A FLEXIBLE SIMULATION FRAMEWORK FOR THE STUDY OF DEADLOCK RESOLUTION ALGORITHMS IN MULTICORE SYSTEMS

by

Dhruv Desai

B.E., Gujarat Technological University, 2012

THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

February 2016

Abstract

Deadlock is a common phenomenon in software applications, yet it is ignored by most operating systems. Although the occurrence of a deadlocks in systems is not frequent, in some cases, the effects are drastic when deadlock occurs. The ongoing trend in processor technology indicates that future systems will have hundreds and thousands of cores. Due to this imminent trend in hardware development, the problem of deadlock has gained renewed attention in research. Deadlock handling techniques that are developed for earlier processors and distributed systems might not work well with multicore systems, due to their architectural differences. Hence, to maximize the utility of multicore systems, new programs have to be carefully designed and tested before they can be adopted for practical use. Many approaches have been developed to handle deadlock in multicore systems, but very little attention has been paid to comparing the performance of those approaches with respect to different performance parameters.

To fulfil the above mentioned shortfalls, we need a flexible simulation testbed to study deadlock handling algorithms and to observe their performance differences in multicore systems. The development of such a framework is the main goal of this thesis. In the framework, we implemented a general a scenario, scenario for the Dining Philosopher's problem and scenario for the Banker's algorithm. In addition to these scenarios, we demonstrate the flexibility, soundness, and use of the proposed framework by simulating two different deadlock handling strategies - deadlock avoidance (the Banker's algorithm) and deadlock detection (Dreadlocks). The deadlock detection is followed by deadlock recovery to resolve the detected deadlock. We also present result analysis for the different set of experiments performed on the implemented strategies. The proposed simulation testbed to study deadlocks in multicore systems is developed using Java.

Contents

List of Figures

List of Tables

Acknowledgements

Although the title page of this thesis shows my name, this thesis would not have been possible without the contributions of many others. I cannot possibly name here all of them, but I would like to forward my special thanks to a few.

My deepest gratitude is to my supervisor Dr. Alex Aravind for his continuous guidance, support, constant encouragement and endless patience. Alex has been a mentor not only academically, but also throughout my graduate experience at UNBC. I would like to thank my supervisory committee members Dr. Waqar Haque and Dr. Geroge Jones, for their time and valuable inputs that helped in shaping this thesis up. I am also thankful to Dr. Ajit Dayanandan for his suggestions during the initial stage. My thanks are also due to the external examiner Dr. Balbinder Deo and the chair of my defense for reading this thesis. I extend a special thanks to Dr. Andreas Hirt for giving me opportunities to assist him in his courses.

A special word of gratitude to all my colleagues over the past few years who have helped me to become a better researcher and a better person. The list would definitely include Dhawal, Denish, Giridhar, Darshik, Rafael, Rahim, Shanthini, Suresh, Behnish, Raman and Mehul. I also thank Brooke for proofreading the draft of my thesis. I specially thank our systems administrator Chris for always being there for technical support. A special thanks to Ms. Mahi Aravind for her support and great food on various occasions.

Most importantly, none of this would have been possible without the love and patience of my family. I will be forever thankful to my parents and my brother for their unwavering support and encouragement.

Dhruv Desai

Dedicated to my family

Chapter 1

Introduction

Deadlock was introduced and studied in the mid-1960s by Dijktra [11], originally referred to as "Deadly Embrace". Additional work was done in late 1960s and early 1970s by Coffman [9], Habermann [16], Holt [20] and others. In this chapter, we start by introducing the problem of deadlock, necessary conditions for deadlock to happen, and techniques to deal with deadlock. A discussion about changing trends in hardware, the rationale behind this thesis, and the contribution it lends to the knowledge will be discussed further on.

1.1 Overview

Deadlock is a situation in a resource allocation system in which two or more processes are in a simultaneous wait state, each one waiting for one of the others to release a resource before it can proceed [4]. In the above definition of deadlock, *processes* are the active entities in a system that share *resources* such as data items in a database, a data structure, a file etc. Figure 1.1 shows a system in a deadlocked state.

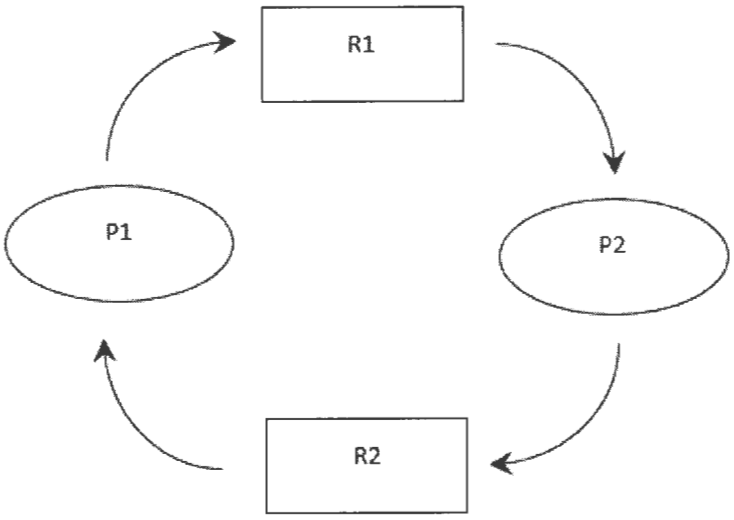

Figure 1.1: Deadlock Situation

In the figure above, ellipses represent processes while rectangles represent resources. An arrow going towards a resource from a process shows that a request is made by the process to acquire the resource, while an arrow going towards a process from a resource shows that the resource is held by the process. Moreover, a resource may have several instances and in that case a process can request one or more instances of a resource.

Presence of four conditions is necessary for deadlock to occur in any system. These conditions are known as Coffman's conditions as they are from a 1971 publication by Coffman et al. [9]. Coffman's conditions are Mutual exclusion, Hold and Wait, No preemption and Circular wait.

- *Mutual exclusion* means that only one process may use a resource at a time, in a given system.
- *Hold and wait* describes that a process may hold allocated resources while wait-

ing for the requested resources to be allocated.

- *No preemption* indicates that no resources can be forcefully taken back from a process holding it, unless a process releases the held resources upon completion. If the first three conditions hold, as a consequence, a system may end up in the fourth condition of circular wait.

- *Circular wait* is defined as, a closed chain of processes, such that each process holds at least one resource needed by the next process in the chain [31].

