution time for the programs running on CBCA based processor is more predictable than on CARER based processor. There is an overall trend of decrease in the number of checkpoints with the increase in the CBC size. However, actual decrease depends upon the number of replacement checkpoints in non-CBC architecture. This finding follows Amdahl's law, i.e., if the number of replacement checkpoints is more than the number of *interrupt checkpoints*, the overall decrease in the number of checkpoints is significant.

Miss rate comparison

Figure 5.4 Comparison of cache miss rate between different cache architectures using spec95 Integer benchmarks

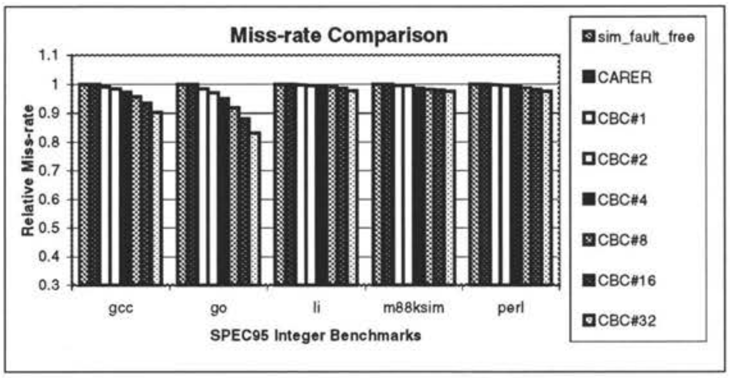
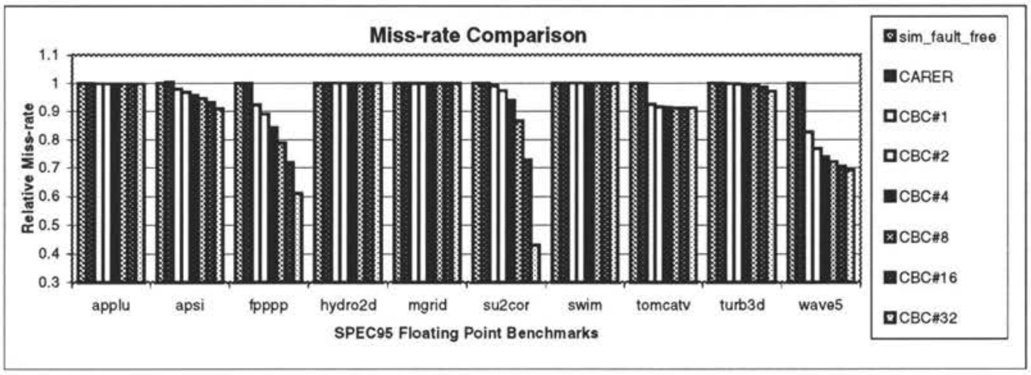


Figure 5.5 Comparison of cache miss rate between different cache architectures using spec95 floating point benchmarks



Figures 5.4, 5.5, and 5.6 show the variation in the cache miss rate for various CBC sizes and sim-fault-free while executing various benchmarks. For each benchmark the cache miss rate for different architectures has been normalized with the miss rate obtained for sim-fault-free.

The cache miss rate for sim-CBCA is less than the cache miss rate for sim-fault-free and sim-CARER. The miss rate decreases with the increase in the size the CBC size. The trend is different for different applications. For example, in the benchmarks like li, m88ksim, perl, applu, hydro2d, mgrid, swim and turb3d, the decrease is not very significant. However, the decrease in the miss rate is significant for other benchmarks. The dirty cache line that is to

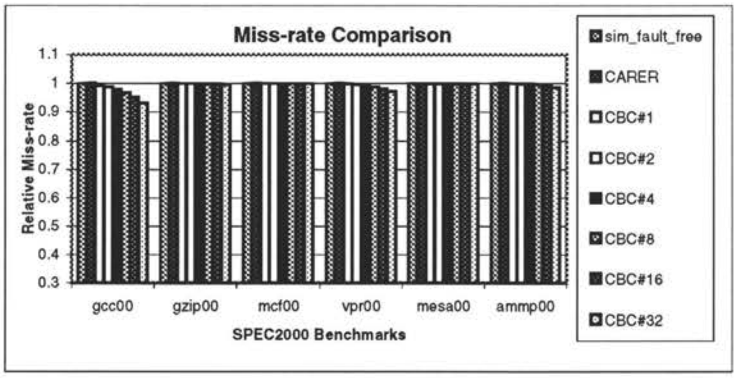


Figure 5.6 Comparison of cache miss rate between different cache architectures using spec2000 benchmarks

be replaced from the data-cache is placed in the CBC and when the processor accesses that line again, its request is serviced by the CBC and no cache miss is generated. However, the decrease in miss rate depends on the actual use of data in the replaced lines. The decrease in the miss rate for the various benchmarks also helps in overall improvement in the performance of the processors.

