Abstract

Check-pointing is an effective and low cost technique to improve the reliability in the presence of hardware faults. Traditionally, we take the checkpoint every fixed interval and roll back to the previous checkpoint if there is a fault. The problem of determining how often to take the checkpoint is called the optimal checkpoint interval problem. In this project, we propose two adaptive check-pointing techniques: The first is the equation based; the second is based on the MTTF and variable x. Designing the simulation to test these two adaptive check-pointing techniques. They will then be compared with the traditional fixed check-pointing technique. Finally, the effectiveness of these two adaptive check-pointing techniques will be validated using the software simple-scalar tool set.

Contents

1. Introduction

1.1Aims & objectives

This project is to develop an adaptive check-pointing technique for software implemented fault tolerance. In this project, the effectiveness of software implemented fault tolerant is investigated using adaptive check-pointing technique. The adaptive check-pointing technique is compared with traditional fixed check-pointing techniques. A System-level simulation tool SimpleScalar is used to model the application and fault model, and validate the effectiveness of the proposed adaptive check-pointing techniques.The objectives are shown below:

1. Understand the fault-tolerant system and the mechanism of check-pointing
2. Learn and become familiar with the software Simple-scalar tool set
3. Design the method to simulate the check-pointing technique
4. Develop and test the fixed check-pointing technique
5. Develop and test the first adaptive check-pointing technique
6. Develop and test the second adaptive check-pointing technique
7. Compare these check-pointing techniques and try to improve the adaptive check-pointing technique

1.2 Organisation of Dissertation

This report contains seven main sections. In the first section, we introduce the background information of fault tolerant, check-pointing technique and provide a summary of the important parts of this section. In the second section, we design the method to do the simulation, the system mode and error injection; fault detection and the use of the software are introduced in this section. In the third section, we introduce and test the fixed check-pointing technique and adaptive check-pointing technique. We will analyse the effectiveness of each check-pointing technique. In the fourth section, we compare the adaptive check-pointing techniques and do some improvements on these techniques. The fifth section is the conclusion; we will conclude the work we have finished. The sixth section is an evaluation of the project to determine if we finished the objectives and met the aim. The final section is to provide some future analysis of software fault tolerant and check-pointing technique and discuss some future work about the project.

2. **Background**

2.1 Reliability and Fault tolerant system

In recent years the complexity of the electric system has increased dramatically and the electric system is a thousand times faster than a few decades ago. However, the complexity and the long-time execution always cause faults in an electronic system. As a result, the reliability which is the probability that an electronic system provides proper service is an emerging designing challenge for the current and future electric system. The computer scientists and engineers have responded to the challenge of designing complex systems with a variety of tools and techniques to reduce the number of faults in the system they build [1]. The system that incorporates techniques to tolerate these faults and gives the system an acceptable level of service is known as a fault tolerant system. In the fault tolerant system, we still get faults. However, the system can still work properly. So the fault tolerance can make the system more dependable. There are two methods to increase the reliability of the system: the hardware based method which uses a hardware fault tolerance mechanism and software implemented fault tolerance technology (SWIFT).

2.2 Software implemented fault tolerance for reliability

2.2.1 SWIFT

The software implemented fault tolerance (SWIFT) is a software-based technology which is cheaper and lower in cost than hardware based method for increasing the reliability. In general, software implemented fault tolerance incorporates the software component such as a C program into the application system to detect the fault that causes the application system to crash and recover the application system from the fault.[1] The operation of a simple SWIFT is shown in Figure 2-1.

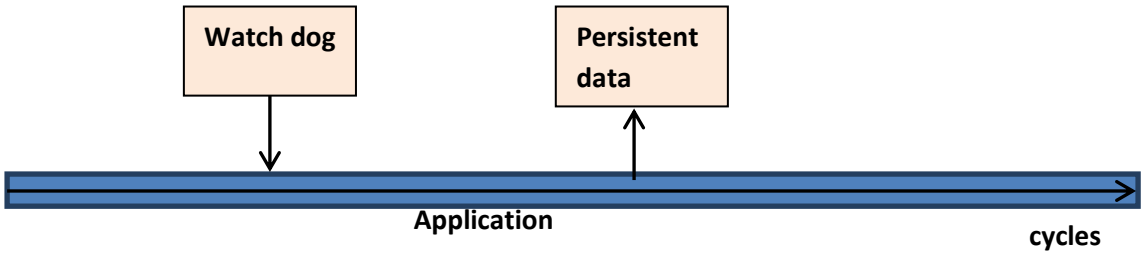


Figure2-1 SWIFT

In Figure 2-1, the watchdog is used to detect the fault of the system application and the persistent data is stored in the memory. When the watch-dog detects a fault, the system will recover the application and rerun the application. In general, there are two main methods to recover the system: roll back recovery and roll forward recovery. For roll back recovery, the system will roll back to an earlier state and restart the application from that state. The roll-back recovery protocol can be classified into checkpoint-based and log-based. For this project, we will use the checkpoint-based roll-back recovery protocols which are simpler to implement than log-based rollback recovery. The roll forward recovery technique will save the state when a fault is detected and correct the fault so the system does not need to roll back. However, the roll forward recovery technique is very complex; it needs online backup image and all transaction log files.

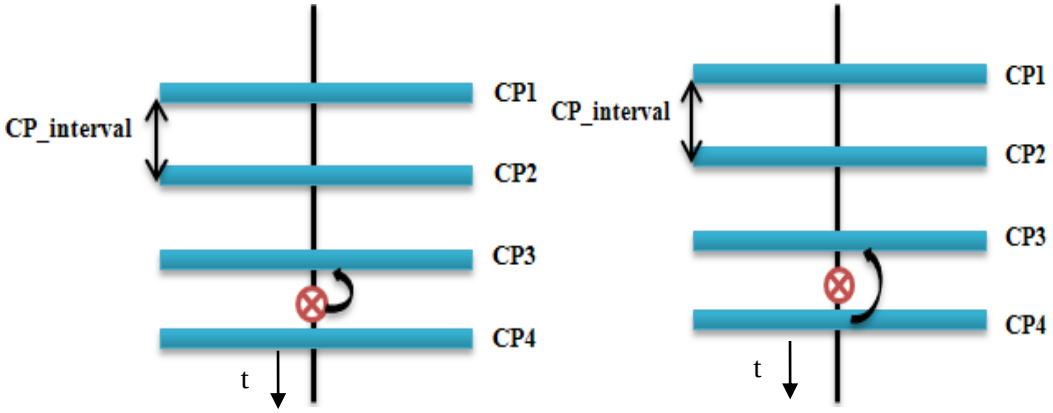
2.2.2 Summary

In this part, the operation of the fault tolerant system and two recovery methods are introduced. For this project, we will use the roll back recovery because roll forward recovery is very complex and it can be a separate project.

2.3 Check-pointing Technique

2.3.1 Check-pointing Mechanism

The check-pointing technique is an effective and low cost technique to improve the reliability in the presence of hardware faults. The checkpoint which is taken every fixed interval is just a snapshot of the entire state of the process at the moment it was taken. Upon a failure, the system will use the last checkpoint to restart the computation from an intermediate state, thereby reducing the amount of lost computation [12]. The basic mechanism of the check-pointing is shown in Figure 2-2.



a. persistent error checking b. flag based error checking

Figure2-2 mechanism of check-pointing

In Figure 2-2a, the system rolls back to the previous checkpoint immediately when there is an error. In Figure 2-2b, if there is an error detected, the system will go on execution and roll back until the next check-pointing time. These two different mechanisms are based on different error checking: persistent error checking and flag based error checking. The algorithms are shown below.

```
if there is an error {
  error_flag = true;
  perform rollback();
}else
  error_flag=false
......
```

```
if there is an error
    error_flag = true;
else
    error_flag = false;
......
......
```

Flag based error checking Persistent error checking

For persistent error checking, we check the system every cycle. If there is an error detected, we roll back the system to the previous state. The persistent error checking can reduce the amount of lost computation, however it will increase the use of memory space and

may make the memory overflow because we compile the function: perform rollback () every cycle and in general ,a system has tens of thousands of cycles.

For the flag based error checking, we will check the error_flag at the end of each checkpoint interval and decide whether to roll back. This method will increase the amount of lost computation rather than persistent error checking but the use of the memory space is small. For this project ,we will use the flag based error checking because when we use the error flag based error checking the lost computation is equal to the checkpoint interval, which makes the result more stable and easy to investigate.

2.3.2 Storage and Overhead

To make sure the system state saved at the checkpoint is correct, we save the checkpoint on stable storage whose reliability is very high. When we choose the stable storage medium, we are making the judgment that its reliability is sufficiently high for the application. Disks are the most commonly used, because they can hold the data even if there is no power (so long as there is no physical damage to the disk) and the amount of data can be stored cheaply. Sometimes, we also use standard RAM which is rendered non-volatile [1].

There are two important terms defined in check-point technique: checkpoint overhead which is the time used to take the checkpoint and the checkpoint latency which is the time used to save the checkpoint (The checkpoint latency always depends on the checkpoint size).For these two terms, the checkpoint overhead has a much greater impact on the process because the overhead is not done in parallel with the application [1]. The more checkpoints, the higher the cost of the check-pointing technique and the overhead of the checkpoint cannot be neglected. Therefore, the check-pointing technique is limited by the number of checkpoints. In a simple system, the checkpoint overhead and checkpoint latency are identical.

2.3.3 Type of check-pointing techniques

There are two broad classifications of checkpoint techniques. The first classification is to categorize them according to the software layer. The second classification is to categorize them according to synchronization of processes of a system during check-pointing which is used in a distributed system [4, 11]. For the software layer responsible, we classify check-pointing into three types: OS, middleware and application layer. When we take a checkpoint in the OS layer, the check mechanism will be transparent to middleware and application because the OS is privileged. The advantage of OS layer check-pointing is that middleware and application running on the OS does not need modification .This category checkpoint is called a transparent checkpoint. When we take a checkpoint in the application layer, we have to modify the application to integrate the check-pointing mechanism. However, it is easier to identify the meaningful application state in the application layer. This category of checkpoint is called explicit. Taking checkpoint at the middleware will combine the benefit of the OS layer and application layer. In middleware, we do not need to modify the applications and it is easier to identify the meaningful application state in middleware than the OS layer. This category of checkpoint is called implicit. If we use the explicit checkpoints, check-pointing should be triggered by the application event so we must modify the application to trigger the check-pointing. If we use the implicit and the transparent checkpoints, we should implement

some logic to trigger the checkpoint timing.

According to check-pointing synchronicity, we can classify the checkpoint technique in two categories. One category of checkpoint timing technique is to coordinate all processes of a system when taking checkpoints. The set of checkpoints which is taken after such coordination will be guaranteed to form a consistent global state of the system for the cost of system-wide coordination [4]. This category of checkpoint technology is called coordinated checkpoints. Another category of checkpoint technology is called independent checkpoint. In this category, the processes take the checkpoint without synchronizing with each other so we need to find a set of checkpoints that constitute a consistent global state of the system. There is another checkpoint technology called communication-induced checkpoints which combines the benefit of these two categories checkpoint technologies.

In the distribution system, there is a parameter named recovery line which is a set of the most recently checkpoints. Because there are many processes in the distribution system so when we roll back the system we should roll back all these processes. The recovery line is used to arrange the recovery sequences of these processes [13].

2.3.4 Summary

For this project, we will use the middleware layer checkpoint because we do not want to change the application and we want to make it easier to identify the meaningful state of the application. The software simple-scalar will be used to simulate the function of check-pointing and identify the meaningful state of the application.

To develop the adaptive check-pointing technique, we do not need to use the distribution system so we assume the test is in a simple system. Therefore, we do not need to consider the coordinated check-pointing or independent check-pointing.

2.4 First order approximation for the optimum checkpoint interval

The first order approximation was developed by John.W.Young[2]. It is used to minimize the time lost because of failures and it can be used when the cost of the checkpoint is fixed and the error probability is fixed.

First, we assume the time between each two checkpoint is T_c , the overhead of the checkpoint is T_s as shown in Figure 2-3.

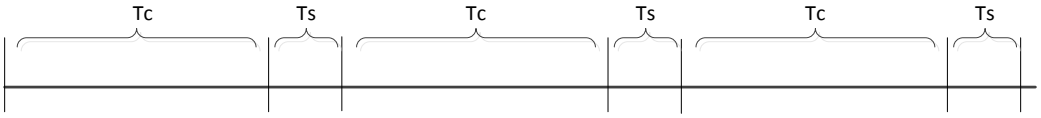


Figure 2-3 The Overhead T_s and checkpoint interval T_c

When there is an error, we need to roll back and rerun the execution. The rerun time t_r is the time from the end of the previous interval T_s to the point of failure as shown in Figure 2-4.

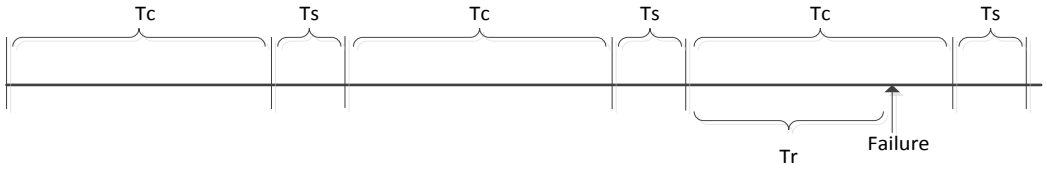


Figure 2-4 The rerun time t_r

We assume t_i is the interval between failures then t_i is composed of n intervals of length (T_s+T_c) plus the rerun time t_r as shown in Figure 2-5.

$$t_i = n(T_s + t_r) + t_r \text{ (Equation 1)}$$

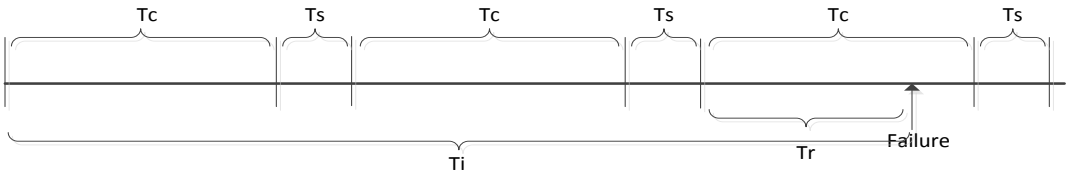


Figure 2-5 The interval between failures t_i

The lost time t_l due to the occurrence of failure consists of the time n T_s taken due to the check-pointing prior to the occurrence of the failure, plus t_r. As shown below

$$t_l = nT_s + t_r \text{ (Equation 2)}$$

Substituting Equation 1 into Equation 2, we obtain

$$t_l = t_i - nT_c \text{ (Equation 3)}$$

We assume the mean time between failures is T_f so the failure rate $\lambda = 1/T_f$, and we can use the density function to get the density of failures: $P(x) = \lambda e^{-\lambda x}$ which means the density of failures during the interval x [2]. So we can obtain the total lost time:

$$T_l = \sum_{n=0}^{\infty} \int_{n(T_c+T_s)}^{(n+1)(T_c+T_s)} [t - nT_c]\left(\lambda e^{-\lambda t_l}\right) dt$$

As the density of t_i is precisely the density of the lost timet_l so we obtain:

$$T_l = \sum_{n=0}^{\infty} \int_{n(T_c+T_s)}^{(n+1)(T_c+T_s)} [t - nT_c]\left(\lambda e^{-\lambda t}\right) dt$$

We can simplify the equation to

$$T_l = \frac{1}{\lambda} + T_c/[1 - \exp\left(\lambda(T_c + T_s)\right)]$$

To find the value of T_c which minimizes T_l , we differentiate T_l with respect to T_c and equate the result to 0.

$$\frac{1-\exp\left(\lambda(T_c+T_s)\right)-T_c(-\exp\left(\lambda(T_c+T_s)\right))}{[1-\exp(\lambda(T_c+T_s)]^2} = 0$$

This equation means that

$$e^{\lambda T_c} e^{\lambda T_s}(1 - \lambda T_c) - 1 = 0$$

Using the expansion of $e^{\lambda T_c}$ as far as the second degree term is concerned, we obtain

$$\frac{1}{2}\lambda^2 T_c^2 = 1 - e^{-\lambda T_s}$$

Since $T_f = \frac{1}{\lambda}$ and in practice $T_s \ll T_f$, we can use second order approximation to $e^{-\lambda T_s}$ to obtain

$$T_c^2 = 2T_s T_f - T_s^2$$

Neglecting the termT_s^2, we obtain $T_c = \sqrt{2T_s T_f}$ (Equation 4) [2]

From Equation 4, we note that the optimal fixed checkpoint interval is influenced by the mean time between failures and the overhead of the checkpoint. This equation can also be written as $T_c = \sqrt{2\frac{T_s}{\lambda}}$ and it can be used to calculate the optimal checkpoint interval when we know the failure rate and the overhead of the checkpoint.

2.5 Step check-pointing

Step check-pointing means that we take the checkpoint with the different checkpoint interval. Traditionally, we take the checkpoint with fixed interval because the checkpoint cost and the fault rate are fixed during the execution so the optimal length of all intervals is identical. However, under many circumstances, the failure rate is various; the actual number of errors may be different from the expected number of errors and the overhead of each checkpoint may be different. In these conditions, the fixed checkpoint interval is not a reasonable choice so we should use the step check-pointing. In this section, we will introduce two algorithms: on-line algorithm and step check-pointing algorithm. These two algorithms are used to solve different problems.

2.5.1 On-line algorithm [3]

In some systems, the cost of the checkpoint depends on the point in the program at which the checkpoint is taken. More specifically, the check-pointing cost depends on the size of the program's state at that point. The main idea of the on-line algorithm is to look for points in the program in which taking a checkpoint is the most beneficial. If such points are found, checkpoints are taken at these points with small intervals between the checkpoints so that the re-processing time after a fault is small and the overhead of the checkpoint is small. If no such point is found after a period of time, a checkpoint is placed at a point with higher cost to avoid a long re-processing time. In this case the interval between the checkpoints is longer to reduce the check-pointing overhead. [3]

To know the state size of the program in advance, the program can keep track of its state size by monitoring memory operations. By monitoring these operations, the program knows its current state size. Therefore, it can estimate the current cost of check-pointing.

2.5.2 Step check-pointing algorithm [5]

Under many circumstances, the failure rate is not evenly distribute. Most of the time, the system runs stably, but in certain periods, the system fails frequently. For example , mobile computer is often used in a complicated electromagnetism-related environment, the factors that cause the system to fail such as electromagnetism disturbance or noise often occur frequently in a certain period of time, so the failure rate of the system in that period is increasing suddenly. In another case, for software environment reasons, the system fails to malloc memory to the application program for the first time, when the system rolls back and re-executes, it is a high rate that the same malloc( ) fails again[5].