In Figure 1.1, process $P1$ has exclusive access to resource $R2$ and process $P2$ has exclusive access to resource $R1$ (*Mutual Exclusion*). To complete their executions, process $P1$ needs resource $R1$ and process $P2$ needs resource $R2$ (*Hold and wait*). Any process can not release the resources that they hold before finishing their executions (*No preemption*). Thus both processes wait for the resources that are held by other processes (*Circular wait*) and hence they stay in waiting state forever. In this situation, the system is deadlocked.

Knapp [22] and Singhal [30] discuss two types of deadlock: *resource deadlocks* and *communication deadlocks*. Resource deadlocks involve reusable resources while communication deadlocks involve consumable resources. A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use such as CPU, printer, disk etc. whereas a consumable resource is one that can be created and consumed such as messages or signals used between processes to communicate [31]. This thesis focus only on resource deadlock, referred to as deadlock further.

There are three general strategies to deal with deadlock (*i*) Deadlock prevention, (*ii*) Deadlock avoidance and (*iii*) Deadlock detection and recovery.

- **Deadlock prevention**: A deadlock prevention technique breaks at least one of the four Coffman's conditions [9]. Breaking one of the first three conditions is hard because it is difficult to determine which of the conditions may lead the system to deadlock. Moreover, satisfying these conditions is the fundamental requirement for most of the systems. Hence, most of the deadlock prevention techniques work by preventing circular waiting of processes.

- **Deadlock Avoidance**: If some specific information, such as all required resources, about processes in a system are known in advance, deadlock can be avoided by always keeping the system in a safe state. A system is considered to be in a *safe state* if all the processes can complete their execution without forming a deadlock [17]. All the systems are in a safe state at the beginning because the processes can always be executed sequentially. Deadlock can never occur in a sequential execution as there is no competition for resources. A safe state never ends in a deadlock. Also not all unsafe states result in a deadlock. Safe or unsafe states only determine the probability that a system might enter a deadlock. One of the limitations of this strategy is that knowing all the required resources in advance is unrealistic. Determining that information in advance is not possible for modern systems due to their dynamic behaviour. The Banker's algorithm is a popular method for deadlock avoidance which is explained in Chapter 3.

- **Deadlock Detection and Recovery**: Due to the limitation of prevention and avoidance algorithms as discussed above, detection and recovery techniques are more popular in practical systems. They allow processes to acquire resources whenever possible and a check can be done periodically to detect if deadlock has occurred in the system. Upon detection of deadlock, a recovery strategy is applied. Typically, a recovery strategy includes aborting all or some of the

deadlocked processes. These techniques are suitable in systems where deadlocks are not frequent, and recovery cost is not very expensive. We have implemented a deadlock detection technique known as *Dreadlocks* [23] in our system. This technique is explained in Chapter 3.

Once a deadlock is formed and detected in the system, the next step is deadlock recovery. *Deadlock recovery* involves the system taking corrective actions by selecting some processes to forcefully terminate and make their acquired resources available. The terminated processes can be re-initiated later. If the selection of victim processes is not done carefully, aborting can lead to livelocks. "*Livelock* is a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work" [31]. The selection criteria to choose one or more processes to terminate are discussed in Chapter 3.

Most of the modern operating systems- including Windows and the UNIX family- ignore deadlock [32]. In these systems, if the user observe that the system is slowing down or freezes due to some problem, probably deadlock, it is simply restarted. There are some reasons to ignore a deadlock: (*i*) If the occurrence of deadlock is very rare, and prevention, avoidance or detection and recovery overhead is very high, (*ii*) If restarting jobs does not incur significantly high costs, and (*iii*) If other deadlock handling techniques make the system slow, or are restrictive and complex, ignoring deadlock is preferred. Ignoring deadlock is chosen assuming that most users would prefer the occasional occurrence of deadlock rather than living with the limitations of deadlock handling strategies.

1.2 Trends in Hardware Development

There has been a significant change in computer systems hardware in the past couple of decades. Initially, most systems were standalone using single cores. Later, distributed systems were developed, followed recently by multicore systems becoming popular for both personal and commercial purposes. Hardware developments in turn demand changes in programming for effective use of multicore systems. So, multithreaded programs designed to frequently share resources is expected to become a norm for program development in the future.

1.2.1 Distributed Systems vs Multicore Systems

Distributed systems consist of geographically distant, separate, autonomous computers, connected through a network, communicating with each other by passing messages. They could often coordinate their activities to solve a common problem or to share the computing resources and storage devices of the systems. System or application processes executing on any network node can use both local and remote shared resources, simultaneously and exclusively. *A multicore processor* is an integrated circuit (IC) to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks.

Distributed and multicore systems share two similarities: concurrency and resource sharing. To achieve these, nodes in distributed systems and cores in multicore systems have to coordinate with each other. For the purposes of communication and coordi-

nation, message passing is used in distributed systems, whereas reading and writing through shared memory is used in multicore systems. Each node in distributed systems is an autonomous system while in multicore systems a core is not. Concurrency and resource sharing can lead to deadlock in both the systems.

1.2.2 Deadlock: Distributed Systems vs Multicore Systems

Two types of deadlock can occur in distributed systems, either resource deadlock or communication deadlock. Resource deadlock happens mainly because of circular dependencies of processes on each other for resources. Communication deadlock primarily occurs due to lost or delayed messages. Because of the architectural differences discussed above, resource deadlocks are more common than communication deadlocks in multicore systems.

Deadlocks are dealt with either in a centralized way or a distributed way in distributed systems. In centralized deadlock handling algorithms, a single node keeps list of all other nodes and has complete authority to allocate and preempt resources from all other nodes. In distributed deadlock handling algorithms all the nodes keep list of other nodes and update the list frequently, whereas in multicore systems, deadlocks are mostly dealt with in a centralized way because cores are not autonomous and are typically managed by a single operating system. Distributed deadlock handling algorithms are developed keeping communication overhead in mind. Multicore systems do not have to deal with the overhead of keeping the node address-lists and different topologies among the nodes. Moreover, to the best of our knowledge, no performance evaluation studies have been performed for multicore systems.

Multicore systems use locks to avoid data races in programs. Improper use of these

locks can create deadlock in the system. Improper lock acquisition can happen due to following reasons: (*i*) It becomes difficult to follow the lock order discipline that could avoid deadlock as most of the time software systems are written by multiple programmers. (*ii*) As stated before, adding new locks to fix race conditions introduces deadlock. (*iii*) Sometimes third-party software, such as plugins, are incorporated in software systems; such third-party software may not follow the locking order discipline followed by the software systems and this could result in deadlock.

Due to the uncertainty of deadlock and its occurrence in modern systems, researchers have come up with different approaches to deal with deadlock in multicore systems. The approaches are classified as follows: Language-level [5, 18], Static analysis [12, 14, 27], Dynamic analysis [1, 21, 29], Model checking [15] and Type and annotation based techniques [7].