Execution cycle comparison

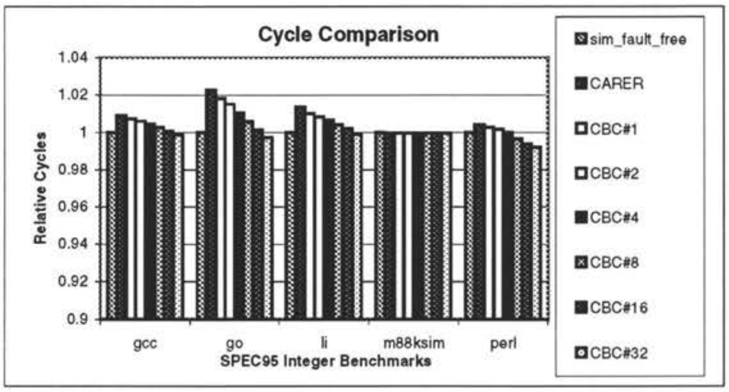


Figure 5.7 Comparison of number of Checkpoints between different cache architectures using spec95 Integer benchmarks

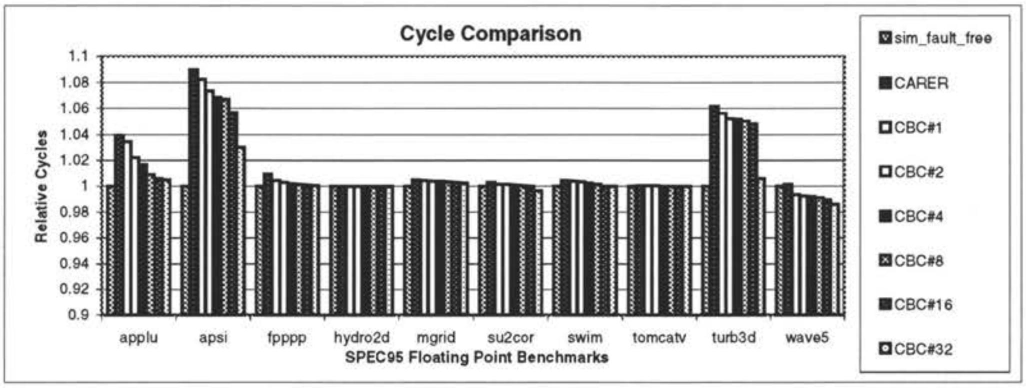


Figure 5.8 Comparison of number of Checkpoints between different cache architectures using spec95 floating point benchmarks

Redundancy is used to improve the reliability of a system. The redundancy can be in the form of extra hardware or extra execution time. In either case the performance of the system decreases as compared to the base system. Subsequently, checkpointing being a time redundancy technique is expected to affect the performance of the system. The following paragraphs discuss the impact the CBCA has on the performance of the system as compared to the classical cache checkpointing architecture.

Figures 5.7, 5.8, and 5.9 illustrate the impact of cache checkpointing on the performance

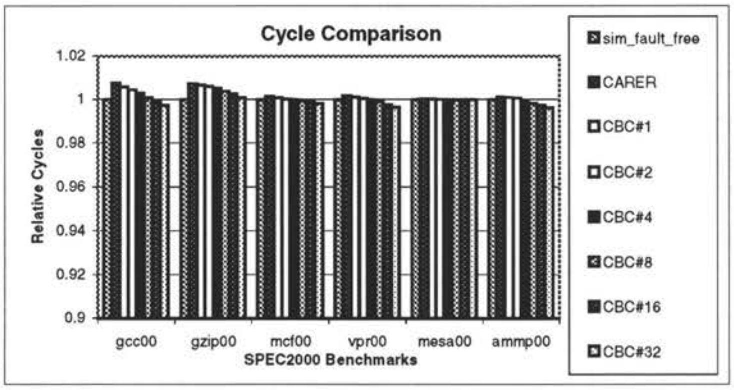


Figure 5.9 Comparison of number of Checkpoints between different cache architectures using spec2000 benchmarks

of the system. It can be seen the CARER degrades the performance for most benchmarks. However, the impact of CBC can be seen to be less degrading as compared to the CARER scheme. Further, this degradation in the system performance is reduced with the increase in the CBC size. For each benchmark the execution cycles obtained for different architectures has been normalized with the execution cycles obtained sim-fault-free.