According to the changing failure rate environment, the step check-pointing algorithm is developed. Predicting that the failure rate of the system is abruptly increasing during a period of time after the system recovers from a failure; we set the checkpoint interval to a minimum value, and then increase it. The dense checkpoint setting can improve the performance of the system by reducing the waste of computation due to the frequent faults, and the exponentially increasing checkpoint interval may promptly normalize the checkpoint interval to its original length. This strategy can effectively shorten the rollback length and reduce the overhead [5].

Assume the fixed checkpoint interval is T; the minimum checkpoint interval is initialized to δ. The function to set the checkpoint interval is shown below

$$f(\gamma) = 2^{[\log_2(\frac{\gamma}{\delta}+1)]}\delta \quad [5]$$

This algorithm is shown below [5]

When system is running well behave
    Check-pointing with fixed checkpoint interval
When the system fails
    Recovery the system;
    The checkpoint interval is reduced to the minimum value δ;
When the system is in the period of check-pointing
    Set checkpoint
    $if(f(\gamma) \leq T)$
        The checkpoint interval is updated followed by the function $f(\gamma)$;
    Else
        The checkpoint interval set to be fixed T;

In the algorithm, there are three cases. When the system is running well behave, the checkpoint interval is fixed. When the system fails, the checkpoint interval will be initialized to δ. When the system needs to take a checkpoint, the system first takes the checkpoint then compares the step checkpoint interval with the fixed checkpoint interval to decide the checkpoint interval.

2.5.3 Summary

In this part, we introduce two types of step check-pointing algorithm. The first algorithm is used when the overhead of the checkpoint is various. The second algorithm is used when the failure rate is various. In our project, the overhead of the checkpoint is fixed and the failure rate is not various. However, we assume the actual failure rate is less than the expected failure rate.

In reality, the expected failure rate is the worst case. However, the actual failure rate sometimes may be less than the expected failure rate. So we assume the actual failure rate is less than the expected failure rate. In this case, the fixed check-pointing technique cannot work very well. To solve this problem, we need to design an adaptive check-pointing technique based on the step check-pointing algorithm. In the adaptive check-pointing technique, we should design the algorithm related to the actual failure rate.

2.6 System Architecture

2.6.1 Architecture

As we said previously, the checkpoint should save the information of the system state. However, the size of the system state is based on the system architecture. A simple system architecture is shown in Figure 2-6.

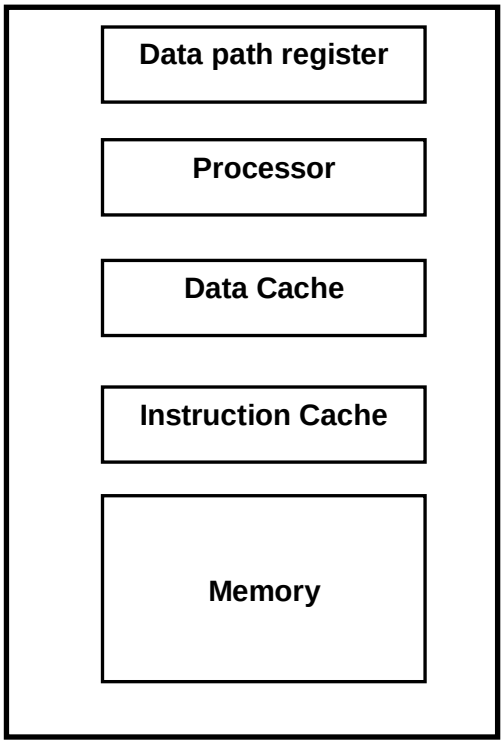


Figure 2-6 simple system architecture

Data path register

The data path is used to provide the routes for the data to transfer. It stores the information between the unit blocks. In general, the size is 512 bytes.

Processor

The processor is used to execute the instructions of the system and control the operation of the system. The processor execution state should be saved in the checkpoint. Now, the most commonly used processor is 16-bit and 32-bit.

Data cache & Instruction cache

Data cache is used to store data and the instruction cache is used to store instructions.

Memory

Memory contains the most information of the system so it will have a significant influence on the size of the checkpoint.

2.6.2 Summary

From this system architecture, we can set the overhead and roll back the time of the checkpoint. For example, we set the size of data path register x, the processor execution state is z and the memory size is y. To save all this information, the size of the checkpoint should be $f(x+y+z)$. In reality, the memory size is much greater than the processor execution state and data path register so we can neglect the processor execution state and data path register. Therefore the $size\ of\ checkpoint = f(y)$.

To calculate the overhead of the checkpoint, we set the processor to be a 32 bit processor then the overhead is $\frac{f(y)}{32}$. For example, if the memory size is 4M and the processor is 32 bits then the time needing to be taken is shown below

$$overhead\ of\ checkpoint = \frac{4M}{32} = 125000\ cycles$$

2.7Software

2.7.1 Introduction

ARM Workbench

The ARM workbench is an integrated development environment based on the open-source Eclipse3.3 IDE. It can be used to create, build, debug program flash, profile, trace and manage C/C++ for ARM processor based targets [6].

Code Warrior

CodeWarrior Development Studio is a complete integrated Development (IDE) that provides a highly visual and automated framework to accelerate the development of the most complex embedded application [7].

Simple-scalar tool set

Simple-scalar tool set performs a fast, flexible and accurate simulation of modern processors that implement the simple-scalar architecture. It consists of compiler, assembler, linker, simulation and visualization tools for the simple-scalar architecture [8].

The simple-scalar tool set has a power function on a checkpoint; it can take a checkpoint automatically and rerun any application from that checkpoint. However, the checkpoint can only be taken in the EIO file which can be converted from a binary file using the simplesim-3.0.

2.7.2 Comparison

Software	ARM Workbench	Code warrior	Simple-scalar tool set
Company	ARM	Freescale	Simple-scalar LLC
Price	Expensive	Expensive	Free
Function on check-pointing	weak	Weak	Strong
Function on tracing	Strong	Strong	Medium
Operating system	Windows	Windows	Linux

2.7.3 Summary

For this project, we will use the simple-scalar tool set. The reasons are shown below

1. Our project is to design the adaptive check-pointing technique. The roll back recovery and check-pointing are not the main points in this project so we should make them as simple as possible. The simple-scalar can do these two operations automatically.
2. There is no funding on this project and the other two software tools are very expensive.

2.7 Fault Detection

In electronic systems, the fault and error have distinctive meanings. A fault can be either a hardware defect or software/programming mistake (bug). However, there is a manifestation of the fault/failure so the fault/failure may cause errors and make the system works incorrectly [1].

2.8.1Hardware Fault

For this project we will focus on the hardware faults. In reality, the hardware faults are caused by the power, the environment and defect of the hardware. The hardware faults in an electronic system can be classified into three types: permanent fault, transient fault and intermittent fault. A permanent fault reflects the permanent going out of commission of a component such as a burned-out light bulb. A transient fault is one that causes a component to malfunction for some time and the component will work correctly after that time. An intermittent fault never quite goes away entirely, it oscillates between being quiescent and active [1, 9].There is also a parameter named component failure rate which is about the hardware fault of an individual component. The component failure rate, also known as hazard rate Z_t, is defined as the number of failures per unit time [1, 9]. The failure rate depends on the age of the component as shown in Figure2-7.

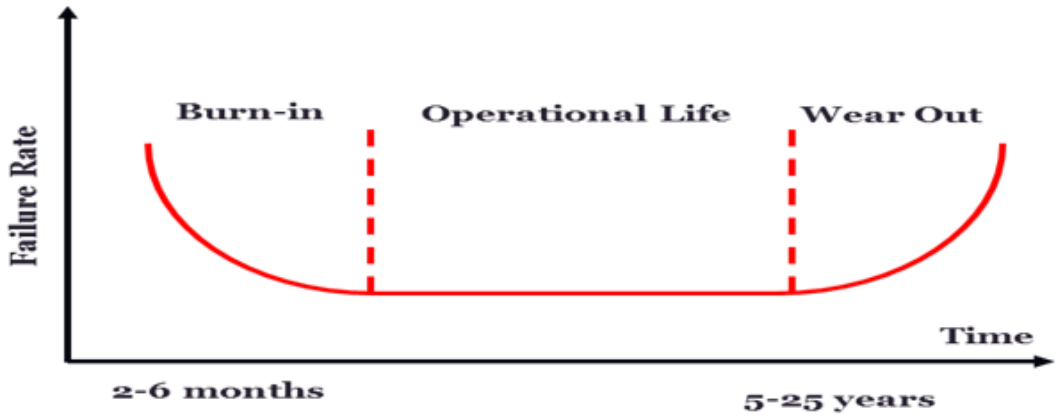


Figure 2-7 Bathtub curve [1]

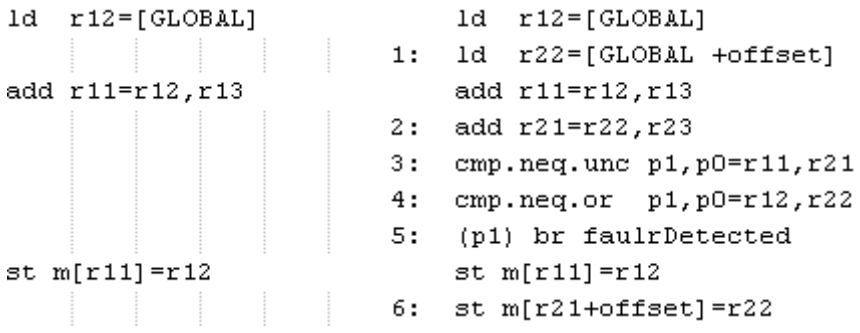
In Figure 2-7, the Burn-in period and Wear-out period get high failure because they do not work as intended. The middle phase (Operational life) gets the constant failure rate so we always use the middle phase to calculate the reliability.

2.8.2 EDDI

Error detection by duplicated instruction (EDDI) is a software-only fault detection system that operates by a duplicating program which should be checked. The duplicated program use different registers and different locations with the original program. The check instructions will be inserted to check if the original program and the duplicated program get the same result.[10]

Since program correctness is defined by the output of the program, then the program has executed correctly if all stores in the program have executed correctly [10]. So we use store

instruction as synchronization points for comparison. The example EDDI code is shown in Figure 2-8.



b. Original Code a. EDDI Code

Figure 2-8 EDDI codes [10]

In Figure 2-8, we duplicate the original instruction and use different registers such as r22 to store the same value of r12 .When we use r12, we also use r22 in EDDI code as instruction 2. When we use a store, we will do a comparison as instructions 3, 4. If any difference is detected instruction 4 will report an error. After comparison, instruction 6 will execute store.

2.8.3 Summary

For this project, we will inject the transient error so the error only occurs once. If we use the permanent error, the system cannot finish the application and it will roll back forever.

The test application is C code so when we duplicate the original code and build the EEDI codes; we should use the C code. There is no store in the C code, so we use Assignment, Arithmetic and Math as synchronization points for comparison.

3. Test method

3.1 System model

There are two main types of system: single process system and distributed system. In the single process system, there is only one process and we only need to monitor the execution of this process. The check-pointing and recovery technique are simply used in the single process system. In the distributed system, there are many processes and it is complex to use the check-pointing and recovery techniques because we should consider the global state which is a collection of individual states of all processes. Therefore, when we need to take checkpoint, we should take the checkpoint for each process and when we roll back to the previous state, we should identify the recovery line and roll back all these processes.

For this project, our aim is to design an adaptive check-pointing technique so we do not need to do complex check-pointing and recovery techniques. Therefore, we use a single process system model. The parameters of the system are shown in Table 3-1.

Test application	11557845 cycles
Memory size 1	4M
Memory size 2	6M
Memory size 3	8M
Error probability 1	$5 * 10^{-13}$
Error probability 2	$1 * 10^{-12}$
Fault detection ratio	0.58 (this will be explained later)
Processor	32 bit processor
Failure rate	error probability $*$ Memory size
Overhead of checkpoint	Memory size/32
Cost of recovery	Memory size/32
Cost of Fault detection	Application $*$ Fault fault detection

Table 3-1

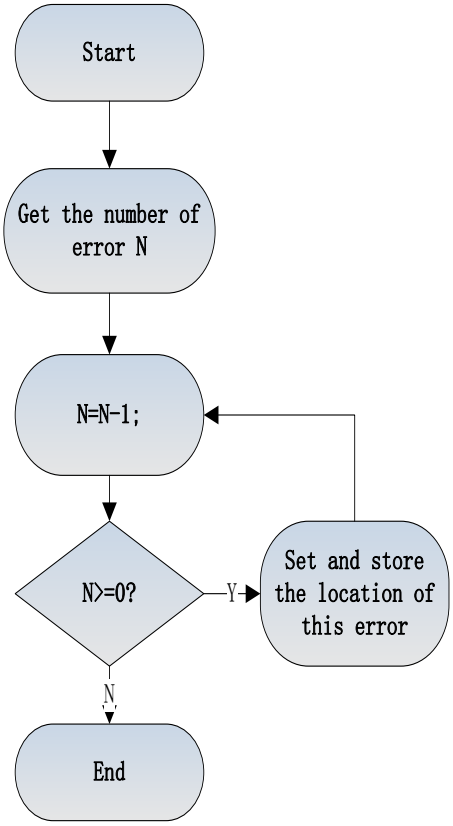
The test application is a program that will be executed in the system. The checkpoints will be taken during the execution of the test application. Every time, a checkpoint interval has finished, the system will detect the error. During the execution of checkpoint interval, the error flag will be used to mark the error. The roll back recovery and the action of taking checkpoint will be finished by the software simple-scalar.

The fault detection ratio is used to calculate overhead of the fault detection, we set it at 0.58(this will be explained later) and the overhead of fault detection can be calculated using 0.58*test application.

There are three different memory size and two different error probabilities. These parameters are used to test the check-pointing technique in system architecture.

3.2 Error Injection

To make the simulation more reliable, we assume the errors are uniformly distributed in the system. As the system model is simulated we cannot just inject the errors into the hardware. As we know, the error will stop the execution of the test application so we can inject the error into the cycles of the test application.



First, we get the number of errors from the error probability and the total cycles of the test application then we inject these errors in the application evenly distribute. The data flow diagram is shown in Figure 3-1.

In the data flow diagram, we set the locations of these errors and store these locations into the memory.

To make sure these errors are evenly distributed, we can use the GNU scientific library which can be downloaded from [14]. This library provides a wide range of mathematical routines such as random number generators, special functions and least-square fitting. The uniform distribution function gsl_ran_flat will be used to set the location of these errors. Actually, we can use another simple method to set the locations of these errors. According to the number of errors and the total cycles of the test application, we inject the error for every fixed cycle. The fixed cycles are equal to (test_application)/ (number of errors).

Figure 3-1 data flow diagram of error injection

3.3 Fault Detection

As we know the fault detection is not free, we cannot neglect the cost of fault detection. To get the cost of the fault detection, we build a new test application which contains the fault detection function (The code is shown in Appendix 1). In the new test application, we duplicate the original variable and compare the original one with the duplicated one when the system does Assignment, Arithmetic and Math. If they are different, we can declare that there is an error.

To duplicate the original value, we use a structured type which collects fixed labelled objects, possibly of different types variable into a single object as shown in Figure 3-2.

```
Struct-regint{
    int reg;
    int reg_copy;
    enum _booltype fault-detected;
}
```

Figure 3-2 the duplicate instruction

In the structure, we declare three variables: reg, reg-copy and fault-detected. The reg is the original variable and the reg_copy is the duplicated variable of reg. The fault-detected is used to detect error; if these two variables are equal, the fault-detected is false. Otherwise, the fault detected is true. When reg does Assignment, Arithmetic and Math, the duplicated also does the same operation. We test the new application and the new application is 18261395 cycles. So we can calculate the overhead of the fault detection

$$\text{overhead of fault detection} = 18261395 - 11557845 = 6703554$$

The fault detection ratio is $\frac{6703554}{11557845} = 0.58$

For this project, as the method to inject the error is special we can detect the error based on the locations of each error and use the fault detection ratio to calculate the overhead of fault detection. The data flow diagram is shown in Figure 3-3.

In the data flow diagram, we use the simple-scalar to get the range of this checkpoint interval first. Then, we check the location of each error, if the location of the error is in the range we increase the error count. When all errors have been checked, we will check the error count. When the error count is greater than 0, detect error. Otherwise, there is no error in this range .The overhead of fault detection in this range can be calculated using fault-detection ratio*range.

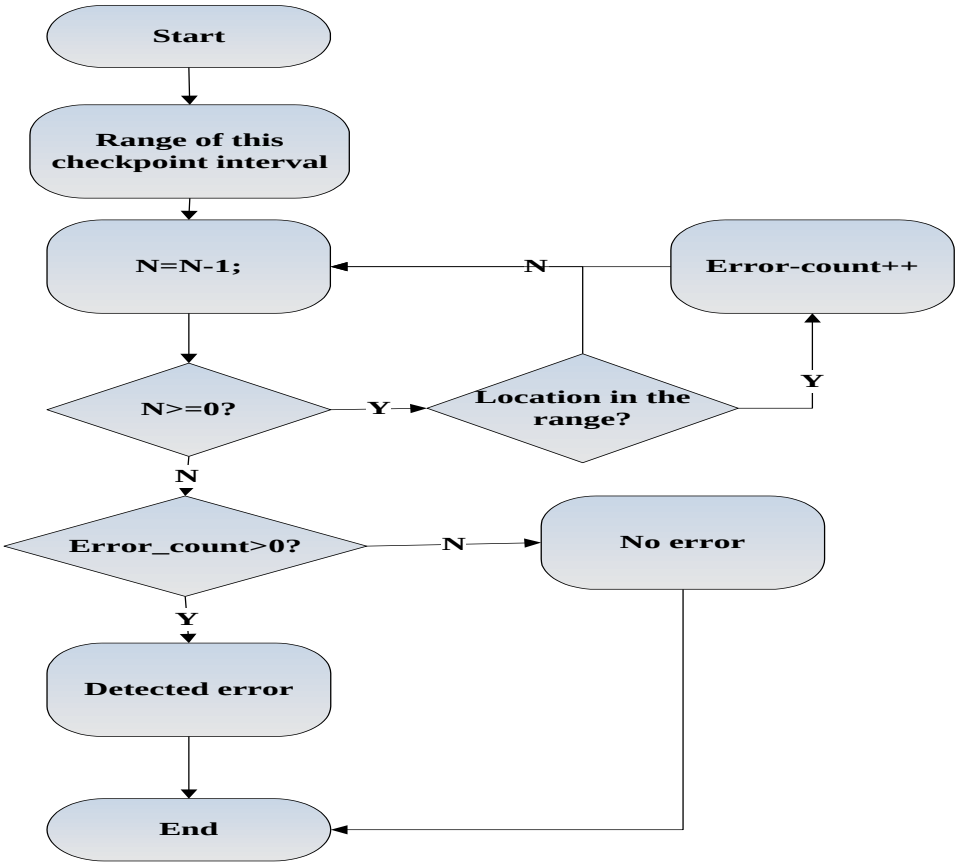


Figure 3-3 data flow diagram of fault detection

3.4 The use of the software

3.4.1 File conversion

In previous parts, we said that the simplesim-3.0 can take checkpoint in the EIO file. However, our test application is a C programme so we cannot take checkpoint in the test application directly. To convert the C file into EIO file, we use the sslittle-na-sstrix-gcc and the EIO simulator. The sslittle-na-sstrix-gcc can convert the C file into a binary file then the EIO simulator can convert the binary file into an EIO file. The steps are shown in Figure 3-4.