Language-level approaches strive to make concurrency control easier by providing higher level constructs that do not allow uses that can cause errors. Even though languages alleviate some of the complexity of concurrency, the problem is not completely eliminated. *Static program analysis techniques* do not require any execution of the application as they directly work with source code. They examine all the possible deadlocks and often give no false negatives, but they report many false positives. *Dynamic program analysis techniques* find all potential deadlocks during the program execution. As they operate at runtime, they only visit feasible paths and have accurate views of the values of variables.

Model checking takes simplified descriptions of the code and uses techniques to explore vast state spaces efficiently as it systematically tests the code on all inputs. Because of this, this approach does not need to explore massive state spaces in order to find deadlocks. *Type and annotation based techniques* help to avoid deadlocks

during coding by allowing programmers to specify a partial order among locks. Also, the type checker statically ensures that the partial order among locks is maintained, and well-typed programs are deadlock-free. We discuss these approaches in detail in Chapter 2.

Parallel systems share more similarities with multicore systems than distributed systems. But due to their architectural and functional level differences, deadlock handling techniques for parallel systems cannot be directly applied to multicore systems. The differences between parallel and multicore system are as follows:

- Large tasks are divided into several, similar, smaller subtasks to carry out computations in parallel systems. Then, each of the smaller subtask is solved independently and at the same time. The results of the smaller subtasks are combined upon completion. In multicore systems, the tasks are often not related to each other. Also, they often do not work at the same time rather multicore systems use concept of interleaving.

- Parallel systems are basically designed for speed hence they have well defined bases such as switching context. They also share less of controllers and cache memory. Context switching depends on type of scheduler in multicore systems. Multicore systems also share more of controllers and cache than parallel systems.

Another type of hardware system, that is similar to multicore systems, is multiprocessor systems. There are multiple ICs in single processor of multicore systems whereas multiprocessor systems consist of more than one processors. Each of the processor in multiprocessor systems may contain one or more ICs in them. In terms of efficiency, multicore systems perform better on a single program because the cores can

execute multiple instructions at the same time but not multiple programs. When using multiple programs, multiprocessor system perform better than multicore systems. Due to the differences in their architectures, multicore system is more favourable system for ordinary users. Any extra support or configuration is not required for multicore systems and they cost less too. Multiprocessor systems are more favourable for special purpose uses. Often they require extra support or configuration based on the purpose, and they are expensive too.

1.3 Motivation

Deadlock is a very common phenomenon in software applications. Lu et al. showed that out of 105 randomly selected real world concurrency bugs that are collected from 4 large open-source applications: MySQL, Apache, Mozilla and OpenOffice - representing both server and client applications, 31 are deadlock bugs [26]. Also, a report produced by Oracle's bug database shows that roughly 6500 out of 198,000 reports, or 3%, contain the keyword 'deadlock' [27]. Although occurrence of deadlocks in systems is not frequent in some cases, the effect are drastic when deadlock occurs.

During the testing phase, deadlocks can be easily created and detected statically but detecting deadlocks dynamically is difficult because concurrent applications are non-deterministic. Deadlocks happen under certain intricate sequences of low-probability events, and producing with this subtle sequence of low-probability events is not guaranteed during execution of the application. Stress testing or random testing may not always show the existence of deadlocks in an application, but the probability of their occurrence increases greatly when an application is released to thousands of users. Therefore, deadlock detection tools are required to analyze and find real and

potential deadlocks in application.

From the discussion in the previous section, we observe that a lot of research has been conducted to develop techniques to handle deadlocks. But very little attention has been paid towards comparing the performance of those approaches to deal with deadlock. To the best of our knowledge, no research has been pursued to compare the performance of different algorithms in multicore systems. All of the performance comparisons that we could find were done on distributed systems, distributed database systems, flexible manufacturing systems and others. Moreover, only a few of the performance studies use simulation testbeds to analyze the performances. The ongoing trend in processor technology indicates that future systems will have hundreds and thousands of cores. To maximize their utility, new programs have to be carefully designed and tested using a large number of cores and a proper set of experiments before they can be adopted for practical use.

Thus the significant presence of deadlocks in large-scale, real-world applications, the uncertainty around their occurrence, their crucial effect and lack of research work towards performance study in multicore systems are the main motivating factors for this research. Awareness that the field of multicore systems is quite new and emerging compared to other well-established computer systems also provides rationale for this study.

1.4 Contributions

As discussed before, deadlock handling techniques that are developed for earlier processors and distributed systems will not work well with multicore systems, mainly

because of the difference in the architecture of these systems. Moreover, communication overhead is one of the most influencing factors for deadlock handling techniques that are used in distributed systems. As multicore systems use shared memory, the effect of communication overhead is rare. Hence, there is a need to explore more deadlock handling techniques that are efficient in multicore systems. However, there is also lack of availability of tools to study deadlock handling techniques with respect to different performance parameters in multicore systems.

This thesis helps to fulfil the above mentioned shortfalls in the field of computer science research. Studying performance of the implemented deadlock handling techniques and its effect in multicore systems is very important. Developing a flexible simulation testbed would be very useful to study deadlock handling algorithms and also to observe different performance parameters for those algorithms in multicore systems. Development of such a flexible framework is one of the primary contributions of this thesis. All the primary contributions of this thesis are listed as follows:

- A flexible simulation framework to incorporate different types of deadlock handling algorithms
- Three different scenarios implemented to simulate different type of deadlock handling techniques. The scenarios are: General scenario, scenario for the Dining Philosopher's problem, scenario for the Banker's algorithm
- Two different deadlock handling strategies are simulated in the framework with the above mentioned scenarios. The deadlock avoidance strategy is the Banker's algorithm and deadlock detection strategy is Dreadlocks. Deadlock detection is followed by deadlock recovery- termination of one of the victim processes to end the deadlock.

- Result analysis of experiments performed on implemented algorithms with respect to various performance metrics

1.5 Thesis Organization

The overview of the problem of deadlock, necessary conditions that are required for the formation of deadlock and different approaches to handle deadlocks were briefly explained in this chapter. In Chapter 2, a detailed literature review of the current methods to deal with the problem of deadlock in multicore system is presented. The review and the methods are categorized by different types. At the end of this chapter we present a comprehensive survey of all the performance evaluation studies done on deadlock handling techniques in different systems. In Chapter 3, we present the details about design and implementation of the simulation framework for deadlock in multicore systems; the primary contributions of this thesis are explained in this chapter. Later in this chapter, we explain different deadlock handling strategies that are implemented in our framework. We discuss various traces generated from the simulation run and the calculation of performance metrics from these traces in Chapter 4. Results and experimentation are described in Chapter 5. Lastly, we conclude in Chapter 6 with discussing possible future work that can be conducted to extend the research carried out in this thesis.