With the exception of m88ksim and hydro2d that do not have significant number of replacement checkpoints, other benchmarks show improvement in the number of execution cycles over CARER scheme, depending on the number of decrease in the checkpointing and miss rate. In the previous paragraphs, it has been shown that the number of checkpoints as well as miss rate improves as the CBC size increases. The improvement of the execution cycles depends on, according to Amdahl's law, the actual fraction of time these two factors represent. The graphs show performance improvement for most the benchmarks. They show that the number of execution cycles for checkpointing buddy cache architecture become *similar* to non-fault tolerant processor with increase in the CBC size.

5.3 CBCA access time

In section 3.3, we discussed that data-cache is working in parallel with a small fully associative cache. Therefore, the overall access time for the entire cache system would not increase. To verify this hypothesis, Cacti 2.0 was modified to include CBC working in parallel with data-cache. Modified simulator takes, data-cache as well as CBC parameters as input. The program calculates overall access-time for the cache system, and individual access-times for data-cache and CBC.

Table 5.2 Input parameters to modified Cacti simulator

Parameter	Value
Technology	0.18 micron
Number of read/write ports	2
Data-cache size	16KB
Data-cache line size	32B
Data-cache Associativity	4
CBC line size	32B
CBC size	32B, 64B, 128B, 256B, 512B 1024B
CBC Associativity	DM, 2, 4, 8 16, 32

Table 5.2 shows different parameters used for calculating access-time for different checkpointing buddy cache architectures.

Figure 5.10 shows the comparison between the access-times of the data-cache and CBC of different sizes. Access time for CBC increases as the size of CBC is increased. For the CBC size of up to 16 lines the access-time of the CBC is less than data-cache. Therefore, the access-time of overall cache unit (CBCA) is equal to access time of data-cache. However, the access-time for the CBC of size 32 lines is 43% greater than the access time for the data-cache, making the access-time for CBCA equal to access-time for CBC-32. Therefore, increase in the CBC size should be limited to 16 lines.

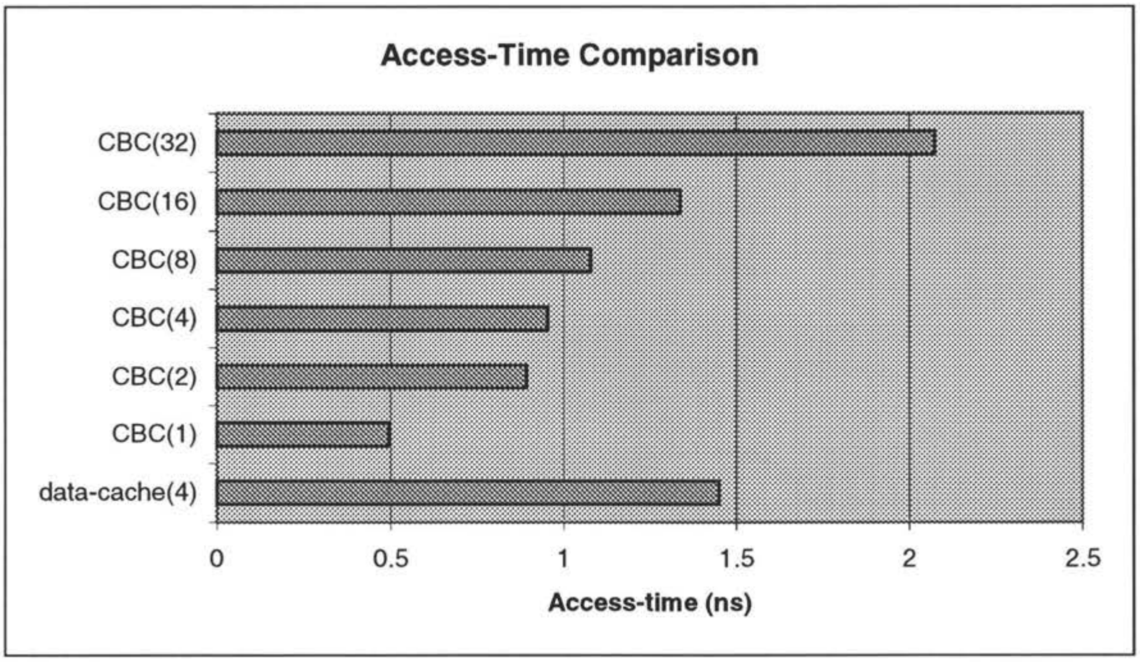


Figure 5.10 Comparison among access time of data-cache and CBC of different sizes