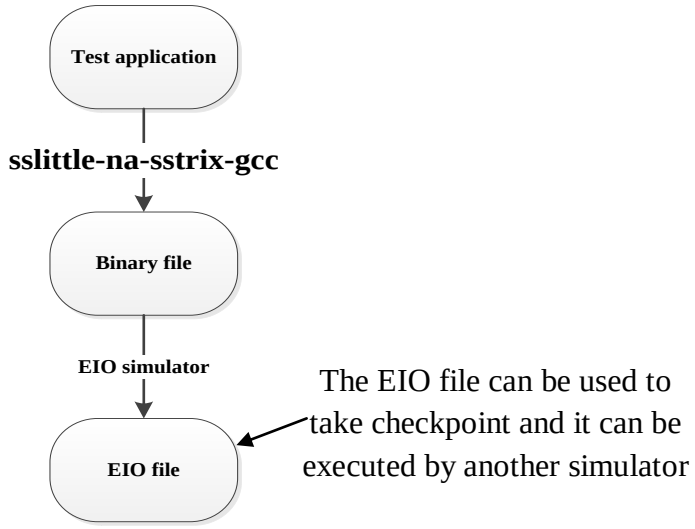


Figure 3-4 data flow diagram of file conversion

3.4.2 Simulator

There are many different types of simulator in simple-scalar: Sim-Fast, Sim-safe,Sim-Profile and so on. For this project, we will use the sim-eio and the sim-outorder simulator. The sim-eio is a new simulator, EIO traces capture initial program state and all subsequent external interactions a program has with the operating system [8]. We create the EIO file using the EIO simulator and the EIO file can be re-executed by any other simple-scalar simulator.

The out-of-order simulator is the most complicated and detailed simulator. This simulator supports out-of-order issue and execution. It will be used to re-execute the EIO file and provide the detail of execution. In the project, we will use the out-of-order simulator to get total instructions and total cycles of the test application.

3.4.3 Check-pointing and Re-execution

Check-pointing is the action to take a checkpoint. The re-execution means roll back recovery and rerun from a checkpoint. These two actions are very important in check-pointing technique. We use the simple-scalar to simulate these two functions.

Check-pointing

The simple-scalar can take checkpoint after a number of instructions so we can set the checkpoint interval as a number of instruction then we use the simple-scalar to take the checkpoint after this number of instructions. The command is shown below:

```
sim-eio -dump checkpoint.chkpt (number of instructions) : F00.eio
```

Name of checkpoint — The name of EIO file

In this command, we set a checkpoint at a number of instructions from file F00.eio. The name of the checkpoint is checkpoint.

Recovery and re-execution

The simple-scalar can re-run the application from the checkpoint with a number of instructions so when there is an error, we rerun the test application from the checkpoint and set the number of instructions equal to the checkpoint interval. This is used to simulate the re-execution. The command is shown below:

```
sim-outorder -max:(number Of instruction) -chkpt checkpoint.chkpt F00.eio
```

Name of checkpoint

In this command, we run the test application for a number of instructions from a checkpoint.

3.4.4 Limitation of simple-scalar

In reality, there are two methods to set the checkpoint interval unit. First is to use the instruction and the second is to use the cycles. In our project, cycle is a better choice because in our calculation the unit of the checkpoint interval is a cycle. However, the simple-scalar cannot take the checkpoint after a number of cycles; it can only take checkpoint after a number of instructions. Fortunately, the number of cycles is close to the number of checkpoints.

To make the simulation more accurate, we calculate the number of cycles for each checkpoint interval. The sim-outorder simulator is used to run the application for a checkpoint interval and read the number of cycles from the output file. However, if the test application is not executed from the beginning such as from a checkpoint, the number of cycles that in the output file is always 0. We send an email to the developer group of the simple-scalar for help but there is no answer for this. To solve this problem, we run the test application from the beginning two times as shown in Figure 3-5.

Figure 3-5 the method to calculate t cycles of the checkpoint interval

From these two executions, we can get the number of cycles $\mathbf{t_1}$ and $\mathbf{t_2}$. The cycles of the checkpoint interval is equal to $\mathbf{t_2 - t_1}$.

3.5 Result analysis and Curve fitting

In our program, the result will be written into output files. We should choose an appropriate method to analyse the result. The software EXCEL and MATLAB will be used. The EXCEL will be used to plot the cost for each check-pointing technique and the checkpoint interval. MATLAB will be used to do the curve fitting and find the relationship between the parameters. The trends of the result can also be analyzed using MATLAB. The software can be download from the website as shown below

http://office.microsoft.com/en-gb/home-and-student/

http://www.mathworks.co.uk/programs/nrd/matlab-trial-request.html?s_eid=ppc_1343

3.6 Simulation algorithm

To do the simulation, we should combine the error injection, fault detection, file conversion, check-pointing and re-execution together as appropriate. Therefore, we should design an algorithm to combine all of these. The parameters and the definitions are shown in Table 3-2.

Parameters	Definition
T_{OV}	Overhead of the checkpoint
I_T	Total instruction of the test application
T_C	Total cycles of the test application
I_{CP}	The length of checkpoint interval
T_{CP}	The cycles of current checkpoint interval
T_E	The extra cycles of the system
I_F	The finished instructions
F_{ratio}	The overhead of fault detection

Table 3-2

The data flow diagram of the simulation is shown in Figure 3-6 and the code is shown in Appendix 2. From the data flow diagram, we note that: at the beginning, we convert the test application into an EIO file then we use sim-outorder simulator to get the total instruction and total cycles of the test application. According to the T_C and failure rate we calculate the number of errors and inject these errors. These three steps are used to initialise the simulation.

When we finish initialization, we will go to a loop. We can divide the loop into 4 steps.

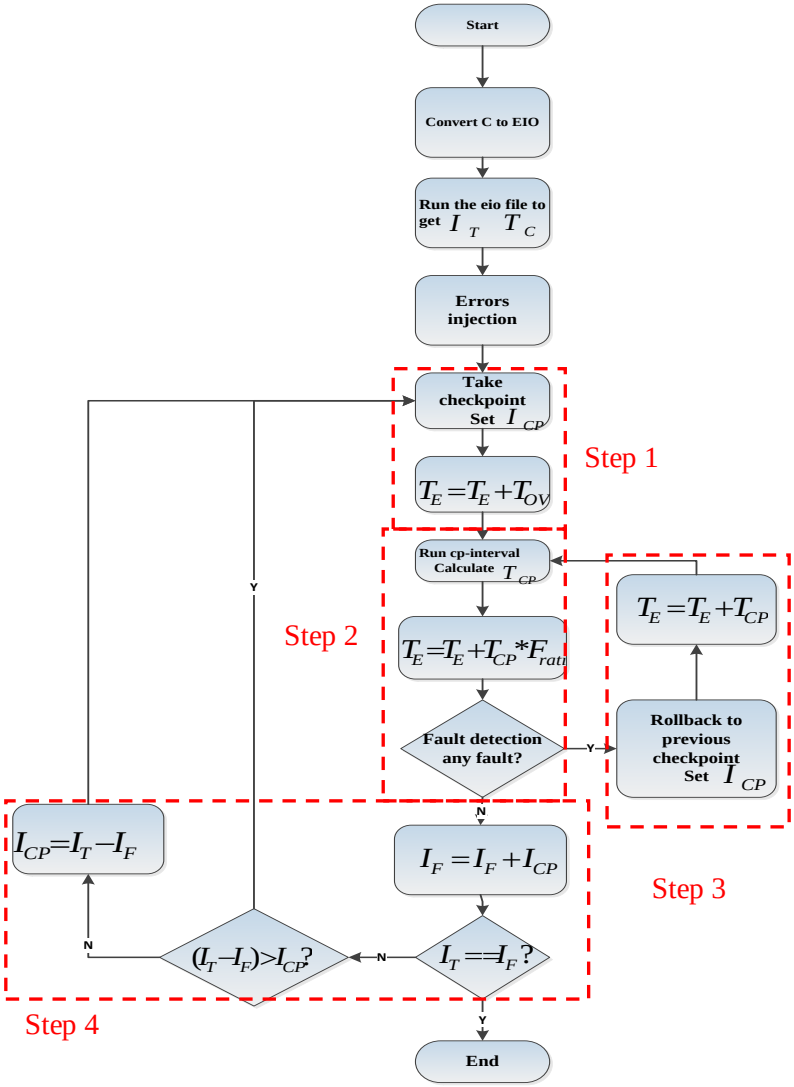


Figure 3-6 the data flow diagram of simulation

Step 1: Take a checkpoint and Set checkpoint interval I_{CP} then run the application I_{CP} long. After step1, we should increase the extra performance using $T_E = T_E + T_{OV}$ .Because of the overhead of the checkpoint.

Step 2: Calculate the cycles of the checkpoint interval T_{CP} and increase T_E using $T_E = T_E + T_{CP} * F_{ratio}$ because we should consider the cost of the fault detection. Then check the location of each injected error. Once the error is located in the checkpoint interval T_{CP}, increase the error counter. If the error counter is greater than 0, we go to step3. Otherwise, go to step 4

Step 3: we roll back the system to the last checkpoint and increase the extra performance using $T_E = T_E + T_{CP}$ because of the waste execution then set the checkpoint interval T_{CP} and rerun the test application from that checkpoint. Go to step 2

Step 4: As there is no error we increase the finished application using $I_F = I_F + I_{CP}$ and compare I_F with I_T. If I_F is equal to I_T, the application has been finished and go to end. Otherwise, if $(I_T - I_F)$ is greater than the I_{CP}, go to step 1. If $(I_T - I_F)$ is less than the I_{CP} set the $I_{CP} = (I_T - I_F)$ and go to step 1.

3.7Summary

In this section, we introduce the system model, the fault detection, the error injection and the use of the software tool simple-scalar. Although there is some limitation in the simple-scalar, we design approximate methods to solve these problems. The uses of other software are also introduced in this section. Finally, we combine all these parts and design the appropriate simulation algorithm.

4. Design & Experiment

4.1 Fixed check-pointing technique

In the fixed check-pointing technique, we set the checkpoint interval using an appropriate equation and take checkpoint every fixed checkpoint interval. The first order approximation which we introduced before used to minimize the time lost because of failures will be used to calculate the optimal checkpoint interval.

4.1.1 Experiment

We design three steps to test the fixed check-pointing technique. Step1 is to test the fixed check-pointing technique with different checkpoint intervals. Step 2 is to test the fixed check-pointing algorithm when the actual failure rate is different from the failure rate we expected. Step 3 is to test the fixed check-pointing algorithm with different memory size.

Step 1:

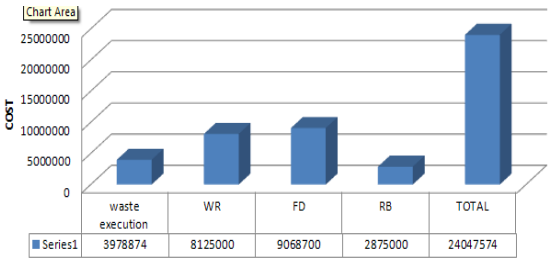
First, we use the first order approximation to set the checkpoint interval $\boldsymbol{I_{FA}}$ then we increase or decrease the checkpoint interval. The parameters are shown in Table 4-1.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	4M	0.58	11557845	$2 * 10^{-6}$	$2 * 10^{-6}$

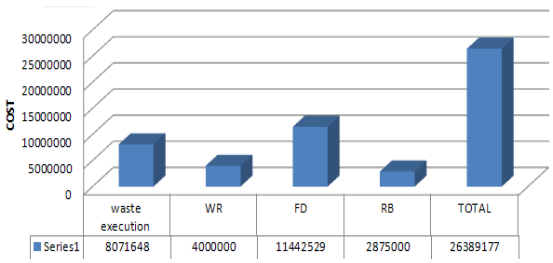
Table 4-1

We vary the checkpoint intervals from $0.3I_{FA}$ to 1.5 I_{FA} and do the simulation; the results are shown in Figure 4-1.

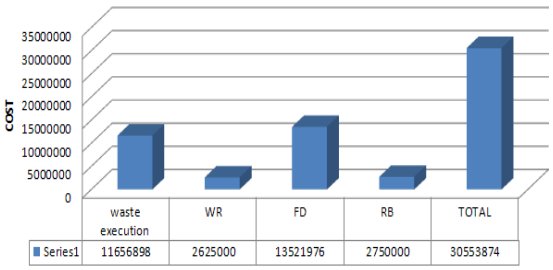
Waste execution: The execution that we roll back
WR: The cost of the checkpoint
FD: The cost of fault detection
RB: The cost of roll back



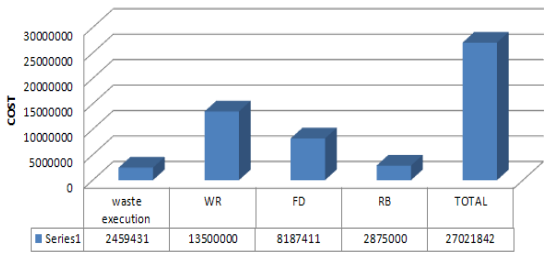
a. Checkpoint interval $0.5I_{FA}$



b. Checkpoint interval I_{FA}



c. Checkpoint interval $1.5I_{FA}$



d. Checkpoint interval 0.3 I_{FA}

Figure 4-1 the distribution of cost with different checkpoint intervals

From this Figure4-1, we can get 3 theories:

Theory1:

From Figure 4-1 a and Figure 4-1b, we note that the total cost of Figure 4-1a is less than Figure 4-1 b which means checkpoint interval $0.5I_{FA}$ is better than the checkpoint interval I_{FA} .This is because we use the flag based error checking in the simulation. The first order approximation is used for persistent error checking so it cannot work very well for this simulation. To compensate this, we develop a new equation to calculate the optimal checkpoint interval as shown below.

$$T_c = \frac{\sqrt{2T_sT_f}}{2} \qquad \text{(Equation 5)}$$

Theory 2:

When we investigate Figure 4-1d, we note that the total cost of Figure 4-1d is higher than figure 4-1a, the WR is very high and the waste execution is low. This is caused by the short checkpoint interval. When the checkpoint interval is short, we need to take more checkpoints during the execution then the overhead of the checkpoint is very high. As we use the flag based error checking the execution we roll back is equal to the checkpoint interval so the waste execution is low. With the same number of errors, the short checkpoint interval will decrease the waste execution and increase the WR.

Theory 3:

When we investigate Figure 4-1c, we note that the total cost of Figure 4-1c is higher than Figure 4-1a, the waste execution is very high, the WR is very low and the RB is lower than others. This is caused by the long checkpoint interval. When the checkpoint interval is long, the execution we roll back is long, so the waste execution is high. However, the long checkpoint interval will reduce the number of checkpoints and reduce the roll back times. For example .when there are three errors in the same checkpoint interval, we will reduce the roll back times from 3 to1 then the RB will be reduced.

Step 2

The fixed check-pointing technique is a good strategy for check-pointing technique when the failure rate and the overhead of the checkpoint are known. However, in reality, the actually failure rate is different from the failure rate we expected. The expected failure rate is always the worst case and the actual failure rate is smaller than the expected failure rate. In this test, we set the expected failure rate to be twice of the actual failure rate and do the simulation. The parameters are shown in Table 4-2. The result is shown in Figure 4-2.

Parameter	Error probability	Memory size	Cost of Fault detection	Test application	Expected λ	Actually λ
Value	$1 * 10^{-12}$	4M	0.58	11557845	$4 * 10^{-6}$	$2 * 10^{-6}$

Table 4-2

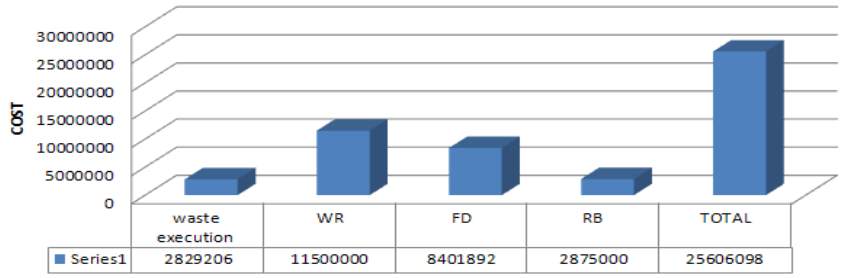


Figure 4-2 expected failure rate is different form the actual failure rate

From Figure 4-2 and Figure 4-1a, we note that the WR is very high, this is because when the actually failure rate is higher than the expected failure rate the calculated checkpoint interval is smaller than the optimal checkpoint interval so there are more checkpoints than we expected and the cost of checkpoint is very high. Therefore, we should design an adaptive check-pointing technique to provide adaptive checkpoint interval when the actual failure rate is less than the expected failure rate.

Step 3

In reality, there are many different size of memory .So we will test the fixed check-pointing technique with 6M and 8M memory in this test. We repeat step 1 for 6M memory size and the parameters are shown in Table 4-3.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	4M	0.58	11557845	$2 * 10^{-6}$	$2 * 10^{-6}$

Table 4-3

To create a new equation for the optimal checkpoint interval, we plot the relationship between the checkpoint interval and the total cost as shown in Figure 4-3.