Chapter 2

Background and Related Work

Research for this thesis began with a paper that my supervisor gave me, about a deadlock detection algorithm, called "Dreadlocks: Efficient Deadlock Detection" [23], developed by Eric Koskinen and Maurice Herlihy. This algorithm to detect deadlocks in multithreaded programs gained our attention and we began to look for other approaches that have been developed to deal with deadlocks in multicore systems. We explain Dreadlocks in detail in Chapter 3. Here, we discuss the other interesting approaches in the literature.

2.1 Deadlock Handling Approaches in Multicore Systems

As briefly discussed in Chapter 1, many different approaches have been developed to handle deadlocks in multicore systems. The approaches are classified in five categories (i) Language-level approaches, (ii) Static Analysis, (iii) Dynamic Analysis, (iv) Model Checking and (v) Type and annotation based techniques. They are explained in detail as follows:

2.1.1 Language-level Approaches

Language level approaches make concurrency control easier by providing higher level constructs that do not allow error-prone uses. This can be achieved by statically binding shared variables to the locks in order to protect them. Two different language-level approaches are discussed here.

First, we discuss a framework implemented in Java which is presented by Bensalem et al. [5]. This framework confirms deadlock potentials, detected by runtime analysis of a single run of a multithreaded program. The multithreaded program under examination is implemented to produce lock and unlock events. A trace is generated when the implemented program is executed. The trace consists of the lock and unlock operations performed during the specific run. An observer is constructed using each cycle that can detect the occurrence of the corresponding real deadlock. Here, *an observer* is a data structure that characterizes all possible interleavings that lead to a deadlock state [5]. It also checks the possibility of the deadlock reoccurring during subsequent test runs. In this framework, a controller determines the optimal scheduling strategy that will maximize the probability for the corresponding real deadlock to occur, when composed with the program.

The methodology of the approach involves four phases: (*i*) By examining a single execution trace, deadlock potentials in multithreaded program and the system under test (SUT) are automatically detected, (*ii*) Automatic generation of an observer for each cycle in the resulting lock graph, (*iii*) Implementation of the system under test in order to confirm deadlock potentials and (*iv*) By performing multiple "controlled runs" searching "incorrect execution traces" [5].

Elaborating on these phases, the first step involves detecting deadlock potentials in the SUT. Then, the instrumentation module automatically implements the bytecode class files of the multithreaded program under test. This is done by adding new instructions that, when executed, generate the execution trace consisting of lock and unlock events. The observer module reads the event stream and performs the deadlock analysis. While the lock and unlock events are observed, the implemented program under observation is executed. Then a graph is constructed with the edges between locks suggesting lock orders. Any cycle in the graph signals a deadlock. Once the lock graph is generated, an observer for each cycle in the lock graph is generated automatically. The observer observes the SUT. In the last step, execution of SUT is controlled. This execution contains deadlock if it is "accepted" by the observer [5].

The second approach works for parallel programs that use message passing for communication. One of the common problems in such programs is the detection of deadlocks. Haque presented a deadlock detector, Message Passing Interface Deadlock Detector (MPIDD), to dynamically detect deadlocks in parallel programs that are written using C++ and Message Passing Interface (MPI) [18]. There are two main parts of the MPIDD system, the detector and the MPIDD wrappers.

The detector receives input from the client program about MPI commands being used. Then by using this information, the detector constructs a state and determines if there is a deadlock in the client programs. The MPI function call wrappers stimulate the client program's original calls and provide the information to the detector. MPI's profiling layer requires no significant modification of the user's code and incurs very little overhead when invoked. This property of the MPI's profiling layer is used by the detector. A wrapper containing information needed by the detector is created using the MPI's profiling layer. The actual function call is embedded inside this wrapper.

The new wrapped routines relay the information of what commands are being issued to the deadlock detector program.

The key advantages of this tool are its portability, very low overhead and the use of MPI's profiling layer that allows use of this tool without modification of the source code. By including the file 'MPIDD.h' in the original source code, the client can obtain all the functionality transparently [18]. Since MPI is a widely used library, it is a useful approach. Although the main limitation of this tool is that it is MPI specific, it can also be easily adoptable to other libraries.

Even though language level approach alleviates some of the complexity of concurrency, the problem is not completely eliminated. Also, to get any benefits from this approach, all the code has to be written in a particular language. These limitations prevent programmers from using other languages that might better suit their requirements.

2.1.2 Static Analysis

Static program analysis techniques do not require any execution of the application as they directly work with source code. Here, we discuss three different static program analysis techniques as follows:

The first technique is called RacerX [12]. It is a static tool that uses flow sensitive, interprocedural analysis to detect deadlocks. This technique checks information such as which locks protect which operations, which shared accesses are dangerous and which code contexts are multithreaded. On higher level, checking a system with RacerX involves five phases [12] (i) Retargeting RacerX to system-specific locking func-

tion, (*ii*) Obtaining a control flow graph from the checked system, (*iii*) Running the deadlock and race checkers over the obtained flow graph, (*iv*) Subsequent-processing and ranking the results, and (*v*) Examining the results. Out of all these phases, the first and the last phases are done by the user, RacerX does the middle three.

The second technique is an effective static deadlock detection algorithm for Java, presented by Naik et al. [27]. The key idea behind this is to show the complex property of deadlock freedom for a pair of threads/locks in terms of six conditions: reachable, aliasing, escaping, parallel, non-reentrant and non-guarded [27].

Effectively approximating these conditions requires precise call-graph and points-to information. This algorithm uses the combined call-graph and may-alias analysis for effective approximation. This combination of analysis is called k-object-sensitive analysis. It reliably approximates the first four conditions using well-known static analysis, a call-graph analysis, a may-alias analysis, a thread-escape analysis, and a may-happen analysis respectively. In order to soundly approximate the last two conditions, it requires a must-alias analysis which is much harder than a may-alias analysis. This technique addresses this difficulty using an unsound solution of using may-alias analysis to pose as a must-alias analysis. Hence, this algorithm failed to report some real deadlocks [27].

All of the deadlock detection tools helped to find deadlock in concurrent programs. The problem of detecting deadlock in libraries has not been investigated much. The third approach, developed by Williams et al. helps with that [34]. This problem is vital as library writers may wish to ensure their library is deadlock-free for any calling pattern. This method checks if it is possible to deadlock the library by randomly calling some set of its public methods. On detecting the possibility of deadlock, it provides the name of the methods and variables involved. The deadlock detector in

this method utilizes an interprocedural dataflow analysis.