CHAPTER 6. Conclusion

In this thesis a new cache checkpointing scheme was developed. It was observed that writing modified cache lines, that are needed to be replaced to the main memory, to a fully associative cache buffer (CBC) would decrease the number of checkpoints by 10% to 80% as compared to existing cache checkpointing schemes. Checkpoint is only established once the CBC is filled with dirty lines. The processor accesses the data-cache as well as CBC concurrently. Therefore, no miss is generated whenever the processor requests for the data from a line that is present either in the data-cache or in the CBC. The mechanism decreases the cache miss rate for various benchmarks executing on the CBCA-based processor up to 26% as compared to the base architecture. Whenever an error is detected, the processor can rollback to fault free state by copying the contents of *backup register file* to the original register file, and invalidating all the dirty lines in data-cache as well as in CBC. It was also found that the performance degradation for the CBCA-based processor is less than the performance degradation for classical checkpointing architecture. The performance of the overall system improved with the increase in the CBC size.

It was also observed that the number of lines in the CBC couldn't be arbitrarily increased. It was found that the access time for the fully-associative CBC of size 32 lines is larger than the access time of data-cache by 43%, which might lead to increase in the cycle time of the system. The access time for CBC of size up to 16 is less than the access time for data-cache. Therefore, the overall cycle time of the processing system is not affected. When fault tolerant schemes are applied to a system, the performance of the system decreases. Checkpointing, being a time-redundant scheme, greatly affects the performance of the system. It was found that the CBCA based processor provides better performance as compared to classical cache checkpointing scheme. The performance is similar to the base architecture for most of the benchmarks. The CBCA based processor takes less number of cycles to execute some benchmarks like perl, wave5, mcf00 and mesa00 than the base processor.

Checkpointing buddy cache architecture is a step toward balanced checkpointing because it eliminates unnecessary checkpoints while maintaining the ones that are necessary to minimize the computational loss in case of faults. The decrease in the number of checkpoints causes the inter-checkpoints interval to increase, thus causing more unchangeable lines to be written back to main memory before the establishment of the next checkpoint. This phenomenon decreases the average time to establish a checkpoint. Because of the unpredictable nature of the checkpoint the conventional cache checkpointing architectures are not suitable for real time applications, as they require the establishment of a lot of checkpoints. However, checkpointing buddy cache architecture based processors with their balanced checkpoints, are more suitable for *reliable*, *real time* applications as compared to conventional cache checkpointing architectures.

BIBLIOGRAPHY

[1] C. B. Robert, "Soft Errors in Advanced Semiconductor Devices—Part I: The Three Radiation Sources," *IEEE trans. on Device and Materials Reliability*, Vol.1, pp 17-22,March 2001.

[2] S. Broker, "Design Challenges of Technology Scaling," *IEEE Micro 19*, pp 23-29, March 1999.

[3] P. Hazucha P., C. M. Svensson, and S. A. Wender, "Cosmic-Ray Soft Error Rate Characterization of a Standard 0.6-mu/m CMOS Process," *IEEE Journal of Solid State Circuits*, Vol. 35, No. 10, pp 1422-1429, 2000.

[4] C. L. Chen and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Application: A State of Art Review," In *Reliable Computer Systems - Design and Evaluation, Digital Press*, 2nd edition, pp. 771-786, 1992.

[5] A. M. Saleh, J. J. Serrano and J. H. Patel, "Reliability of Scrubbing Recovery-Techniques for Memory Systems," *IEEE Transactions on Reliability*, Vol. 39, No. 1, pp. 114-122, April 1990.

[6] J.A. Stankovic and K. Ramamrithan, "What is Predictability for Real Time Systems?" *Real-Time Systems*, No. 2, pp. 247-254, 1990.