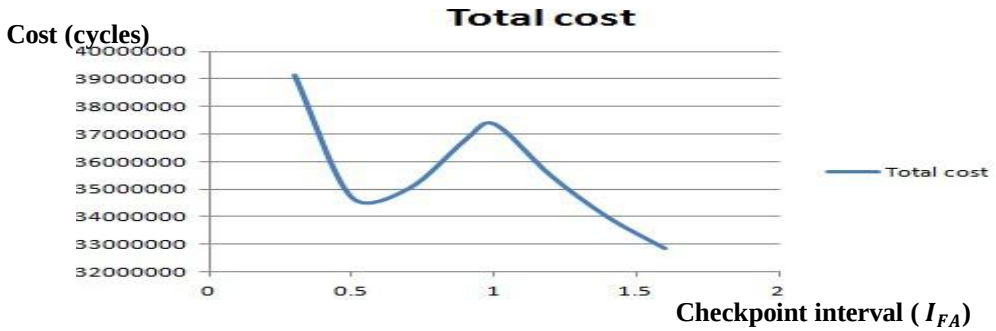
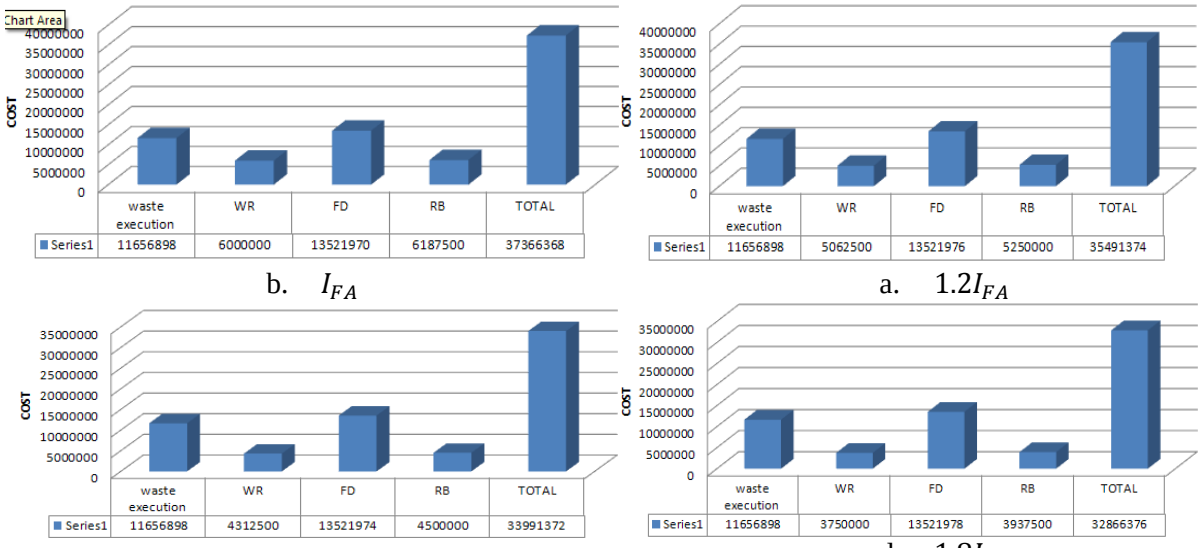


Figure 4-3 The relationship between checkpoint interval and the total cost

From figure4-3, we note that the best checkpoint interval is about 0.55 $\boldsymbol{I_{FA}}$ when the checkpoint interval is less than $\boldsymbol{I_{FA}}$. However, when the checkpoint interval is longer than $\boldsymbol{I_{FA}}$, the total cost will decrease when we increase the checkpoint interval. This is different from 4M memory size. To analyse this difference, we plot the cost distribution for checkpoint

interval I_{FA}, checkpoint interval $1.2I_{FA}$, checkpoint interval $1.4I_{FA}$, and checkpoint interval $1.6I_{FA}$. The figures are shown in Figure 4-4.

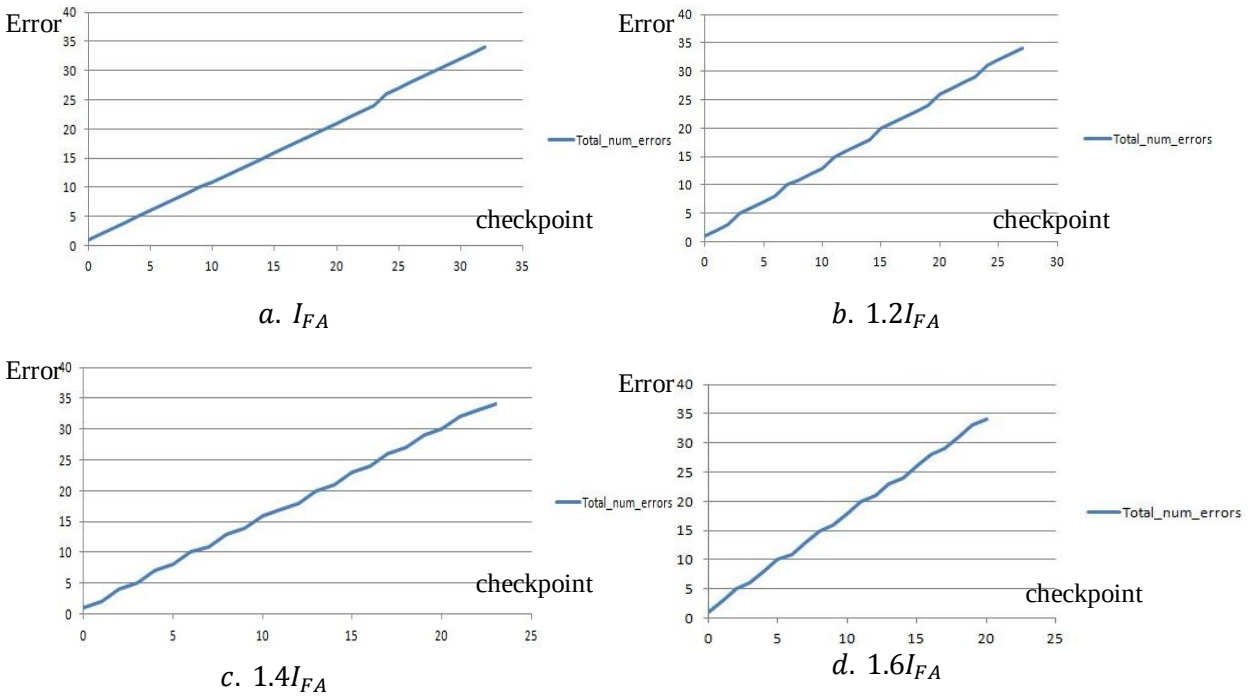


b. I_{FA} a. $1.2I_{FA}$

c. $1.4I_{FA}$ d. $1.8I_{FA}$

Figure 4-4 cost distribution when checkpoint interval greater than I_{FA}

From Figure4-4, we note that the waste computations of all checkpoint intervals are identical, equal to the total cycles of the test application. This means the total test application is rolled back, for every checkpoint interval we roll back to its last checkpoint and rerun this checkpoint interval again. To prove this assumption, we plot the error distribution as shown in Figure 4-5.



a. I_{FA} *b.* $1.2I_{FA}$

c. $1.4I_{FA}$ *d.* $1.6I_{FA}$

Figure 4-5 Error distribution

From Figure 4-5, we note that although the increase speed is various, the number of errors increases between each two checkpoints. Therefore, we need to roll back each checkpoint interval. We did not get this problem for 4M memory size; this is because 4M memory gets fewer errors than 6M and 8M memory with the same error probability and same test application. In reality, the system cannot get so many errors in such a short test application. To test 6M and 8M we must reduce the number of errors. The parameters for 6M and 8M are shown in Table 4-4 and 4-5. In these two tables, we decrease the error probability.

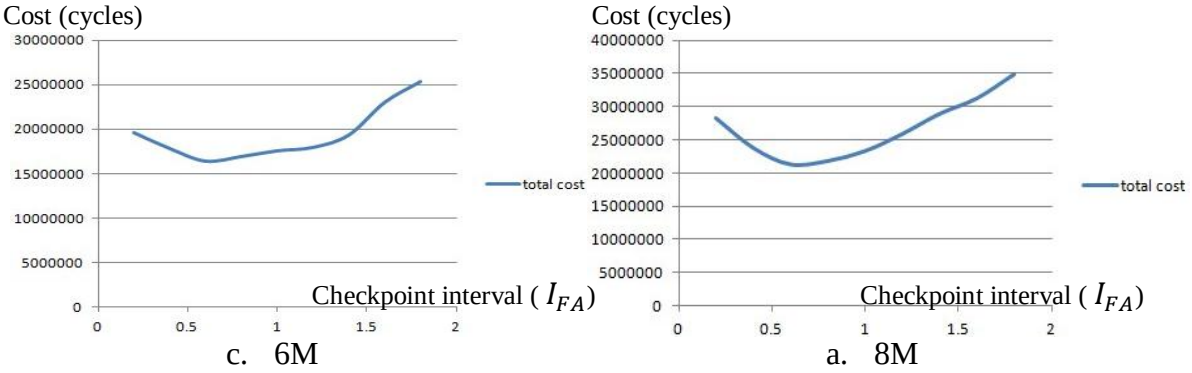
Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$1 * 10^{-13}$	6M	0.58	11557845	$6 * 10^{-7}$	$6 * 10^{-7}$

Table 4-4

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$1 * 10^{-13}$	8M	0.58	11557845	$8 * 10^{-7}$	$8 * 10^{-7}$

Table 4-5

We do the simulation with these new parameters and plot the relationship between the total cost and the checkpoint interval as shown in Figure 4-6.



c. 6M a. 8M

Figure 4-6 the relationship between the checkpoint interval and total cost for 6M and 8M

From Figure 4-6 a, the lowest total cost is at 0.62 I_{FA}so we can develop an equation for optimal checkpoint interval: $T_c = 0.58\sqrt{2T_F T_S}$ (Equation 6)

From Figure 4-6 b, the lowest total cost is at 0.63 I_{FA} so we can develop an equation for optimal checkpoint interval: $T_c = 0.6\sqrt{2T_F T_S}$ (Equation 7)

These two equations are very close to each other. To explain this, we should consider two aspects. First, the larger memory size will increase the overhead of the checkpoint and cost of roll back so we should increase the checkpoint interval for larger memory size system to reduce the cost of checkpoints. Second, with the same error probability and same test application, the large memory size will cause more errors so we should decrease the checkpoint interval to reduce the waste computation. Therefore, the equation is very close to each other.

4.1.2 Summary

The fixed check-pointing technique takes the checkpoint every fixed checkpoint interval. We test the fixed check-point technique with different checkpoint intervals and develop the equation for the optimal checkpoint interval of 4M memory size.

$$T_c = \frac{\sqrt{2T_sT_f}}{2}$$

When the checkpoint interval is long, the waste execution and the cost of fault detection are high but the cost of the checkpoint is low.

When the checkpoint interval is short, the cost of checkpoint is high but the cost of fault detection and waste execution are low.

For different memory size, we get the equation for the optimal checkpoint interval and these equations are similar to each other. This is caused by the increased overhead of checkpoint and the increased number of errors.

When the expected failure rate is greater than actual failure rate the fixed check-pointing technique is not a good choice. The total cost will be high.

4.2 Adaptive check-pointing #1: Inspired by Young [2]

From Young's first order approximation, we know that there is an optimal checkpoint interval for each system and we develop equations to calculate the optimal checkpoint interval in section 4.1. The first adaptive check-pointing technique is to find the optimal checkpoint interval using the equation we developed. The main idea of the first adaptive check-pointing technique is to set the checkpoint interval every time we finish error detection. According to the number of detected errors and the finished cycles of the test application we calculate the new checkpoint interval using the equation. The parameters and the definitions are shown in Table 4-6.

Parameters	Definition
T_F	Finished cycles of the total test application cycles
NMTTF	Normal mean time to failure
MTTF	Mean time to failure
N_E	The total number of errors we have found
P_e	Error probability
M	Memory size
I_{cp}	Checkpoint interval
I_{icp}	Initial checkpoint interval
T_{OV}	The overhead of the checkpoint

Table 4-6

The NMTTF is the normal mean time to failure and it can be obtained using the equation:

$$\lambda = P_e * M$$
$$NMTTG = 1/\lambda$$

The MTTF will be various with the detected number of errors and finished cycles T_F . We can obtain MTTF using equation: $MTTF = T_F/N_E$.
The initial checkpoint interval $\mathbf{I_{icp}}$ for 4M memory can be calculated using equation 5:

$$I_{icp} = \sqrt{2 * NMTTF * T_{OV}}/2$$

The data flow diagram of first adaptive check-pointing technique is shown in Figure 4-7.(The code is shown in Appendix 3)

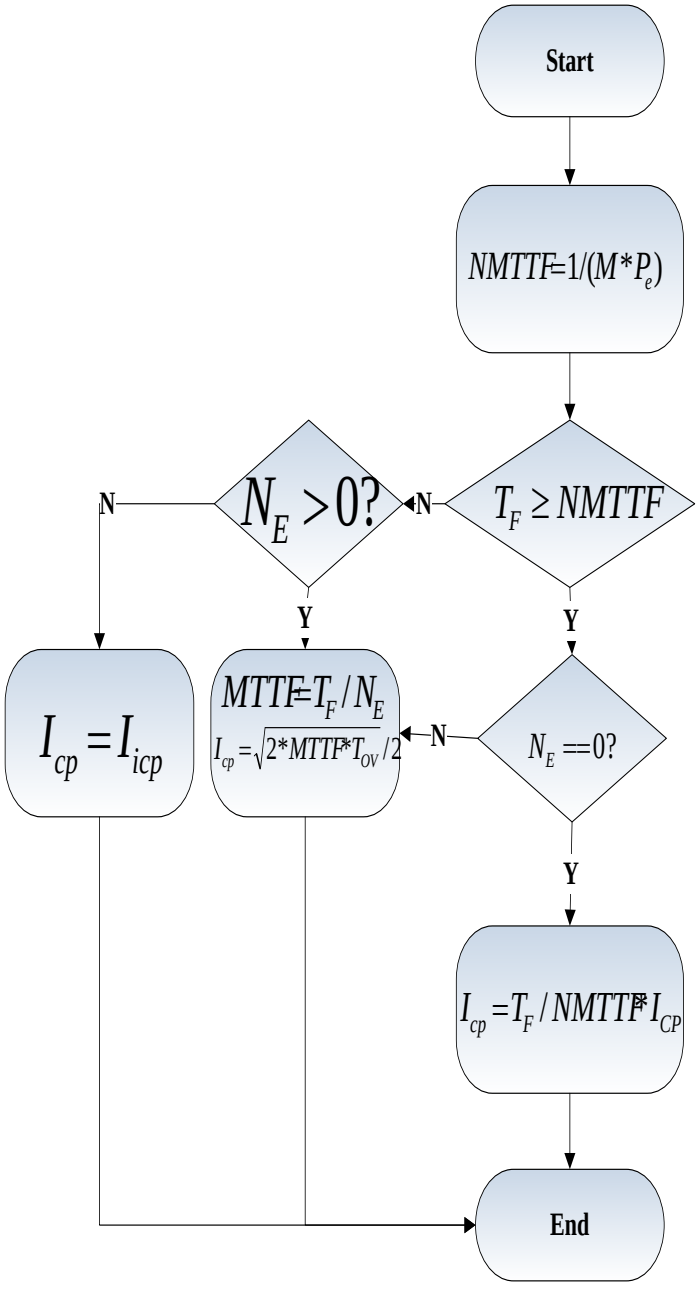


Figure 4-7 data flow diagram of first adaptive Check-pointing technique

First, we calculate the NMTTF using the memory size and the error probability. Then we set the checkpoint interval. There are three different methods to set the checkpoint interval. N_E and T_F will decide which methods are to be used. There are four cases.

Case1: T_F is less than the NMTTF and N_E is greater than 0.First, we use N_E and T_F to calculate the MTTF then we use equation 5 to calculate the checkpoint interval.

Case 2: T_F is less than the NMTTF and N_E is equal to 0. As we are not sure how many errors will be detected before T_F increases to NMTTF we assign the initial checkpoint interval to the checkpoint interval.

Case3: T_F is not less than the NMTTF and N_E is equal to 0. As there is no error detected, the MTTF will be infinite so if we use equation5 to calculate the checkpoint interval, the checkpoint interval will be infinite. Therefore, we use another method to increase the checkpoint interval. We compare T_F and NMTTF then increase the checkpoint interval according to their ratio.

Case 4: T_F is no less than the NMTTF and N_E is greater than 0. We calculate the MTTF then use the equation to calculate the checkpoint interval.

4.2.1 Experiment

4.2.1.1 4M

To test the first adaptive check-pointing technique, we set the parameters as shown in Table 4-7 and do the simulation. The distribution of cost is shown in Figure 4-8.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$\mathbf{1 * 10^{-12}}$	4M	0.58	11557845	$\mathbf{4 * 10^{-6}}$	$\mathbf{2 * 10^{-6}}$

Table 4-7

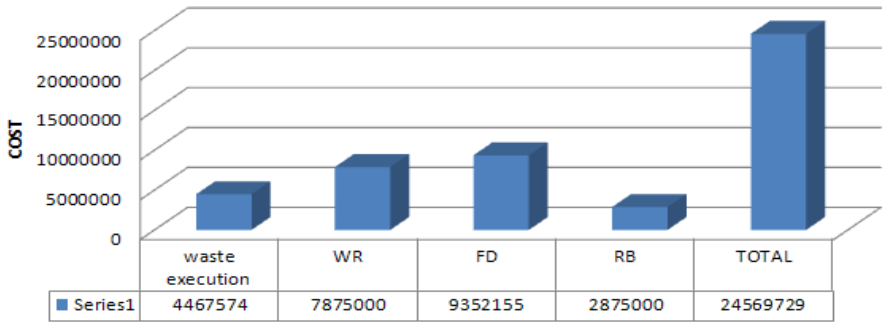


Figure 4-8 the cost distribution of first adaptive check-pointing algorithm

We compare Figure 4-8 with Figure 4-2; we note that the WR of Figure 4-8 is lower than Figure 4-2 and the waste execution is higher than Figure 4-2. The total cost is lower than Figure 4-2.According to conclusion 2; the long checkpoint interval can decrease WR and increase waste execution so the first adaptive check-pointing technique can adjust the checkpoint interval. To prove this assumption, we plot the checkpoint interval of the adaptive check-pointing technique as shown in Figure 4-9.

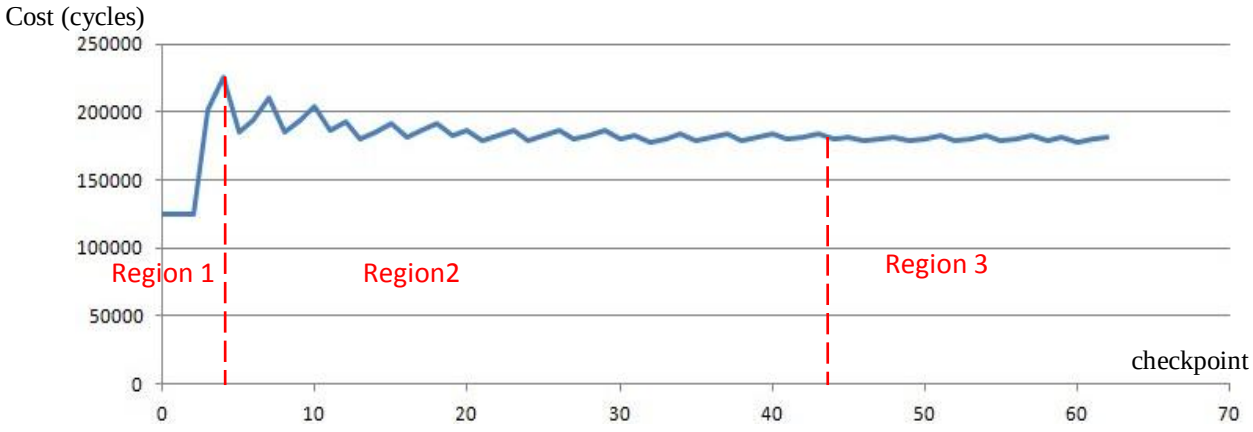


Figure 4-9 the checkpoint interval of adaptive check-pointing technique

In Figure 4-9, we can divide the checkpoint interval line into three regions:

In the first region, the checkpoint interval will keep constant for a short time then increase significantly. This is because at the beginning the system is in case 2, the checkpoint interval is equal to the initial checkpoint interval. When T_F is greater than NMTTF, the system will go to case 3. When an error occurs, the checkpoint interval will be calculated using the equation5 and the system will go to case 4.