By using this method, it is possible to track possible sequences of lock acquisitions inside a Java library. The analysis is flow-sensitive and context-sensitive. At each program point, the analysis computes a symbolic state modeling execution state of the library. At the end of a method, this symbolic state serves as a method summary. The analysis is executed frequently over all methods until a fixed point is reached [34].

One of the limitations of static analysis approaches is that they examine all the possible deadlocks and often give no false negatives, but they report many false positives. For example, the static deadlock detector developed by Williams et al. reports 100,000 deadlocks in Sun's JDK 1.4, while only 7 are real deadlocks [34] .

2.1.3 Dynamic Analysis

Dynamic program analysis techniques use an execution of program to find all potential deadlocks. They only visit feasible paths and have precise views of the values of variables because they operate at runtime. Here, we discuss three different dynamic analysis techniques as follows:

Agarwal and Stoller describe a runtime notion of potential deadlock in programs, written in any languages that use synchronization mechanism like locks, semaphores and condition variables [1]. They test the potential for deadlock by checking if there is any feasible permutation of the execution results in deadlock. The viability of such permutations is determined by ordering constraints amongst events in the execution. Havelund proposed the GoodLock Algorithm that detects potential deadlocks involving two threads [19]. This was later generalized to have any number of threads by

Bensalem et al. [6] and Agarwal et al. [2]. The algorithm presented Agarwal et al. [1] is extended in [2] to handle non block structured locking as well.

The second dynamic analysis technique we discuss is DEADLOCKFUZZER, developed by Joshi, Park and Naik [21]. This technique has two stages. In the first stage, a multithreaded program is executed and observed to find potential deadlocks that could happen in some executions of the program. This phase uses an informative and a simple variant of the Goodlock algorithm, called informative Goodlock or iGoodlock [19]. *iGoodlock* identifies potential deadlocks even if the observed execution does not end in a deadlock state. It provides debugging information suitable for identifying the cause of deadlock. The second stage uses debugging information provided by the first stage to create a real deadlock with a high probability. In the second stage, a scheduler is favored to generate an execution that creates a real deadlock that is reported in the previous stage with high probability of occurrence. A limitation of iGoodlock is that it can give false positives because it does not consider the happens-before relationship between lock acquisition and release into account. As a result, the user has to manually go through such potential deadlocks. This burden is removed from users by the second stage of DEADLOCKFUZZER [21].

Qi et al. presented an efficient dynamic deadlock detection tool called *MulticoreSDK* [29].The algorithm works in two phases: In the first phase, a reduced lock graph is constructed to identify program locations where lock operations may cause deadlocks. This phase examines the execution trace and constructs a lock graph based on the program locations of lock events. Once the lock graph is built, the algorithm finds cycles in it to compute a program locations set comprising of deadlock cycles. This set signifies program locations that may associate with deadlocks and is used in the second phase to filter deadlock free lock events.

In the second phase, an analysis report from the first phase is used to determine deadlocks in the lock graph with filtered locks. This phase examines the execution trace again, and constructs another lock graph based on the lock id of lock events. Particularly only lock events from the set derived from the first phase are recorded in the second lock graph. Finally cycles in this lock graph are found to determine potential deadlocks. By using two passes over the program trace and filtering out the locks that are not involved in deadlock, *MulticoreSDK* efficiently removes multiple lock nodes and edges which do not take part in deadlock formation [29]. Thus, it is more scalable for large real-world applications and more efficient in terms of memory utilization and time.

The downsides of dynamic deadlock detection techniques are that they suffer from lack of scalability and performance problems due to the large size of the lock graphs of large industrial strength applications and cannot handle them. Moreover, they can only find errors on a small number of execution paths as the number of feasible paths grows exponentially with the size of code. This renders the use of dynamic analysis impractical due to a lack of scalability. Another limitation of this approach is that dynamic deadlock detection has a very high computational price which causes runtime overhead. Hence, it is time consuming to run test cases, which makes this approach impossible for the programs with strict timing requirements. In theory, dynamic deadlock detection can compute arbitrarily accurate information, but in practice, they highly depend on the availability of computing resources.

2.1.4 Model Checking

Model checking is another way to find deadlocks in an application. Model checking takes a simplified description of the code and uses techniques to explore vast state spaces efficiently as it systematically tests the code on all inputs. Because of this, this approach does not need to explore massive state spaces in hardware circuits in order to find deadlocks. It analyses the correctness of concurrent reactive systems doing verification by state-space exploration.

An extension of model checking, *VeriSoft*, is presented by Godefroid [15]. VeriSoft directly deals with implementations of communication protocols written in programming languages such as C or C++. Thus it extends the state space exploration from modeling languages to programming languages. It is a tool to systematically explore the state space of systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages such as C or C++.

The algorithm can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results for finite acyclic state spaces. In addition to that, VeriSoft also checks for divergence and livelocks. A *divergence* occurs when a process does not attempt to execute any visible operation for more than a specified amount of time. A *livelock* occurs when a process has no enabled transition during a sequence of more than a specified number of successive global states. In practice, the algorithm can be used for systematically and efficiently testing the correctness of any concurrent system with or without acyclic state space [15].

Unfortunately, model checking fails to scale for large systems as it requires sig-

nificant effort both to specify the system, and in scaling it down enough to execute in the model checking environment. Due to the huge size of current Operating Systems, model checking an entire system of the size of an Operating System is still not possible. Hence, model checking techniques are only restricted to the verification of properties of models.

2.1.5 Type and Annotation Based Techniques

Type and annotation based techniques help to avoid deadlocks during coding by allowing programmers to specify a partial order among locks. Also the type checker statically ensures that the partial order among locks is maintained and well-typed programs are deadlock-free.

Boyapati, Lee and Rinard presented an ownership type system for multithreaded programs that allows programmers to specify a partial order among locks [7]. It allows them to partition the locks into a fixed number of equivalence classes and specify a partial order among equivalence classes. The type checker then statically ensures that whenever more than one lock is held by a thread, the thread acquires the lock in descending order. Well-typed programs in this system are guaranteed to be deadlock-free. The type system also allows programmers to use recursive tree-based data structures to describe the partial order.

This system allows changes to partial order through mutation to data structure at runtime. The basic idea for this system is that the programmers keep locking discipline in mind while writing multithreaded programs. It allows programmers to declare this locking discipline in the form of type declarations in their programs. It also statically verifies consistency of a program with its type declaration. The prototype of this

system is implemented in Java including threads, arrays, constructors, static fields, dynamic class loading, runtime downcasts, exceptions and interfaces.