[7] Tzi-cker Chiueh, "Stonehenge: a Fault-Tolerant Real-Time Network-Attached Storage Device," *Hot Interconnects 9, 2001*, pp. 57-61, 2001.

[8] P. Richardson, L. Sieh, and A. M.Elkateeb, "Fault-Tolerant Adaptive Scheduling for Embedded Real-Time Systems," *IEEE Micro*, Vol. 21, No. 5, pp. 41-51, Sep-Oct. 2001.

[9] Y. Wu, "Evaluation of Write-Back Caches for Multiple Block-Sizes", In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994 MASCOTS*, pp 57-61, 1994.

[10] Y. C. Chen and A.V. Veidenbaum, "An Effective Write Policy for Software Coherence Schemes," In *Proc. of Supercomputing '92*, pp. 661-672, 1992.

[11] D. P. Siewiorek, and S. S. Robert, "The Theory and Practive of Reliable System Design,". *Digital Press*, 1982.

[12] P. L'ecuyer and J. Malenfant, "Computing Optimal Checkpointing Strategies for Rollback and Recovery Systems," *IEEE Transaction on Computers*, pp 491-496, April. 1988.

[13] Y. Tamir, M. Tremblay, and David A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI System," *The International symposium on Fault-Tolerant Computing*, pp. 234-239, 1988.

[14] W. S. Wong and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Transactions on Computers*, Vol. 37, No. 6, pp. 637-645, June 1988.

[15] B. H. Douglas and P. N. Marinos, "A General Purpose Cache-Aided Rollback Error and Recovery Systems," *The 17th International symposium on Fault-Tolerant Computing*, pp. 170-175, 1987.

[16] Ho-Shyan Lin, "High-Performance Comparison-Based Cache-Aided Rollback Error Recovery Computing System," *MSc Dissertation*, University of Washington, 1989.

[17] D. W. Anderson, J. Sparacio, and T. M. Tomasulo, "The IBM 360 Model 91:Processor Philosophy and Instruction Handling," *IBM J. Research and Development*, pp. 8-24.

[18] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. On Computers*, Vol. 37, No. 5, pp. 562-573, May 1988.

[19] www.simplescalar.com

[20] www.spec.org

[21] P. A. Lee, N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," *IEEE transaction on Computers*, Vol. C-29, No. 6, June 1980.

[22] R. E. Ahmed, "Checkpointing and Error Recovery in a Uniprocessor System with On-chip Cache," *Canadian Conference on Electrical and Computer Engineering, 2001* , Vol. 1, pp. 417 - 422, 13-16 May 2001.

[23] R.E. Ahmed, R.C. Frazier and P.N. Marinos, "Cache-Aided Rollback Error Recovery (CARER) Algorithm for Shared-Memory Multiprocessor Systems," *The 20th International Symposium on Fault-Tolerant Computing* , pp. 82-88, 1990.

[24] S. Kim and A. K. Somani, "SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors," Proceedings of Pacific Rim International Symposium on Dependable Computing, pp. 27-34, 2001.

[25] J. B. Nickel, and A. K. Somani, "REESE: a Method of Soft Error Detection in Microprocessors," Proceedings of *The International Conference on Dependable Systems and Networks* , pp. 401-410, June 2001.

[26] K. L. Wu, W. K. Fuchs, and J. H. Patel, "Error Recovery in Shared Memory Multiprocessors using Private Caches," *IEEE Transactions on Parallel and Distributed Systems*, Vol 1, Issue 2, pp 231-240, April 1990.

[27] P.A. Bernstein, "Sequoia: a Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *Computer*, Vol. 21, Issue 2, pp. 37-45 Feb. 1988.

ACKNOWLEDGEMENTS

I would like to thank Dr. Arun K. Somani, my major professor, for his kind guidance, patience and constant support during this research. I would also like to thank Dr. Akhilesh Tyagi and Dr Govindarasu Manimaran who expressed their interest in my work and helped me by giving a number of suggestions and ideas. I would like to thank all the members of DCNL for providing me support and help throughout the course of my master program.

I would also like to mention that National University of Science and Technology, Pakistan, sponsored my graduate studies in USA. I am thankful to them for providing me with this opportunity.

I am grateful to my parents for their patience and *love*. Without their prayers this work would never have completed. I would also like to thank all my friends for their constant support.

Adeel Israr,

Ames, Iowa,

October 18, 2002