In the second region, the system is in case 4. The checkpoint interval will increase when these is no error detected in this checkpoint interval. However, if an error is detected, the checkpoint interval will decrease. Therefore, the checkpoint interval oscillates around the optimal checkpoint interval and the oscillation will be small with the execution time.

In the third region, the checkpoint interval oscillation is very small and we can neglect the oscillation, the checkpoint interval is almost 19000.

4.2.2.2 for 6M and 8M memory size

In the fixed check-pointing technique section, we have got the equation for 6 M and 8M as shown below.

$$T_c = 0.58\sqrt{2T_F T_S}\ (6\text{M})$$

$$T_c = 0.6\sqrt{2T_F T_S}(8\text{M})$$

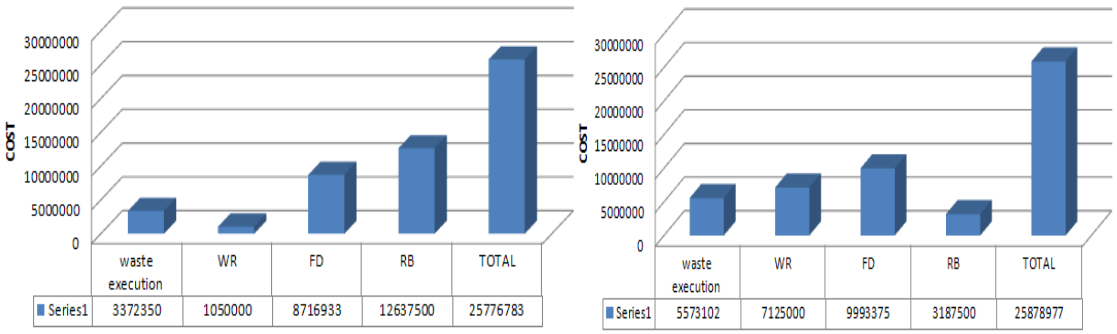
We set the parameters as shown in Table 4-8, 4-9 and do the simulation using fixed check-pointing technique and first adaptive check-pointing technique separately. The result is shown in Figure 4-10 and figure 4-11.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	6M	0.58	11557845	$3 * 10^{-6}$	$1.5 * 10^{-6}$

Table 4-9 6M

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	8M	0.58	11557845	$4 * 10^{-6}$	$2 * 10^{-6}$

Table 4-10 8M



a. Fixed check-pointing technique b. First adaptive check-pointing technique

Figure 4-10 the fixed and first adaptive check-pointing technique for 6M

a. Fixed check-pointing technique d. adaptive check-pointing technique

Figure 4-11 the fixed and first adaptive check-pointing technique for 8M

In Figure 4- 10, the total cost of the fixed check pointing technique is less than the total cost of first adaptive check-pointing technique. To analyse this problem, we plot the checkpoint interval of 6 M memory size as shown in Figure 4-12. From Figure 4-12, we note that the checkpoint interval is oscillating around 300000 and the final checkpoint interval is almost 30000 so the optimal checkpoint interval the first adaptive check-pointing provides is 300000.

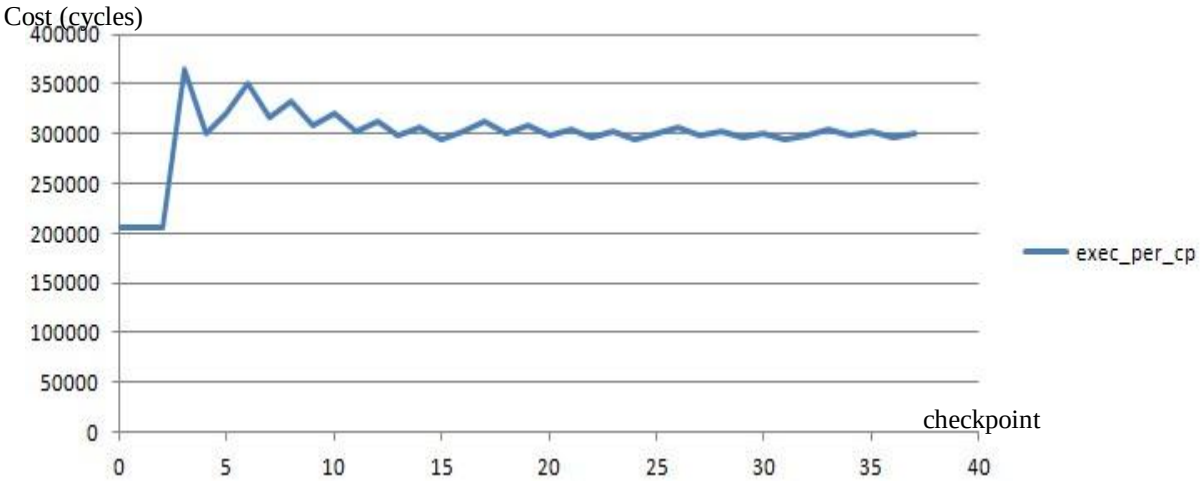


Figure 4-12 the checkpoint interval for 6M memory

Now, we find the optimal checkpoint interval when the error probability is$5 * 10^{-13}$. The parameter is shown in Table4-11.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$2.5 * 10^{-13}$	6M	0.58	11557845	$1.5 * 10^{-6}$	$1.5 * 10^{-6}$

Table 4-11

We vary the checkpoint interval and check the total cost. The plot is shown in Figure 4-13.

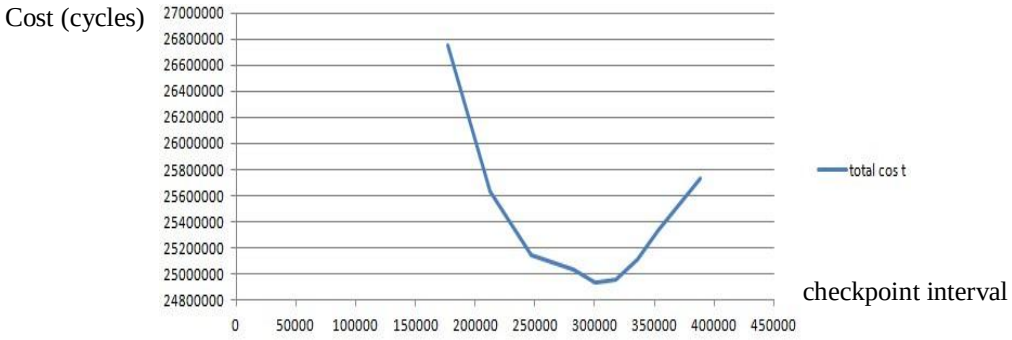


Figure 4-13 the optimal checkpoint interval for 6M memory

From Figure 4-13, we note that the optimal checkpoint interval is almost 30000. This means that the first adaptive check-pointing technique adjusts the checkpoint interval to optimal checkpoint interval successfully. From Figure 4-13, we note that when the checkpoint interval is 300000 the total cost is about 24850000 which is less than Figure 4-10 a. Therefore, the reason that makes the total cost of Figure 4-10b higher than Figure 4-10a is caused during the process of adjusting the checkpoint interval. We plot the cost for each checkpoint of fixed check-pointing technique and first adaptive check-pointing technique as shown in Figure 4-14.

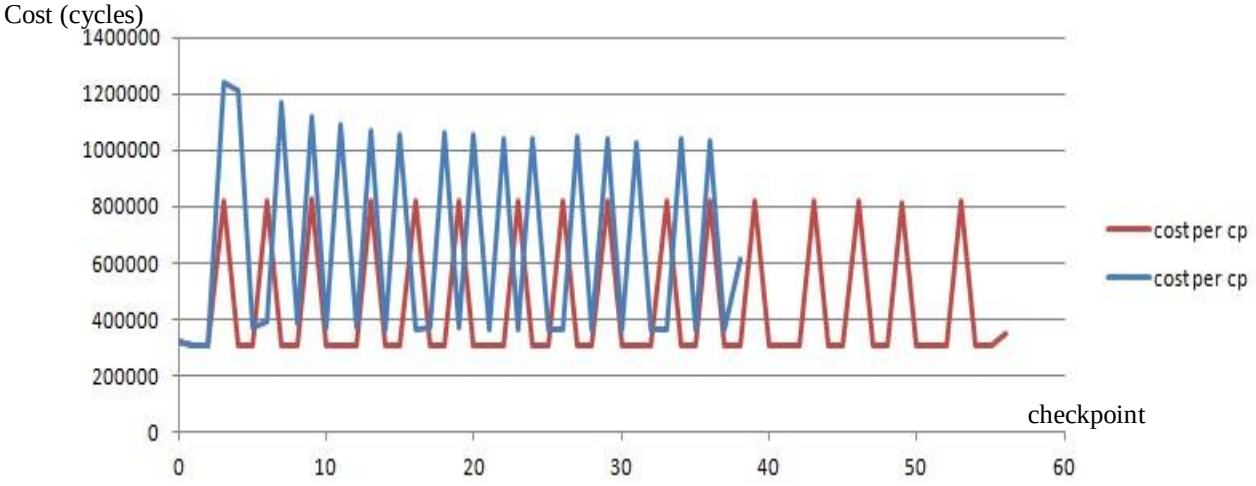


Figure 4-14 cost for each checkpoint interval

The cost for each checkpoint means the cost from this checkpoint to next checkpoint. This cost contains the cost of FD, the cost of RB, the cost of checkpoint and the waste execution during this interval. In Figure 4-14, the blue line represents the first adaptive check-pointing technique and the red line represents the fixed check-pointing technique. For the blue line, the maximum cost is very high at the beginning and this is the reason for the high total cost.

When the checkpoint interval is adjusted to the optimal checkpoint interval, the cost will be less than the fixed check-pointing algorithm. Therefore, if the test application is longer, the test application will be executed with the optimal checkpoint interval for a long time then the total cost should be less than the total cost of the fixed check-pointing technique.

4.2.2 Summary

This adaptive check-pointing technique is based on the equation to calculate the checkpoint interval so the equation is very important for this adaptive check-pointing algorithm. When we do not know the optimal equation of the system, we should not use this adaptive check-pointing technique.

The first adaptive check-pointing technique can adjust the checkpoint interval to be the optimal checkpoint interval during the execution and when we compare Figure 4-8 with Figure 4-5, we note that the first adaptive check-pointing technique is better than the fixed check-pointing technique when the expected failure rate is greater than the actual failure rate. The first adaptive check-pointing technique is better for long test application. When the test application is short, the effectiveness of the first adaptive check-pointing technique is very small.

4.3 Adaptive check-pointing #2: MTTF based algorithm

The first adaptive check-pointing technique is based on the equation so we cannot use the first adaptive check-pointing technique when we do not know the equation. Therefore, we design the second check-pointing technique which is based on the MTTF. In the second adaptive check-pointing technique, we also set the checkpoint interval every time we finished fault detection. The difference is that we change the checkpoint interval based on the MTTF and a variable x. First, we compare MTTF with the NMTTF to know if the expected failure rate is higher or less than the actual failure rate then we compare the MTTF with the MMTTF(Maximum Mean time to failure) to decide whether to increase or decrease the checkpoint interval. The parameters and definitions are shown in Table4-12.

Parameters	Definition
T_F	Finished cycles of the total test application cycles
NMTTF	Normal mean time to failure
MTTF	Mean time to failure
N_E	The total number of errors we have found
M	Memory size
I_{cp}	Checkpoint interval
I_{icp}	Initial checkpoint interval
T_{OV}	The overhead of the checkpoint
P_e	Error probability
MMTTF	Max MTTF we calculated
x	The increase/decrease percentage of the checkpoint interval
I_{scp}	Stable checkpoint interval

Table 4-12

The MMTTF is the maximum MTTF during T_F

The x is the increase/decrease percentage of the checkpoint interval, every time we set the checkpoint interval, the checkpoint interval will increase or decrease x percentage.

The MMTTF and I_{cp} will be initialised before we use the algorithm.

The MMTTF can be updated by longer MTTF

The data flow diagram of the second adaptive check-pointing algorithm is shown in Figure 4-15.(The code is shown in Appendix 4)

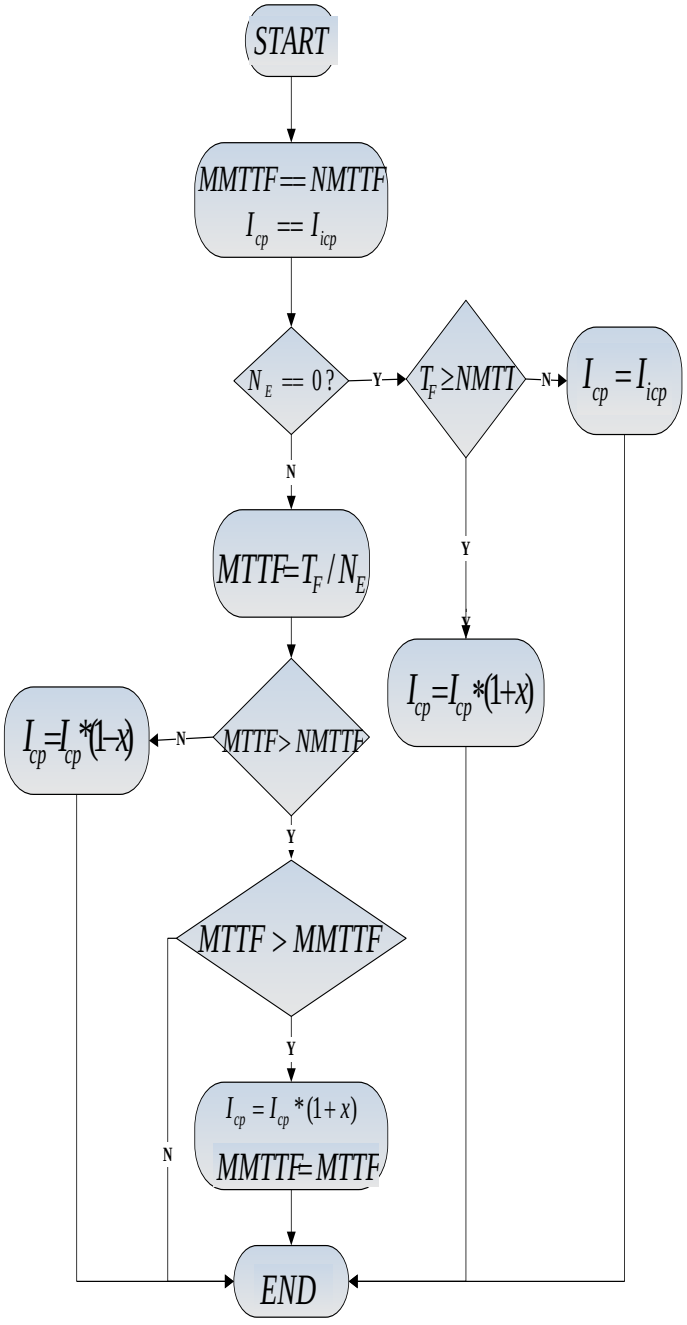


Figure 4-15 the data flow diagram of second adaptive check-pointing technique

In the data flow diagram, we initial the MMTTF and the checkpoint interval. Then there are 4 different methods to set the checkpoint interval. The N_E , T_F MTTF and MMTTF will decide which method to be used. There are 5 cases.

Case 1: N_E is equal to 0 and T_F is greater than the NMTTF. We will increase the checkpoint interval because when N_E is equal to 0 the MTTF is infinite and it must be greater than NMTTF and MMTTF so we increase the checkpoint interval.

Case 2: N_E is equal to 0 and T_F is less than the NMTTF. As T_F is less than the NMTTF we are not sure how many errors will be detected when T_F is equal to NMTTF. Therefore, we will not increase or decrease the checkpoint interval so we assign the initial checkpoint interval to the checkpoint interval.

Case 3: N_E is not equal to 0 and MTTF is less than the NMTTF. We detect errors before T_F reaches NMTTF which means there may be more than 1 error when T_F reaches NMTTF so we decrease the checkpoint interval to reduce the waste of computation.

Case 4: N_E is not equal to 0,MTTF is greater than the NMTTF and MTTF is greater than the MMTTF. We compare the MTTF with NMTTF to check the failure rate. When MTTF is greater than the NMTTF, the actual failure rate is less than the expected failure rate. Then we compare MTTF with MMTTF to decide if we should increase the checkpoint interval. When the MTTF is greater than the MMTTF, we increase the checkpoint interval and assign MTTF to MMTTF. The parameter MMTTF is very important for this technique. If there is no MMTTF and we increase the checkpoint interval every time MTTF>NMMTTF

the checkpoint interval will be infinite and the waste computation will be unpredictable so we use the MMTTF to limit the checkpoint interval.

Case 5: N_E is not equal to 0, MTTF is greater than the NMTTF and MTTF is less than the MMTTF. We do not change the checkpoint interval.

4.3.1Test

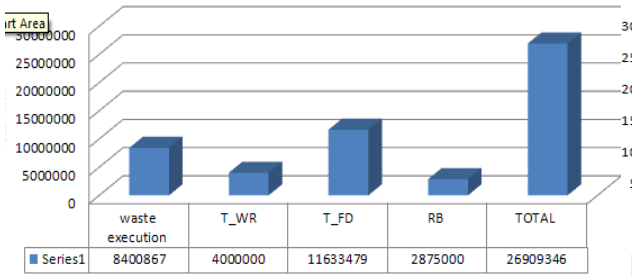
There are three steps to test the second adaptive check-pointing technique. Frist, we test the technique with different variable x. Second, we test the relationship between the initial checkpoint interval and variable x. Third, we test the algorithm with a different memory size.

Step 1
We initialise the checkpoint interval and the MMTTF; the parameters are shown in Table4-13.

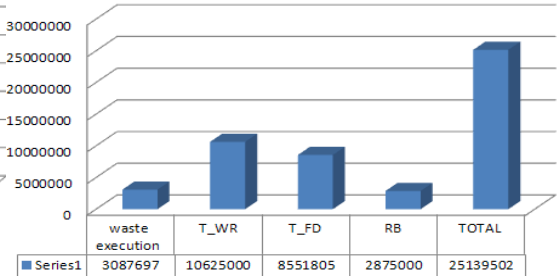
Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$1 * 10^{-12}$	4M	0.58	11557845	$4 * 10^{-6}$	$2 * 10^{-6}$

Table 4-13

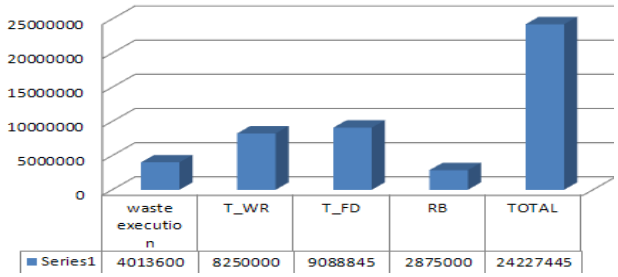
We vary x from 0.01 to 0.25 and do the simulation, the cost distribution is shown in Figure 4-16.



a. x=0.25



b. x=0.01



c. x=0.05

Figure 4-16 Cost distribution with different x

From Figure 4-16, we note that the different x will produce different total cost. The lowest total cost of Figure4-16 is 24227445 which is less than the total cost of Figure 4-2. This means the second adaptive check-pointing technique is better than the fixed check-pointing technique.