Some limitations of this approach are that it imposes the burden of annotation on programmers and scaling it to larger systems is also not feasible.

2.2 Performance Evaluation Studies of Deadlock Handling Algorithms

Deadlock is one of the key issues faced in many different fields like routing, databases, operating systems, manufacturing systems and others. Most of the research carried out in this field focuses on solving the problem of deadlock. Some of the proposed solutions have also undergone comparative performance evaluations but there has not been any research done on comprehensive performance comparisons of those solutions. Studies [8, 24, 25, 13] and [35], entail comparative performance analyses. Among these, [25] and [35] talk about the detection of deadlock in distributed systems, [8] and [24] provide a performance analysis for the solutions of deadlock detection techniques in Distributed Database Systems and [13] details a comparative performance analysis of a deadlock avoidance control algorithm for Flexible Manufacturing Systems.

A probabilistic performance analysis of a deadlock detection algorithm in distributed systems is presented by Lee and Kim [25]. The distributed deadlock detection algorithm declares deadlock upon finding back edges in a distributed search tree constructed by the propagation of probes. The tree is built as follows: The initiator of the algorithm becomes the root of a tree. Then it sends out probes, called ASK, to all of its successors at once. If a node receives the probe for the first time, it becomes

the child of the sender of the probe. The probe is then further propagated until it reaches an executing node or a tree node that has already received a probe. Each tree is assigned a unique path string to represent the level of the node in the tree and to distinguish one branch from another in the tree. With the help of path strings, not only back edges are identified, but other types of edges are also found, such as cross and forward edges. The ASK probe carries the candidate victim identifier which has the lowest priority among those nodes visited. If the candidate victim is inside the detected deadlock cycle, it will receive an abort message [25].

The performance of this algorithm is evaluated on measures such as deadlock duration, number of algorithms initiated, and mean waiting time of a blocked process. *Deadlock duration* is the elapsed time until a deadlock is detected after it is formed. For distributed systems, a detection algorithm can be started and executed by several sites simultaneously. That may degrade performance of the system by generating too many messages. So the expected number of algorithm initiations throughout the system is also an important measure.

Bukhres compared two distributed deadlock detection algorithms for distributed database systems [8]. The algorithms are from two different categories, one is centralized and the other is distributed. Centralized algorithms maintain the wait-for-graph (WFG) at the control site. Control site is designated to perform a deadlock detection activity by gathering the relevant information from all the other sites. At this site the graph is updated and the search for the cycle involving deadlock is carried out. In distributed algorithms, the WFG is maintained at a different site in the system. A cycle may be composed of different transactions at different sites, and several sites participate in deadlock detection.

Centralized algorithms have poor reliability compared to distributed algorithms

because the whole system fails if the control site fails in the centralized system. However, distributed algorithms are difficult to design and implement as detection of deadlock can be initiated from any site. Simulation studies presented by Bukhres show that the centralized algorithms perform a little better than the distributed algorithms under a lightly-loaded condition. Whereas under a heavily-loaded condition, the distributed algorithms perform better than the centralized algorithms [8]. The measures taken in to consideration to compare performances are throughput, restart rate, deadlock life time and the effect of request pattern against throughput and number of messages. The conclusion for this study is that the choice of the best deadlock detection algorithm is dependent upon the operating region of the system.

The problem of evaluating and comparing the performance of deadlock avoidance control policies applied to Flexible Manufacturing Systems (FMS) is addressed by Ferrarini [13]. The problem is discussed with both timed and untimed models. For both models the problem is considered with and without deadlock avoidance control policies. Some of the key metrics measured are deadlock percentage, frequency of occurrence, deadlock rejection percentage, unreachable stage percentage, product survival rate and blocking capacity [13].

From the above discussion, we observe that there is no simulation tool available to study and compare performance of different deadlock handling approaches in multicore systems. Moreover, the studies mentioned above are very old and not available to further extend their work on. Hence, we believe that our simulation testbed will be really useful to study and compare performance of different deadlock handling approaches in multicore systems.

2.3 Summary

In this chapter, various approaches that have been developed to handle deadlock in multicore systems were discussed. Then, we presented a brief survey on different performance evaluation studies done on deadlock in distributed systems, distributed database systems, and a flexible manufacturing. In the next chapter, we discuss implementation detail of simulation testbed to study deadlock handling approaches in multicore systems.

Chapter 3

Simulation Framework for Deadlock in Multicore Systems

This chapter describes the current implementation of simulation framework for deadlock handling algorithms in multicore systems. The implementation is done by using Discrete Event Simulation (DES). Before describing the actual framework, DES is briefly explained.

3.1 Simulation

A computer simulation is a technique that attempts to model and observe the behavior of an abstract module of a particular system. A simulator is a computer program to transform the states of a system in discrete time points over a specific period of time. Simulation is a very effective tool to study the dynamic behavior of complex systems.

Computer Simulations can be classified in two types, either discrete or continuous, based on how the system state is modeled and simulated. In the continuous simulation, the state variables of the system change continuously over time while in discrete simulation the state variables change only at discrete times. Based on the advancement of simulation time and the updates of the system state, discrete simulation can

be further divided into two types: time-stepped and event-driven. As names suggest, in time-stepped simulation the system state is updated at every time step while in event-driven simulation, the states of the system are updated at the occurrence of events.

Özgün and Barlas presented a comparison between approaches of discrete event simulation and continuous simulation on a simple queuing system [28]. Discrete event simulation comprises an event list, a simulation clock and an event scheduler. The simulator maintains a queue of events sorted according to the occurrence of their simulation time. The simulation time is advanced to the time of occurrence of the next event in the event list by a Simulation clock. The system states are changed by each execution of events, which is done by an event scheduler.

For example, in simulating the behavior of university students, the number of students enrolled and the number of students graduated are state variables and they will be updated on the occurrence of the events in the system. The number of students enrolled will be updated at the start of each semester and the number of students graduated will be updated when the students graduate at the end of academic year. Simulation ends at the time when the event list is empty or simulation time is up. The advancement of a simulation time can be the same, faster or slower than real-time as it is not important to execute simulation in real-time. For example, in the university example, the simulation does not have to wait for an entire year length and it can be faster than real-time. Using the example of a driving test, simulation can be exhibited in real-time speed. On the other hand, protein synthesis in a cell can be simulated slower than real-time.