In Figure4-16, the large x will cause high waste execution and low WR. The small x will cause high WR and low waste execution. In theory 2 and theory 3, we said that the

checkpoint interval can influence the WR and waste execution. So we assume that the x will influence the checkpoint interval. To prove this assumption, we plot the checkpoint interval as shown in Figure 4-17.

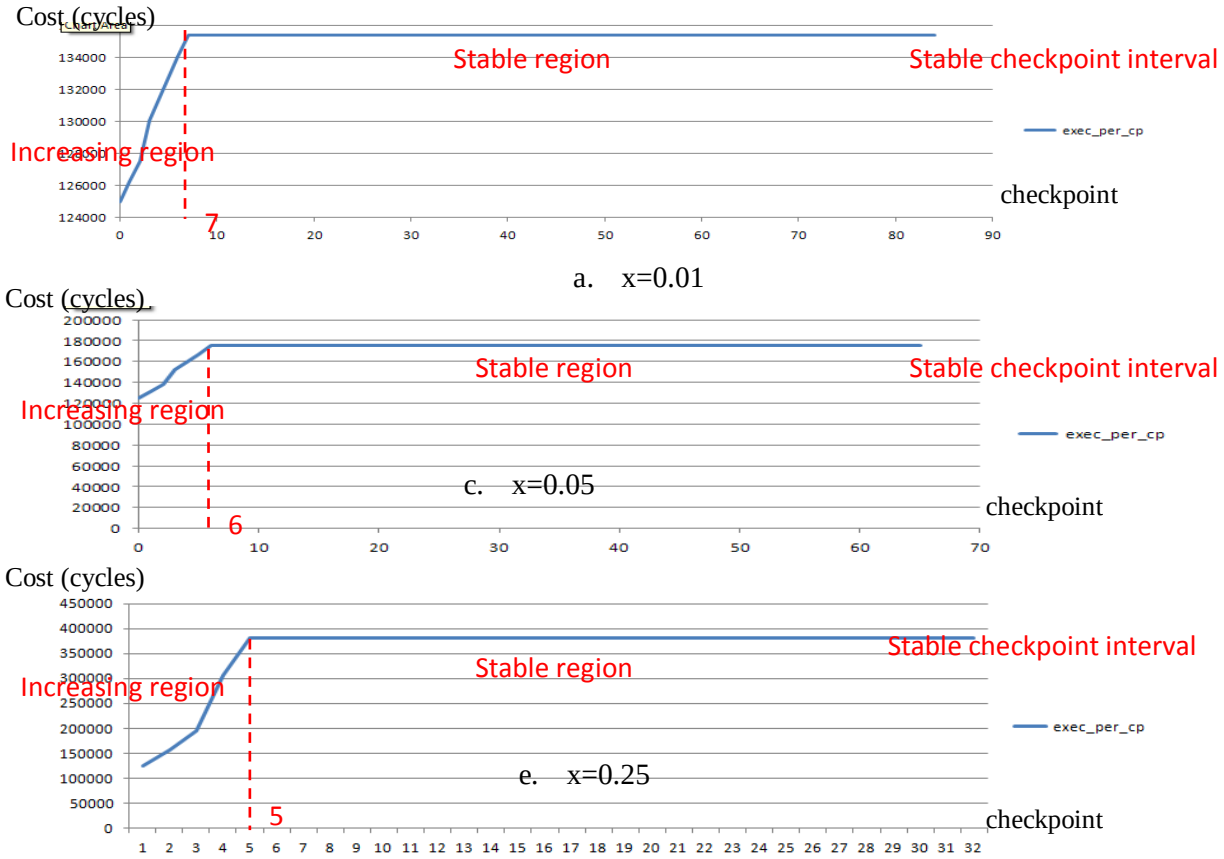


Figure 4-17 checkpoint intervals with different x

From Figure 4-17, we can conclude two theories.
Theory 4:

When x is equal to 0.01, the $\boldsymbol{I_{scp}}$ is 135354. When x is equal to 0.05 the $\boldsymbol{I_{scp}}$ is 175884.When x is 0.025 the $\boldsymbol{I_{scp}}$ is 381468. Therefore, when variable x is large, the stable checkpoint interval is long. On the other side, the stable checkpoint interval is short when the x is small.

Theory5:

We can divide the plot into two main regions: increasing region and stable region. In the increasing region, we find that for different x, the stable checkpoint interval will be reached at a different number checkpoint. When x is 0.01, the stable checkpoint interval is reached at a seventh checkpoint. When x is 0.05, the stable checkpoint interval is reached at the sixth checkpoint. When x is 0.25 the stable checkpoint interval is reached at the fifth checkpoint. Therefore, when x is large the stable checkpoint interval will be reached at a small number of checkpoints. When x is small, the stable checkpoint interval will be reached at a big number of checkpoints.

Step2

In step1, we get theory 4 and theory 5. The variable x will influence the stable checkpoint interval so there must be an appropriate variable x for each initial checkpoint interval to provide an optimal checkpoint interval. Now, we test the relationship between x and checkpoint interval. First, we vary the checkpoint intervals from 50000 to 250000 then find the optimal x for these checkpoint intervals; the results are shown in table 4-14 to table 4-18.

x	checkpoints in increasing region	Total cost	Final cycles before fixed	Final checkpoint interval
0.1	11	25939437	897999	129683
0.18	9	24481985	849416	187938
0.19	9	24478364	876044	201062
0.2	9	24607250	902097	214989
0.25	8	24616153	812202	238416

Table 4-14 Initial checkpoint interval = 50000

x	checkpoints in increasing region	Total cost	Final cycles before fixed	Final checkpoint interval
0.09	6	24071985	993952	199253
0.1	5	24065122	833401	194871
0.11	5	24162692	855945	207613
0.12	5	24117688	899781	221065

Table 4-15 Initial checkpoint interval = 100000

x	checkpoints in increasing region	Total cost	Final cycles before fixed	Final checkpoint interval
0.03	4	24004042	838223	179106
0.04	4	24030717	855601	189796
0.05	4	23968389	877455	201013
0.06	4	23983494	909003	212776
0.1	4	24639441	1007709	265733

Table 4-16 Initial checkpoint interval = 150000

x	checkpoints in increasing region	Total cost	Final cycles before fixed	Final checkpoint interval
0.001	3	23963020	832530	200999
0.007	3	23942319	839972	205048
0.01	3	23886315	846510	210201
0.05	3	24465888	916068	255256

Table 4-17 Initial checkpoint interval = 200000

x	checkpoints in increasing region	Total cost	Final cycles before fixed	Final checkpoint interval
0.0001	2	24228408	780364	250100
0.0005	2	24210794	780687	250500
0.002	2	24188639	781255	250999
0.005	2	24320617	787269	255035
0.01	2	24427151	793660	260150

Table 4-18 Initial checkpoint interval = 250000

From these 5 tables, we can conclude three theories.

Theory 6:

When the initial checkpoint interval is long, the best x should be small. Otherwise, when the initial checkpoint interval is short the best x should be big.

Theory 7:

From these 5 tables, we note that when the initial checkpoint interval is long, only a few number of checkpoints are in the increasing region. Therefore, when the initial checkpoint interval is long, the stable checkpoint interval will be reached at a small number of checkpoints. Otherwise, when the checkpoint interval is short, the stable checkpoint interval will be reached at a big number of checkpoints.

Theory8:

In the first 4 tables, we note that although all these 4 tables have a similar stable checkpoint interval, the total cost is different. The smaller initial will get a larger total cost. This is caused by the increasing region. When the initial checkpoint interval is small, we will take more checkpoints before in the increasing region. For example, when initial checkpoint interval is 50000, we take 9 checkpoints before we reach a stable checkpoint interval. When initial checkpoint interval is 200000, we only take 3 checkpoints before we reach the stable checkpoint interval. The more checkpoints we take, the higher the cost of the checkpoint. Therefore, if the initial checkpoint interval is close to the optimal stable checkpoint interval the total cost is small.

To investigate the relationship between the x and the initial checkpoint interval, we use the curve fitting to plot the relationship as shown in Figure 4-18.

Initial checkpoint interval	50000	100000	150000	200000	250000
Optimal x	0.19	0.1	0.05	0.007	0.002

Table 4-19

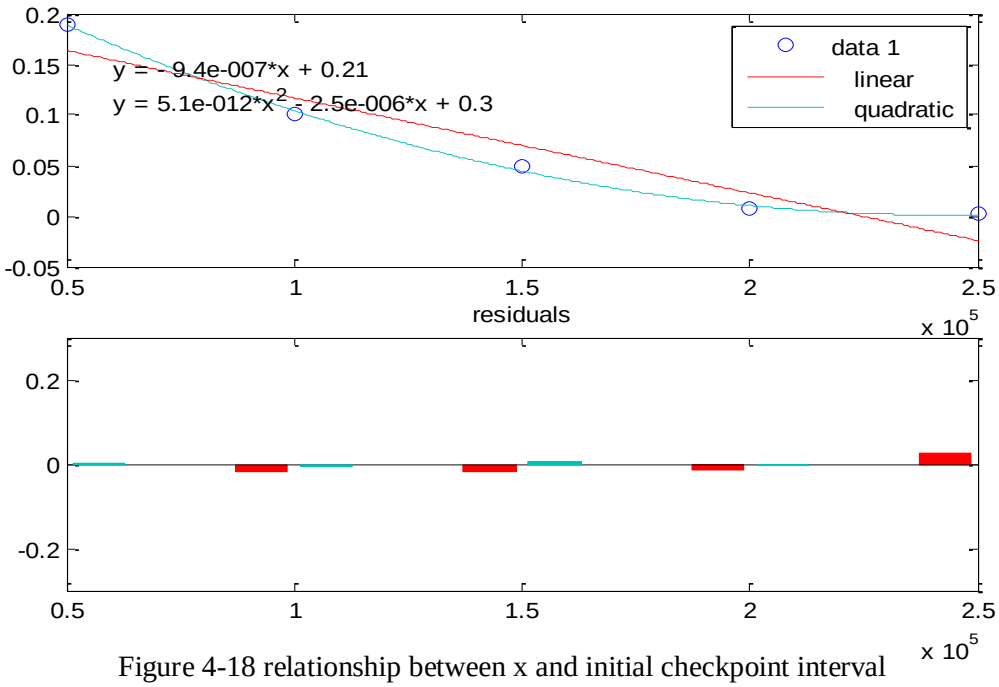


Figure 4-18 relationship between x and initial checkpoint interval

In Figure 4-18, there are two lines. The red line is created using the linear curve fitting and the green line is created using the quadratic curve fitting. We compare the residuals of these two lines and find that the green line is more accurate than the red line. So we use the equation $y = 5.1e-012x^2 - 2.5e-006x + 0.3$ to represent the relationship between the x and the initial checkpoint interval.

From the green line, we note that when the initial checkpoint interval increases the variable x will decrease. However, in these 5 tables, we increase the initial checkpoint interval 50000 for each table; the decrease quantity of the variable x is becoming smaller and smaller as shown in Table4-20.

Table	Initial checkpoint interval	Increase quantity	Variable	Decrease quantity
Table1	50000		0.19	
Table2	100000	50000	0.1	0.09
Table3	150000	50000	0.05	0.05
Table4	200000	50000	0.007	0.043
Table5	250000	50000	0.002	0.005

Table 4-20

This condition cause a quadratic line .To explains this relationship. First, we should consider the theory 7. In theory 7, we said that the number of checkpoints in the increasing region will decrease when the initial checkpoint interval increases. As we know, in the adaptive check-pointing algorithm, we set the new checkpoint interval every time we finished the error detection and the error detection is before check-pointing. Therefore, the number of checkpoint will influence the times we set the checkpoint interval. Now, we can get conclusion 9: The long checkpoint interval will provide fewer times to set the initial checkpoint interval. As the times we increase the checkpoint interval is fewer, we cannot decrease the variable the same quantity every time. Finally, the relationship is quadratic.

Step 3

In step1 and step2, we test second adaptive check-pointing technique for 4M memory size and the algorithm works effectively. Now, we will test the second adaptive checkpoint technique for 6M and 8M memory size. The parameters are shown in Tables4-21 and 4-22.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	6M	0.58	11557845	$2 * 10^{-6}$	$1 * 10^{-6}$

Table 4-21

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$5 * 10^{-13}$	8M	0.58	11557845	$2 * 10^{-6}$	$1 * 10^{-6}$

Table 4-22

We do the simulation and plot the relationships for 6M and 8M as shown in Figure 4-19.

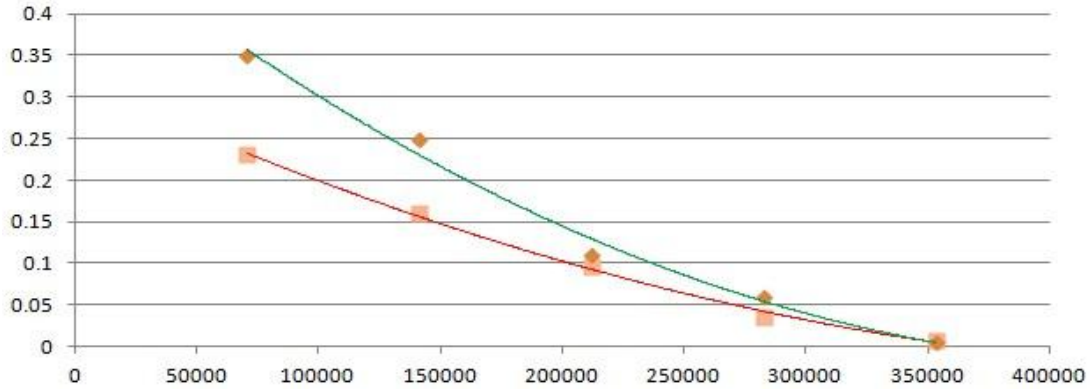


Figure 4-19 the relationship of x and initial checkpoint interval for 6M and 8M

In Figure4-19, there are two quadratic lines. The green line represents 8M and the red line represents 6M. From these two lines, we note that for different memory size the optimal x is different. However, the trend of these two lines is the same. Therefore, for different memory size, we should use different equations to represent the relationship between x and the initial checkpoint interval.

4.3.2 Summary

The second adaptive check-pointing technique is based on the MTTF and variable x. We compare the second adaptive check-pointing technique with fixed check-pointing technique. The second adaptive check-pointing technique can provide a better fault tolerance and produce less total cost. The relationship between the initial check-point interval and the variable x is quadratic. For different memory size, the equation for the relationship is different but the trend is the same. Finally, we conclude 5 theories:

1. When variable x is large, the stable checkpoint interval is long. On the other side, the stable checkpoint interval is short when the x is small.

2. When x is large the stable checkpoint interval will be reached at a small number of checkpoints. When x is small, the stable checkpoint interval will be reached at a big number of checkpoints.

3. When the initial checkpoint interval is long, the best x should be small. Otherwise, when the initial checkpoint interval is short the best x should be big.

4. When the initial checkpoint interval is short, the stable checkpoint interval will be reached at a big number of checkpoints. When the initial checkpoint interval is long, the stable checkpoint interval will be reached at a small number of checkpoints.

5. If the initial checkpoint interval is close to the optimal stable checkpoint interval the total cost is small.

5. Comparison and analysis

Now, we have introduced two adaptive check-pointing techniques. One is based on the equation and the other one is based on MTTF and variable x. To compare these two algorithms, we assume the actual number of errors is 1/3 the expected number of errors and do simulation with fixed check-pointing technique, adaptive check-pointing techniques separately. The parameters are shown in Table4-23.

Parameter	Error probability	Memory size	Fault detection ratio	Test application	Expected λ	Actually λ
Value	$1 * 10^{-12}$	4M	0.58	11557845	$4 * 10^{-6}$	$1.3 * 10^{-6}$

Table 4-23

First, we do the simulation with fixed check-pointing algorithm. The result is shown in Figure 4-20.

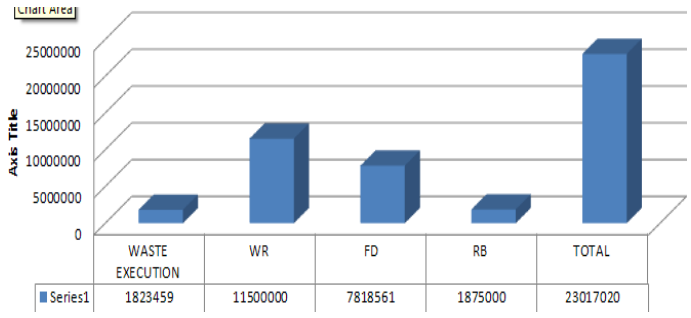


Figure 4-20 the distribution of cost for fixed check-pointing technique

Second, we use the equation $T_c = \frac{\sqrt{2T_sT_f}}{2}$ to calculate the checkpoint interval and do the simulation with first adaptive check-pointing technique. The result is shown in Figure 4-21.

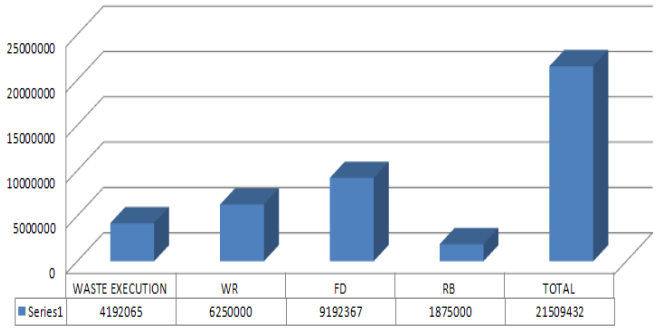


Figure 4-21 the distribution of cost for first adaptive check-pointing technique

Third, we assume the initial checkpoint interval is equal to the fixed checkpoint interval so we can obtain the initial checkpoint interval:

$$Initial\ checkpoint\ interval = \frac{\sqrt{2 * \frac{M}{32 * M * P_e}}}{2}$$

As $P_e = 1 * 10^{-12}$ and $M = 4000000$, we can obtain $I_{icp} = 125000$.
Now, we have got the initial checkpoint interval. To get the optimal x we will use the equation we get from the curve fitting and the result is shown in Figure 4-22.

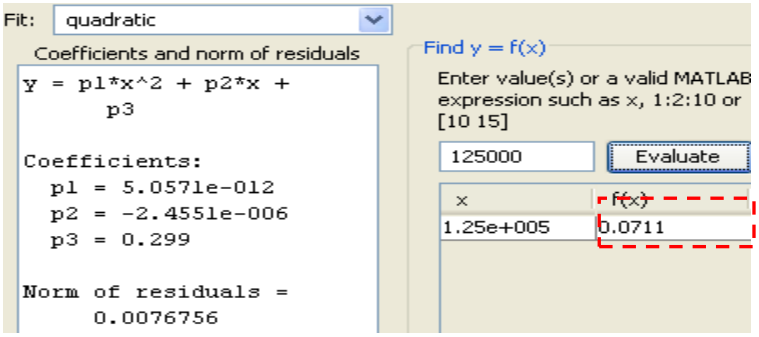


Figure 4-22 the optimal x value for the initial checkpoint interval

We do the simulation and the result is shown in Figure 4-23.

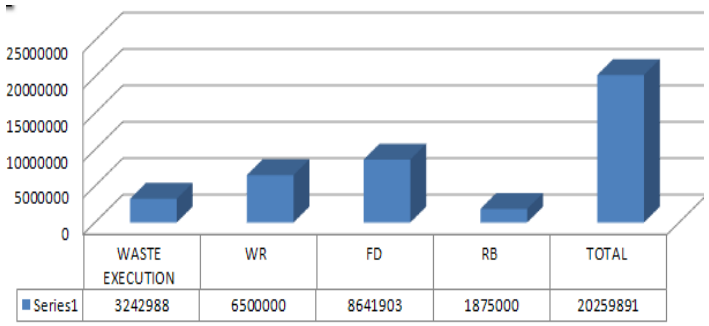
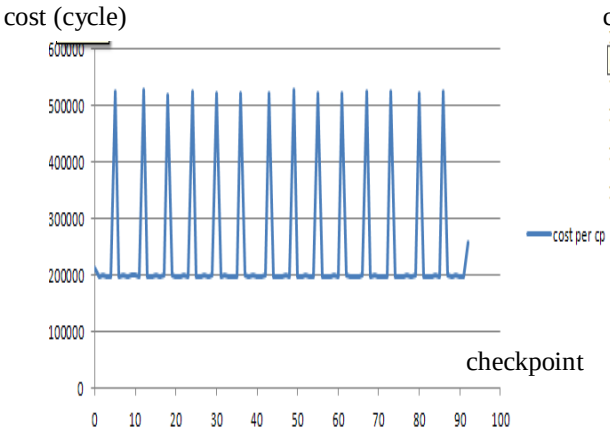


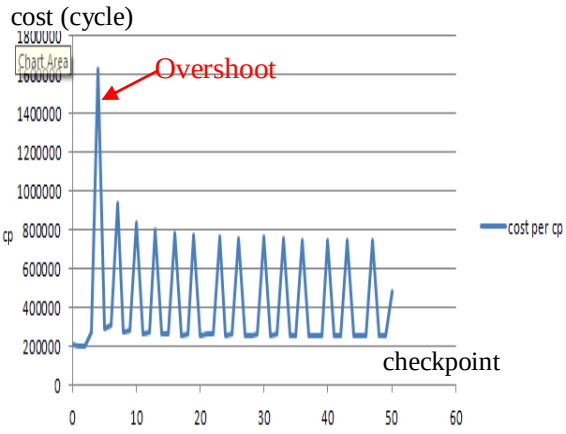
Figure 4-23 the distribution of cost for second adaptive check-pointing technique

From Figure 4-20, 4-21, 4-23, we note that the total cost of first adaptive check-pointing technique and the total cost of second adaptive check-pointing technique are less than the fixed check-pointing algorithm. So the adaptive check-point techniques can improve the fault tolerant effectiveness.

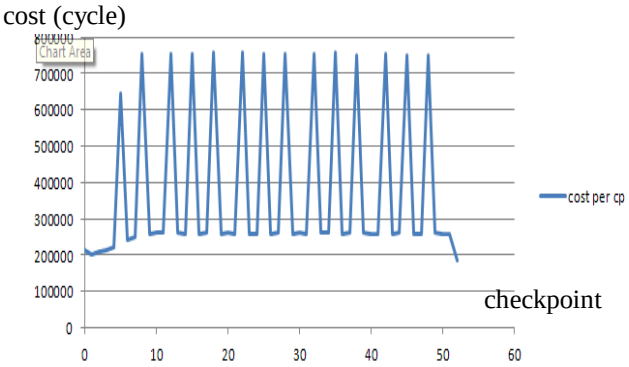
We compare the total cost of these adaptive check-pointing techniques. We note that the second adaptive checkpoint can provide a better effectiveness than the first adaptive check-pointing algorithm. To analyse the cost of these algorithms, we plot the cost at each checkpoint as shown in Figure 4-24.



a. Fixed check-pointing algorithm



b. First adaptive check-pointing algorithm



c. Second adaptive check-pointing algorithm

Figure 4-24 the cost for each checkpoint

From Figure 4-24a, we note that in the fixed check-point technique, the oscillation of the cost is stable and the maximum cost is 523896 which is smaller than the adaptive check-pointing techniques. However, the fixed check-pointing technique needs 92 checkpoints. In Figure 4-24 b and 4-24c, the oscillation and the number of checkpoints are similar to each other .However, there is an overshoot in Figure 4-24b and the overshoot is at checkpoint 4. To find the reason for the overshoot, we use Table4-20 as shown below.

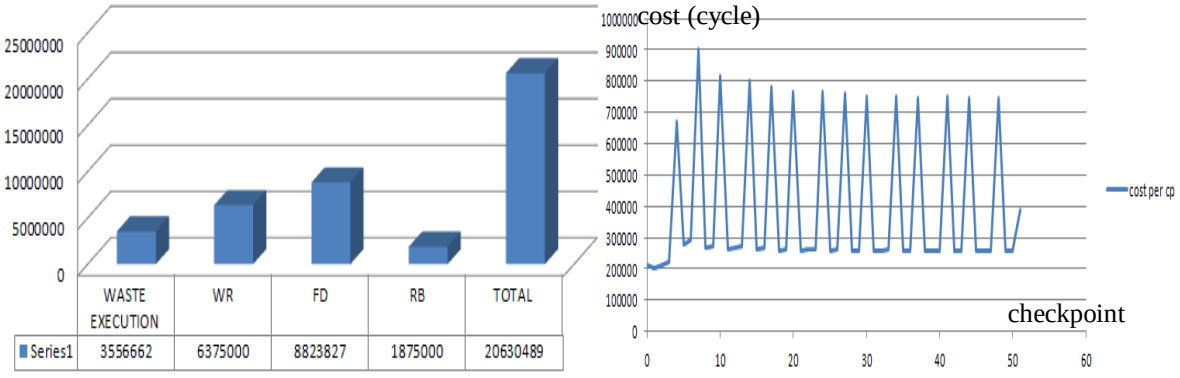
Total errors	Checkpoint	Checkpoint interval	Finished cycles	Cost per checkpoint
0	0	125000	152333	213353
0	1	125000	275328	196337
0	2	125000	402861	198969
0	3	250000	653511	270377
1	4	296955	951962	1619804
1	5	279554	1234352	288786
1	6	307890	1544352	304800

Table 4-24

In Table4-24, the checkpoint interval is the interval between the current checkpoint and next checkpoint. For example when the checkpoint is 0, the checkpoint interval 125000 is the interval between 0 and 1.The cost for checkpoint 0 is the cost during the interval between checkpoint 0 and checkpoint 1.In Table4-20, we note that the cost between checkpoint 4 and

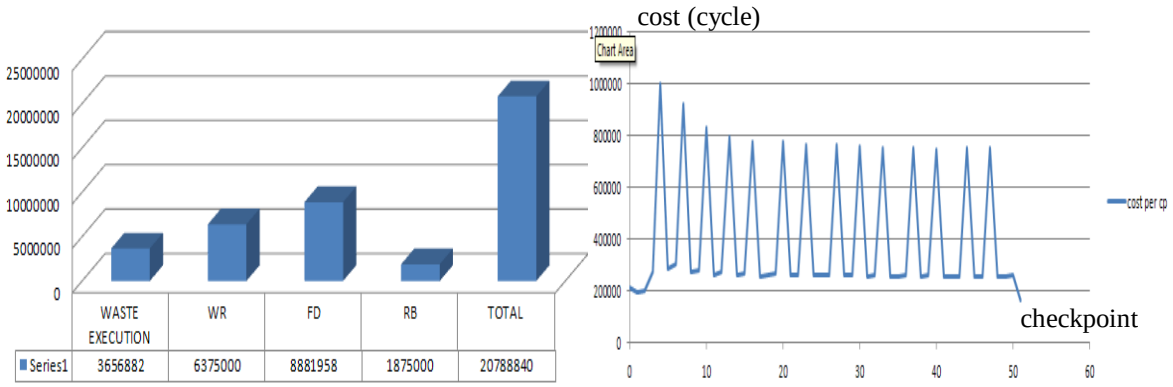
checkpoint 5 is very high. To find the reason causing such a high cost, we investigate checkpoint 3. From checkpoint 3 to checkpoint 4 there is no error detected, the checkpoint interval is 250000 and the finished cycles are 653511. In the first adaptive check-pointing technique, we first compare the finished cycles with the MTTF, in the simulation the $MTTF = \frac{1}{P_e * M} = 250000$ is less than the finished cycles so the system will increase the checkpoint interval based on the ratio of MTTF and finished cycles. The new checkpoint interval will be 25000*653511/250000=653511 which is too long. From checkpoint 4 to checkpoint 5 there is an error detected so the 653511 instructions are wasted. The sum of waste execution, fault detection and checkpoint cost cause the overshoot. To solve this overshoot, we change the method to increase the checkpoint interval when the system is in case 3. There are two methods are introduced. First, we can increase the checkpoint interval 10% when the system is in this case. Second, we can increase the checkpoint interval using the initial checkpoint interval * (Finished cycles/NMTTF).

We test these two methods separately and the results are shown in Figure 4-20.



c. total cost using first method

a. Cost for each checkpoint



c. total cost using second method

d. Cost for each checkpoint

Figure 4-20 the total cost and each checkpoint cost for these two methods

From Figure 4-20, we note that both of these methods will reduce the total cost. However, the first method is not very reasonable, for a different number of errors or different system the increase percentage increase may be different. Therefore, we adjust our first adaptive check-pointing technique with a second method. The data flow diagram is shown in Figure 4-21.

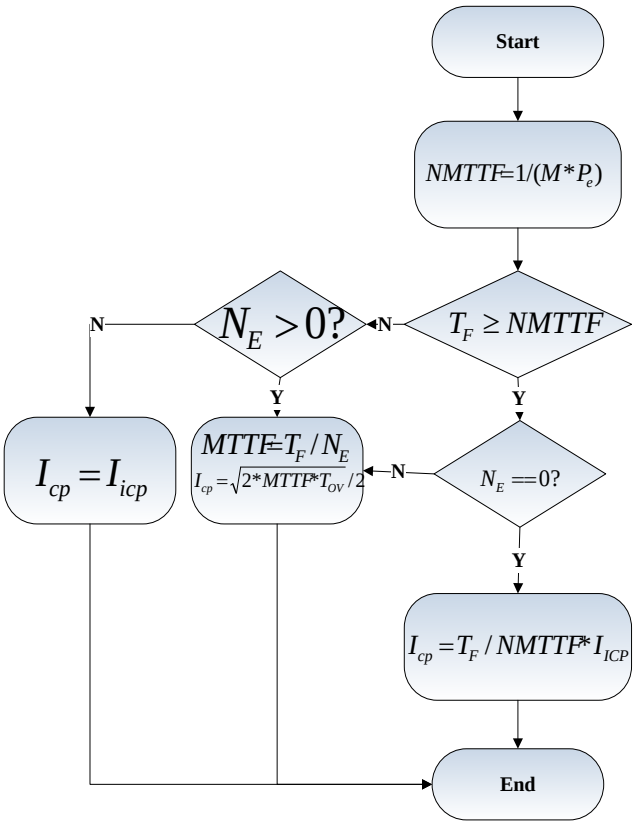


Figure 4-21 the data flow diagram of improved first adaptive check-pointing technique

5.1Summary

These two adaptive check-pointing techniques are better than the fixed check-pointing technique. For the first adaptive check-pointing technique, we can use the equation $T_c = \frac{\sqrt{2T_sT_f}}{2}$ to set the checkpoint interval .For the second adaptive check-pointing technique, we set the initial checkpoint interval and according to the equation $y = 5.1e-012x^2 - 2.5e-006x + 0.3$ to calculate the optimal x. We adjust the first adaptive check-pointing algorithm to make it more effective. The effectiveness of these two adaptive check-pointing algorithms is close to each other.

6. Conclusion

In this report, we propose two adaptive check-pointing techniques and use the software simple-scalar to simulate the check-pointing technique. In the simulations, the fixed check-pointing technique and adaptive check-pointing techniques are tested and compared with each other. There are 5 main sections in this report.

In the first section is the introduction of the project. We introduce the objectives and aim of the project in this section.

In the second section, we introduce the background information. We choose the flag based error checking, the software simple-scalar, the type of check-pointing technique and the fault detection method.

In the third section, we explain the file conversion and simulate the error injection and fault detection. The method of error analysis and the simulation algorithm is also introduced in this section.

In the fourth section, we test the fixed check-pointing technique and propose two adaptive check-pointing techniques. The first adaptive check-pointing technique uses the equation to adjust the checkpoint interval to be the optimal checkpoint interval and it is better to be used in long test application. The second adaptive check-pointing technique is based on the MTTF and we get the relationship between the x and the MTTF. The second adaptive check-pointing can be used when we know the relationship between the initial checkpoint interval and the x. These two adaptive check-pointing techniques are better than the fixed check-pointing technique.

In the fifth section, we compare these two adaptive check-pointing techniques and improve the first adaptive check-pointing technique. The effectiveness of these two adaptive check-pointing techniques is close to each other. We also improve the first adaptive check-pointing technique.

7. Evaluation

The first objective of this project is to understand the fault tolerant system and the mechanism of check-pointing. We finish this objective by introducing the reliability and fault tolerant system and the SWIFT. The mechanism of check-pointing technique is also investigated and analysed. We choose the flag based error checking for our project.

The second objective is to design the method to simulate the check-pointing technique. We compare some software and choose the simple-scalar tool set to do check-pointing and re-execution. The fault detection and the error injection are also simulated with different methods. We combine all these sections and design the simulation algorithm as shown in Figure 3-6.

The third objective is to develop and test the fixed checkpoint algorithm. The fixed check-pointing technique uses the fixed checkpoint interval and we test the fixed check-point algorithm. There are three theories we get from the fixed check-pointing technique:

Theory 1: The new equation to set the checkpoint interval for 4M memory size

$$T_c = \frac{\sqrt{2T_s T_f}}{2}$$

Theory 2: With the same number of errors, the short checkpoint interval will decrease the waste execution and increase the WR.

Theory 3: With the same number of errors, the long checkpoint interval will increase the waste execution and decrease the WR.

The fifth and sixth objectives are to develop and test the adaptive check-pointing technique. We use the step check-pointing technique and set the checkpoint interval every time we have finish error detection. We propose two adaptive check-pointing techniques. The first adaptive check-pointing technique is based on the equation. The second adaptive check-pointing technique is based on the MTTF and a variable x. Both of them are better than fixed check-pointing technique when the actual failure rate is less than the expected failure rate. Finally, we conclude 6 theories:

1. The first adaptive check-pointing technique is appropriate for long test application.

2. When variable x is large, the stable checkpoint interval is long. On the other side, the stable checkpoint interval is short when the x is small.

3. When x is large the stable checkpoint interval will be reached at a small number of checkpoints. When x is small, the stable checkpoint interval will be reached at a big number of checkpoints.

4. When the initial checkpoint interval is long, the best x should be small. Otherwise, when the initial checkpoint interval is short the best x should be big.

5. When the initial checkpoint interval is short, the stable checkpoint interval will be reached at a big number of checkpoints. When the initial checkpoint interval is long, the stable checkpoint interval will be reached at a small number of checkpoints.

6. If the initial checkpoint interval is close to the optimal stable checkpoint interval the total cost is small.

The seventh objective is to compare these two adaptive check-pointing techniques and improve the adaptive check-pointing techniques. We compare these two adaptive check-pointing techniques and improve the first adaptive check-pointing technique. The effectiveness of these two adaptive check-pointing techniques is close to each other. The choice of these two adaptive check-pointing techniques is based on the equation of optimal checkpoint interval and equation of the relationship between the checkpoint interval and x.

8. Future work

The check-pointing technique is very popular in electronic system. In this project, we test our adaptive check-pointing in a simple system using the roll back recovery and simulate the fault detection and error injection. The future work can move in three directions.

The first direction is to test the adaptive check-pointing technique in a distribution system. In the distribution system, the check-pointing and recovery are more complex. We need to consider the consistent and the mechanism to identify the recovery line.

The second direction is to use the roll forward recovery technique instead of the roll back recovery. The roll forward recovery technique will reduce the waste execution because the system will rerun from the point that the error is detected. So the adaptive check-pointing technique should be adjusted for roll forward recovery technique.

The third direction is to assume there are more errors than we expected. In this project, we assume there are fewer errors than we expected. This is because in reality the expected failure rate is the worst case. However, sometimes the actual failure rate is worse than the expected failure rate. So we should design the adaptive check-pointing technique when the system is in this case.

9. Bibliography

[1] I.Koren and C.M.Krishna, 2007. *Fault-Tolerant Systems.* United Kingdom:Denise Penrose,147-223.

[2]John W.Young, 1974. *A First Order Approximation to the Optimum Checkpoint Interval*. Communication of the ACM Computing Machinery Machinery,Inc.

[3] Avi Ziv and Jehoshua Bruck. *An On-Line Algorithm for Checkpoint Placement*. IEEE Transactions on Computers Volume 46 Issue 9, September 1997 Page 976-985.

[4] T.Saridakis 2003.*Design Patterns for checkpoint-Based Rollback Recovery*. NOKIA Research Center PO Box 407, FIN-00045, Finland

[5] ZHANG zhan,ZUO De-cheng, CI Yi-WEI, yang Xiao-zong 2008.*The checkpoint interval optimization of kernel-level rollback recovery based on the embedded mobile computing System*. School of Computer Science and Technology, Harbin Institute of Technology, China.

[6]ARM Workbench 2012

http://www.arm.com/products/tools/software-tools/rvds/arm-workbench-ide.php on line

[Accessed on 20/04/2012]

[7]Freescale 2004-2012

http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME

[Accessed on 20/04/2012]

[8] Dough Burger Todd M. Austin 1997, *The Simple-scalar Tool Set*. Simple Scalar LLC

[9] Nathaniel H.Rollins and Michael J. Wirthlin 2007. *Software Fault-Tolerant Techniques for Softscor Processors in Commercial SRAM-based FPGAs*. Department of Electrical and Computer Engineering Brigham Young University.