We mainly use discrete event simulation to implement different components of our framework. Next, we move onto a discussion of the building processes of our frame-

work. As we expanded the multicore scheduling framework, developed by Manickam under the guidance of Aravind, to simulate and study deadlocks. We first explain the architecture of the Multicore Scheduler System framework.

3.2 Architecture of Multicore Scheduler System Framework

To observe deadlock in multicore systems, first, we need a framework that simulates the behavior of multicore systems. For that purpose, we are using an architecture developed by Aravind and Manickam [3]. We first briefly explain the architecture of the multicore scheduling simulation (MSS) framework. Later, we explain the expansion to this framework, that is, what are the added components and how they are inserted into MSS. Although the MSS framework consists of different scheduling algorithms, we are using only Linux Completely Fair Scheduling in our system.

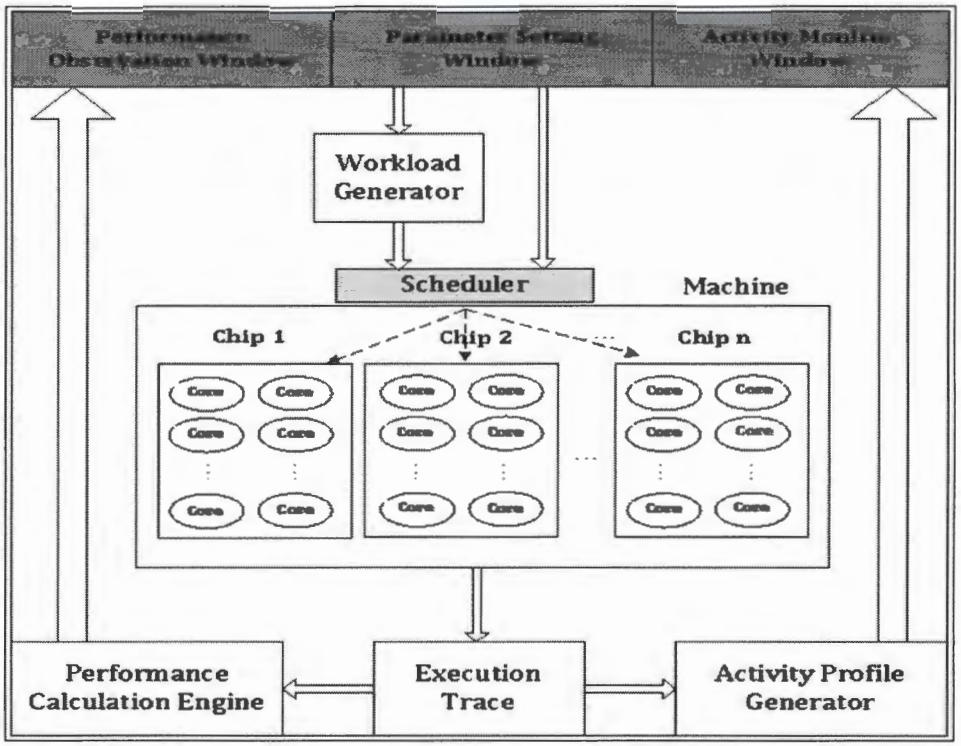

Figure 3.1: Architecture of MSS Framework

Source: A Flexible Simulation Framework for Multicore Schedulers [3]

The higher level architecture of MSS is shown in Figure 3.1. There are five key components in it: workload generator, machine, scheduler, execution trace and performance calculation engine. We explain them next.

3.2.1 Workload Generator

The workload generator is implemented to generate threads. We have to give the number of threads, thread's mean arrival rate, and thread's mean service rate as input to generate a workload. In the output, each thread will have a unique id,

arrival time, execution time, I/O points, priority and application class. The times when a thread will go for I/Os are determined by its I/O points. For our system, we added three scenarios to workload generator: (i) General scenario, (ii) Scenario for Dining Philosopher's problem, and (iii) Scenario for the Banker's algorithm. We also added more numerical inputs for resources and instances per resource.

3.2.2 Machine

All the computations are done in the machine which is also called a multicore machine. The machine has a hierarchical structure with the machine at the highest level, followed by chips and cores. One or more chips can fit on a machine and two or more cores can be in each chip. In the simulation context, each core is capable of executing one thread at a time. Cache memory is also an important part of machine, which is also structured in a hierarchical manner. Current multicore system have three - L1, L2 and L3 levels of cache memories. L1 cache is a core level local cache, L2 cache is shared among the cores in a chip and all the cores in the machine share the L3 cache.

3.2.3 Scheduler

The implemented framework for a scheduler has two tasks: maintaining the load among the cores and multitasking threads in each core. According to their functions, the first task is referred to as load balancing and the second task as load scheduling, hence the components in the implementation are called load balancer and load scheduler. The load balancer is responsible for dispatching the new jobs to the appropriate local scheduler, and for migrating jobs from one to another local scheduler. The task of local scheduler involves scheduling jobs to the cores for execution. The simulation

gives the local scheduler centralized control. From the simulation point of view, the tasks of the local scheduler are to make a decision to choose a job for execution, to determine the amount of execution time, to calculate the progress rate and to produce the trace. The progress rate of is a crucial design factor as it can also affect the accuracy of execution. The progress rate of a job is dependent on factors such as execution speed of the core and the contention for shared resources. The implementation uses a simple cache contention model which can easily be replaced with the implementation of a more refined model.

3.2.4 Execution Trace and I/O Trace

During the simulation, two types of traces are recorded: the execution trace and the I/O trace. In order to generate an activity profile and to calculate performance metrics, the execution trace is collected at every context switch. The format of the execution trace is a quintuplet: <core id, thread id, execution start time, scheduling end time, status>. The status can have values between 0-3 where status is 0 if the thread is preempted by quanta expiration, 1 if by a thread with a higher priority, 2 if the thread is going for I/O and 3 if completed. The format of I/O thread is a triple: <thread id, I/O start time, I/O end time>. The performance calculation engine calculates the values of criteria such as response time, fairness and utilization of resources for a given set of data, and the result can be passed to the performance observation window. The users can then study results from the performance observation window.

3.2.5 Performance Calculation Engine

The performance study mainly focuses on determining how well the algorithm attempts to fulfil criteria such as response time fairness and utilization of resources. Users can study algorithms based on these criteria, calculated and passed to the performance observation windows by the performance calculation engine. The performance criteria involve wide range of metrics such as interactive response time, CPU utilization, throughput, turnaround time, waiting time and response time.

Next, we will discuss the actual simulation and architecture aspects of Deadlock in Multicore Systems.