[10] George.A.R,Chang.J,Neil.V, Rangan.R and David.I.A. 2005. *SWIFT: Software Implemented Fault Tolerance* Department of Electrical Engineering and Computer science Princeton University.

[11] Prusty.S 2009 *Checkpointing and Rollback Recovery in Distributed Systems* Department of Computer Science and Engineering Indian Institute of Technology, Guwahati

[12] Bhargave.B and Lian.S.R 1988 *Independent Checkpointing and Concurrent Rollback for Recovery in Distributed system*. Computer Science Department Purdue University

[13] Yi-Min Wang 1997.*Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints*. IEEE Computer Society.

[14]GNU Operating System

http://www.gnu.org/software/gsl/

[Accessed on 1-0/05/2012]

10. Appendix

Appendix 1:Fault detection code

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
enum _booltype{
        false,
        true=1
};

struct _regbool{
        enum _booltype reg;
        enum _booltype reg_copy;
        enum _booltype fault_detected;
};


struct _regchar{
        char reg;
        char reg_copy;
        enum _booltype fault_detected;
};

struct _regchar1{
        char reg[10];
        char reg_copy[10];
        enum _booltype fault_detected;
};

struct _reguchar{
        char reg;
        char reg_copy;
        enum _booltype fault_detected;
};

struct _regint{
        int reg;
        int reg_copy;
        enum _booltype fault_detected;
};

struct _reguint{
        unsigned int reg;
        unsigned int reg_copy;
        enum _booltype fault_detected;
};


struct _regshort{
        short reg;
        short reg_copy;
```

```
	enum _booltype fault_detected;
};


struct _regushort{
	unsigned short reg;
	unsigned short reg_copy;
	enum _booltype fault_detected;
};


struct _reglong{
	long reg;
	long reg_copy;
	enum _booltype fault_detected;
};


struct _regulong{
	unsigned long reg;
	unsigned long reg_copy;
	enum _booltype fault_detected;
};


struct _reglonglong{
	long long reg;
	long long reg_copy;
	enum _booltype fault_detected;
};


struct _regulonglong{
	unsigned long long reg;
	unsigned long long reg_copy;
	enum _booltype fault_detected;
};


struct _regfloat{
	float reg;
	float reg_copy;
	enum _booltype fault_detected;
};

struct _regdouble{
	double reg;
	double reg_copy;
	enum _booltype fault_detected;
};

struct _reglongdouble{
	long double reg;
	long double reg_copy;
	enum _booltype fault_detected;
};
```

```
struct _regenum{
	int reg;
	int reg_copy;
	enum _booltype fault_detected;
};


struct _regptr{
	short reg;
	short reg_copy;
	enum _booltype fault_detected;
};

typedef struct _regbool RegBool;
typedef struct _regchar RegChar;
typedef struct _regchar1 RegChar1;
typedef struct _regchar RegUChar;
typedef struct _regint RegInt;
typedef struct _reguint RegUInt;
typedef struct _regshort RegShort;
typedef struct _regushort RegUShort;
typedef struct _reglong RegLong;
typedef struct _regulong RegULong;
typedef struct _reglonglong RegLongLong;
typedef struct _regulonglong RegULongLong;
typedef struct _regfloat RegFloat;
typedef struct _regdouble RegDouble;
typedef struct _reglongdouble RegLongDouble;
typedef struct _regenum RegEnum;
typedef struct _regptr RegPtr;


#define Initialize(x, __value) ({x.fault_detected = false; x.reg = x.reg_copy = __value;})
#define DEBUG(x) printf("fault_detected: %s::line:%d\n",(x.fault_detected == true ? "true" : "false"),
__LINE__)



//assign string
#define AssignString(op_one,op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false; strcpy(op_one.reg,op_two);strcpy(op_one.reg_copy,op_two);})

//assign value
#define AssignValue(op_one, op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false; op_one.reg = op_two; op_one.reg_copy = op_two;DEBUG(op_one);})

//assign variable
#define AssignVariable(op_one, op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ?
true : false; AssignValue(op_one, (typeof(op_one.reg)) op_two.reg);DEBUG(op_one);})

//increment
#define Increment(op_one) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true : false;
op_one.reg = op_one.reg+1; op_one.reg_copy = op_one.reg_copy+1;DEBUG(op_one);})

//atof
```

```
#define ATOF(op_one) ({op_one.fault_detected = (strcmp(op_one.reg,op_one.reg_copy)==0) ? true : false;
\
RegDouble op_three2 ; op_three2.reg=op_three2.reg_copy=atof(op_one.reg);DEBUG(op_one);
op_three2;})

//atoi
#define ATOI(op_one) ({op_one.fault_detected = (strcmp(op_one.reg,op_one.reg_copy)==0) ? true : false;
\
RegDouble op_three2 ; op_three2.reg=op_three2.reg_copy=(double)atoi(op_one.reg);DEBUG(op_one);
op_three2;})

//POWER
#define POWER(op_one,op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false ; \
int op_three2 ; op_three2=pow(op_one.reg,op_two); DEBUG(op_one);op_three2;})

//Mulvalue
#define MulValue(op_one, op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false; \
typeof(op_one) op_three2  ; op_three2.reg=op_three2.reg_copy = op_one.reg*op_two;op_three2;})

//exp
#define EXP(op_one) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true : false; \
double op_three2 ; op_three2= exp(op_one.reg);DEBUG(op_one);op_three2;})

//Divide value
#define DivValue(op_one, op_two) ({op_one.fault_detected = (op_one.reg != op_one.reg_copy) ? true :
false; \
typeof(op_one) op_three2 = op_one; op_three2.reg = op_three2.reg_copy = op_one.reg /
op_two;op_three2;})

//divide variable
#define DivVariable(op_one, op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false;\
typeof(op_one) op_three2 = op_one; op_three2.reg = op_three2.reg_copy=DivValue(op_one,
(typeof(op_one.reg)) op_two.reg);op_three2;})

//LOG
#define LOG(op_one) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true : false; \
typeof(op_one) op_three2 = op_one; op_three2.reg = op_three2.reg_copy =
log(op_one.reg);DEBUG(op_one);op_three2;})

//ADD VALUE
#define AddValue(op_one, op_two) ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ? true :
false; op_one.reg = op_one.reg+op_two; op_one.reg_copy =
op_one.reg_copy+op_two;DEBUG(op_one);})
//ADD Variable
#define AddVariable(op_one, op_two); ({op_one.fault_detected = (op_one.reg == op_one.reg_copy) ?
true : false; AddValue(op_one, (typeof(op_one.reg)) op_two.reg);DEBUG(op_one);})



#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "V2_test .h"
main(void)
```

```
{
//  printf("ok\n");
  RegChar1 str;
  char co[]="123.456";
  RegDouble x,h,li,lr,a,lrr,t;
  RegInt i;
  FILE *file;
  file=fopen("go.txt","w");
  Initialize(x,0);
  Initialize(h,0);
  Initialize(li,0);
  Initialize(lr,0);
  Initialize(a,0);
  Initialize(lrr,0);
  Initialize(t,0);
  Initialize(i,0);
  AssignString(str,co);
 // printf("ok\n");
  for(AssignValue(i,0);i.reg<1;Increment(i)){
  fprintf(file,"pow(12.0, 2.0) == %f\n", pow(12.0, 2.0));
  fprintf(file,"pow(10.0, 3.0) == %f\n", pow(10.0, 3.0));
  fprintf(file,"pow(10.0, -3.0) == %f\n", pow(10.0, -3.0));

  fprintf(file,"str: %s\n", str);
 // printf("ok\n");
  AssignVariable(x,ATOI(str));
  fprintf(file,"x: %f\n", x.reg);

  fprintf(file,"str: %s\n", str);
  AssignVariable(x,ATOF(str));
  fprintf(file,"x: %f\n", x.reg);

  fprintf(file,"str: %s\n", str);
  AssignVariable(x,ATOF(str));
  fprintf(file,"x: %f\n", x.reg);

  fprintf (file,"%g %f %d %g\n", x.reg, x.reg, (int)x.reg, pow (10.0, 3.0));

  AssignValue(x,sinh(2.0));

  fprintf(file,"sinh(2.0) = %g\n", x);

  AssignValue(x,sinh(3.0));

  fprintf(file,"sinh(3.0) = %g\n", x);

  AssignValue(h,hypot(2.0,3.0));
//  printf(" h.reg%lf h.copy:%lf \n",h.reg,h.reg_copy);
  fprintf(file,"h=%g\n", h);

  AssignValue(a,atan2(3.0,2.0));

  fprintf(file,"atan2(3,2) = %g\n", a);

  AssignValue(lr,POWER(h,4.0));
//  printf("lr.reg:%lf lr.copy:%lf h.reg%lf h.copy:%lf \n",lr.reg,lr.reg_copy,h.reg,h.reg_copy);
```

```
  fprintf(file,"pow(%g,4.0) = %g\n", h.reg, lr.reg);

  AssignVariable(lrr,lr);

  AssignVariable(li,MulValue(a,4.0));
  t=MulValue(a,5.0);
  AssignVariable(lr,DivValue(lr,EXP(t)));

  fprintf(file,"%g / exp(%g * 5) = %g\n", lrr, a, lr);

  lrr = li;
  t=LOG(h);
  AddVariable(li,MulValue(t,5.0));

  fprintf(file,"%g + 5*log(%g) = %g\n", lrr, h, li);

  fprintf(file,"cos(%g) = %g,  sin(%g) = %g\n", li.reg, cos(li.reg), li.reg, sin(li.reg));

#if 0
  AssignValue(x,drem(10.3435,6.2831852));

  fprintf(file,"drem(10.3435,6.2831852) = %g\n", x);

  AssignValue(x,drem(-10.3435,6.2831852));

  fprintf(file,"drem(-10.3435,6.2831852) = %g\n", x);

  AssignValue(x,drem(-10.3435,-6.2831852));

  fprintf(file,"drem(-10.3435,-6.2831852) = %g\n", x);

  AssignValue(x,drem(10.3435,-6.2831852));

  fprintf(file,"drem(10.3435,-6.2831852) = %g\n", x);
#endif

  fprintf(file,"x%8.6gx\n", .5);
  fprintf(file,"x%-8.6gx\n", .5);
  fprintf(file,"x%6.6gx\n", .5);

  {
    double x = atof ("-1e-17-");
    fprintf (file,"%g %c= %g %s!\n",
                x,
                x == -1e-17 ? '=' : '!',
                -1e-17,
                fabs(x - -1e-17) < 0.000000001 ? "Worked" : "Failed");
  }
}
            printf("ok\n");
            fclose(file);
  return 0;
```

Appendix2: The simulation code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <math.h>
#define FD_ratio 0.58;
#define Processor 4;//4 bytes 32 bit processor
float error_probability=0.0000000000005;// set the error probability
int Memory_size=6000000;//The memory size can be 4M,6M and 8M
int number,n,CP_time,Left,T_cycles,E_cycles,N_cycles,Extra_cycles,flag,sum,WR_time,RB_time;
int
Total_error,P_sum,Ass_error58s,NMTTF,Total_instruction,Errors,MTTF,Ini_CP,A_cycles,Error_number,
Location,T_WR,T_FD,T_RB;
int faults[200];
float Failure_rate;

/* This function is used to simulate the error , when the randodm number less than
error_probability*100 there is an error*/
int detect_error(){
        int E,i,j;
        float point;
        E=0;
        for(j=0;j<Error_number;j++){
                if(sum <faults[j]&& faults[j] <=(sum+E_cycles)){
                        E++;
                        Location=faults[j];
                        faults[j]=-1;
                }
        }

        Extra_cycles=Extra_cycles+E_cycles*FD_ratio;
        T_FD=T_FD+E_cycles*FD_ratio;
        T_cycles=T_cycles+E_cycles*FD_ratio;
        return E;
}

/*This function is used to calculate cycles between each two checkpoints because Simplescalar can not
count
the cycles when I start simulation from a checkpoint*/
int Calculate_Cycles(){
   int C_number,jump1;
   char *j,*name;
   char command1[50];
   char out_file1[100];
   FILE *file1;
   jump1=0;
   sprintf(command1,"./sim-outorder -redir:sim Step2/various/Detail.txt -max:inst %d
F00.eio",Total_instruction);//run the program Total_number instruction from start
   system(command1);
   sprintf(command1,"Step2/various/Detail.txt",number);
   file1=fopen(command1,"r");
   while((fgets(out_file1,50,file1))!=NULL&&jump1!=1){
                name=strtok(out_file1," ");
                if(strcmp(name,"sim_cycle")==0){
                        j=strtok(NULL," #");
                        N_cycles=atoi(j);
                        C_number=N_cycles-sum;//calculate the cycles of the interval
                        jump1=1;
                }
   }
   fclose(file1);
```

```
   return C_number;
}
int main(int argc, char *argv[]){
        int error_found,i,jump,z,divide,p,L;
        long double D;
        FILE *file;
        FILE *csv_file;
        FILE *file1;
        char *name,*number_ins;
        char out_file[100], command[100];
        RB_time=WR_time= Memory_size / 32;//the Roll back time and write time
        csv_file = fopen("V4/NFixed_4_13_4M_.csv", "w");//output file
        if(csv_file == NULL){
          printf("Error Opening file\n");
          exit(1);
        }
        L=T_RB=T_FD=T_WR=sum=Extra_cycles=Errors=jump=flag=0;
        number=0;
        srand(time(NULL));
        system("rm -rf *.chkpt");
        system("rm -rf *.eio");
        /*get the number of instructions*/
        system("./sim-eio -redir:sim Step2/various/number.txt -trace F00.eio tests/bin.little/test-math");
        file=fopen("Step2/various/number.txt","r");
        if(file==NULL){
                printf("Failed to open the filename.");
                return 1;
        }
        while((fgets(out_file,50,file))!=NULL&&jump!=1){
                name=strtok(out_file," ");

                if(strcmp(name,"sim_num_insn")==0){
                        number_ins=strtok(NULL," #");
                        i=atoi(number_ins);// i is the number of  total instruction
                        jump=1;
                }
        }
        fclose(file);
        jump=0;
        sprintf(command,"./sim-outorder -redir:sim Step2/various/Detail1.txt -max:inst %d
F00.eio",i);//run the program Total_number instruction from start
        system(command);
        file1=fopen("Step2/various/Detail1.txt","r");
        if(file==NULL){
                printf("Failed to open the filename.");
                return 1;
        }
        while((fgets(out_file,50,file))!=NULL&&jump!=1){
                name=strtok(out_file," ");
                if(strcmp(name,"sim_cycle")==0){
                        number_ins=strtok(NULL," #");
                        A_cycles=atoi(number_ins);//A_cycles is the number of  total cycles
                        jump=1;
                }
        }
        fclose(file1);
        printf("The total cycles is %d\n",A_cycles);
```

```
Error_number=error_probability*Memory_size*A_cycles;
printf("There are %d errors\n",Error_number);
CP_time=sqrt(2*WR_time/error_probability/Memory_size);
CP_time=CP_time*0.58;
Error_number=Error_number/2;
printf("There are %d errors\n",Error_number);
p=A_cycles%Error_number;
z=(A_cycles-p)/Error_number;
//inject the error every fixed number of cycles to make the uniform distribution
for(p=0;p<Error_number;p++){
        L=L+z;
        faults[p]=L;
}
fprintf(csv_file, "Total_num_errors, CP, exec_per_cp, original_total_exec, perf_cost\n");
system("./sim-eio -dump 0.chkpt 0: F00.eio");// set the initial checkpint
while(Total_instruction<i){
//run the program the cp_time number instructions
        Total_instruction=Total_instruction+CP_time;//increase the finished total instructions
        E_cycles=Calculate_Cycles();
        error_found = detect_error();
        Errors=Errors+error_found;
        if(Total_instruction!=i&&error_found==0){
                sprintf(command,"./sim-eio -dump checkpoint.chkpt %d: F00.eio",
Total_instruction);// set the first checkpoint
                system(command);
                Extra_cycles=Extra_cycles+WR_time;
                T_cycles=T_cycles+WR_time;
                T_WR=T_WR+WR_time;
        }
        if(error_found==0){
           sum=sum+E_cycles;
           T_cycles=T_cycles+E_cycles;
           number++;
        }else{
           Extra_cycles=Extra_cycles+RB_time+E_cycles;
           T_RB=T_RB+E_cycles;
           T_cycles=T_cycles+RB_time+E_cycles;
           Total_instruction=Total_instruction-CP_time;
        }
        if(error_found==0)
       fprintf(csv_file, "%d, %d, %d, %d, %d\n",Errors,(number-1),CP_time,sum, Extra_cycles);

        printf("error found %d\n ",error_found);
        printf("CP_time %d\n",CP_time);
        printf("Total instruction %d\n",Total_instruction);
        printf("The total cycles is %d\n",A_cycles);
        printf("There are %d errors\n",Error_number);
        if((Total_instruction+CP_time)>=i)
             CP_time=i-Total_instruction;

  }
printf("The number of instruction is %d\n",i);
printf("The T_RB is %d\n",T_RB);
printf("the T_WR is %d\n",T_WR);
printf("the T_FD is %d\n",T_FD);
printf("There are %d errors\n",Error_number);
printf("Extra performance cost for fault tolerance:%d\n",Extra_cycles);
```

```
        printf("The total cycles is %d\n",T_cycles);
        printf("The application cycles is %d \n",sum);
        fclose(csv_file);
        return 0;
}
```

Appendix3: The first adaptive check-pointing and second adaptive check-pointing code

3.1 First adaptive check-pointing technique

```
int Set_Interval(int errors){// The first check-pointing technique to set the checkpoint interval
        int Value,m;
        if((sum+E_cycles)>=NMTTF){
                if(Errors==0){
                        m=(sum+E_cycles)/NMTTF;
                        CP_time=Ini_CP*m;
                }else{
                        MTTF=(sum+E_cycles)/Errors;
                        CP_time=(long double)sqrt(2*(long double)WR_time*(long double)MTTF);
                        CP_time=CP_time*0.58;
                }
        }else{
                if(Errors>0){
                        MTTF=(sum+E_cycles)/Errors;
                        CP_time=(long double)sqrt(2*(long double)WR_time*(long double)MTTF);
                        CP_time=CP_time*0.58;
                }else
                        CP_time=Ini_CP;
        }
        return CP_time;
}
```

Appendix4: Second adaptive check-pointing technique

```
int Set_Interval(int errors){ //set the checkpoint interval useing the seocnd adaptive checkpoint technique
        int Value,m,N_rate;
        float probability;
        if(Errors==0){
                if((sum+E_cycles)>=NMTTF){
                                CP_time=CP_time*(1+x);
                }else
                                CP_time=Ini_CP;
        }else {
                MTTF=(sum+E_cycles)/Errors;
                if(MTTF>NMTTF){
                        if(MTTF>M_rate){
                                M_rate=MTTF;
                                CP_time=CP_time*(1+x);
                        }
                }else
                        CP_time=CP_time*(1-x);
        }
        if(CP_time<Ini_CP*0.8)
                CP_time=Ini_CP;
        return CP_time;
}
```