3.3 Simulation of Deadlock in Multicore Systems

The basic deadlock environment involves processes and resources in it. Resources can have one or more instances and processes can request up to a maximum number of instances of resources available. If the requested resources are available, they could be granted access to the process that requested them otherwise that process can wait until the resources are available, meanwhile holding on to current resources.

The simulation of deadlock in multicore systems has three main components: entities, events, and states, in addition to the components of the Multicore Scheduler Simulation framework (an event list, a simulation clock and an event scheduler) described in the previous sections. The event scheduler manages all of the events from an event list according to the simulation clock. Each event from an event list occurs at a particular instance of time and changes the state of the entity. Here we consider

a system as an entity as well. For our system, entities, states, and events are shown in Table 3.1. We define these terminologies first, and then we discuss the simulation aspect of the system.

Entity	States	Events
Process	Waiting Running Blocked Completed	Request Acquire Release Going for I/O
Resource	Acquired Free	
System	Deadlocked Deadlock Free	

Table 3.1: Entities, States and Events

- **Processes**: Processes are the main entities in the system. A process requires resources while executing in its critical section. A critical section is a part of the executing process which uses shared resources exclusively. A process can have different states such as running, waiting, blocked or completed. Due to their concurrent access to finite resources, deadlock could occur in the system.

- **Resources**: Resource can be virtual entities such as memory, data structures and CPU time or physical entities such as printers, scanners and other I/O devices. A resource can have one or more units of the same type, which are called instances of a resource. Different instances of a resource can be shared among different processes but an instance of a resource can be held by only one process at a time. A resource can be either free or held by a process.

- **System**: A system consist of all the processes, resources and their instances and techniques to handle deadlock.

- **Deadlock**[4]: "Deadlock is a situation in a resource allocation system in which two or more processes are in a simultaneous wait state, each one waiting for one of the others to release a resource before it can proceed."

The main events are request, acquire and release of resources by processes. These events change states of the system as either deadlocked or deadlock free. For processes the state changes affiliated with these events are waiting, running, blocked or completed. If a process acquires all the requested resources then it is in the running state, if the process can not acquire all the requested resources due to unavailability of them, then the process is in a waiting state, holding its current resources. Though I/O is a physical type of resources that can be accessed by processes, we consider I/O as a special case of resources, hence events associated with processes accessing I/O are: going for and coming back to a waiting state from I/O. When a process goes to I/O the state is changed to blocked and when all the requests for a process are satisfied, its state is changed to completed.

The simulation of the system basically includes updating the state of the entities at every simulation point as well as increasing the simulation clock. It involves various interactions between these entities and occurrence the of deadlocks due to that. With the help of these components, various scenarios and solutions of deadlock can be simulated using DES. The simulation records the executions as a simulation trace, and they are recorded at every occurrence of an event. With this background we further introduce the architecture of deadlock in multicore system.

3.4 Architecture of Simulation Framework for Deadlock in Multicore Systems

To simulate a basic deadlock prone environment, we need resources, processes that can request these resources and a system (Resource Manager) to manage the resources for the requests made by processes. From the point of view of the system, processes communicate with each other through shared memory using read and write operations in a multicore system. During execution, processes require resources. A process requests resources, acquire them if granted, access and then release after using them. As explained above, the MSS Framework has threads, machine and scheduler. The functions of processes and threads are the same, so hereafter we use the term "process" in general.

In order to simulate deadlock on top of the MSS framework, we added extra components such as resources, a Resource Manager, and the ability for the processes to request resources as well as to communicate with each other. Alongside these additional components, architectural level changes are also required for some existing components to fulfil the criteria of simulating deadlock. Later, in this chapter, we explain in detail the components related to the implementation of deadlock scenarios and viable handling techniques.

The higher level architecture of the Multicore Scheduler Simulation (MSS) framework has five main logical components as shown in Figure 3.1: workload generator, multicore machine, multicore scheduler, execution trace and performance calculation engine. Out of those, the core simulation engine involves three basic components as shown in Figure 3.2: workload generator, multicore scheduler and multicore machine.

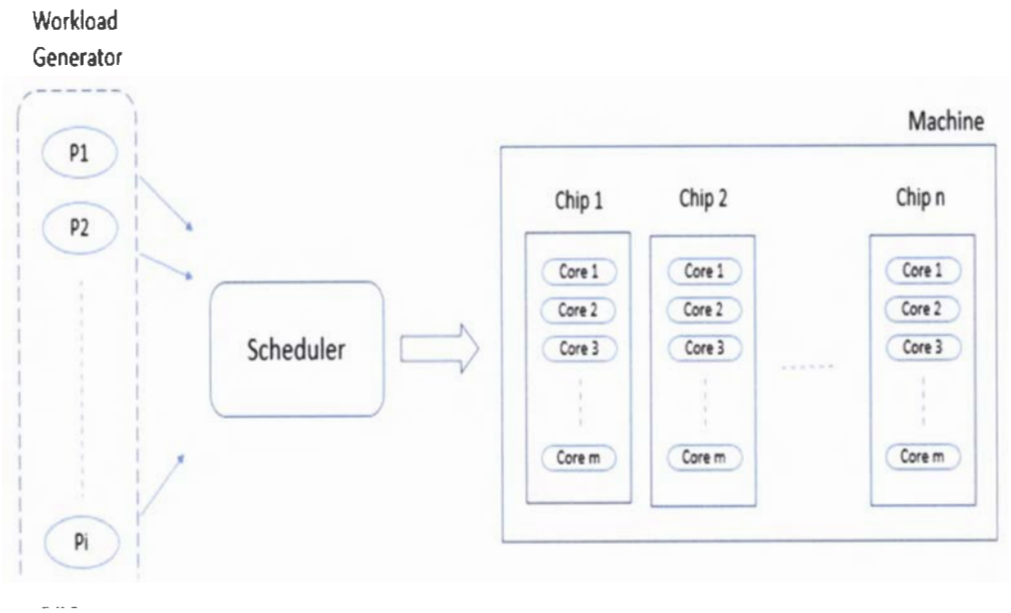

Figure 3.2: Core Architecture of MSS Framework

As discussed above, we need resources that can be accessed by processes to simulate a system compatible for deadlock. In order to manage these resources, we need a module, what we refer to as the Resource Manager (RM). The above mentioned system can be portrayed as shown in Figure 3.3. A detailed explanation of RM and its implementation details are as follows:

Resource Manager: For the smooth and flawless execution of system, multiple requests by processes to access different resources have to be managed well. Improper management of resources may lead the system to an unwanted situation called deadlock. To prevent the system from deadlock or to avoid this situation of deadlock in the system, granting resources to multiple processes according to their requests and availability of resources is managed by a Resource Manager (RM).

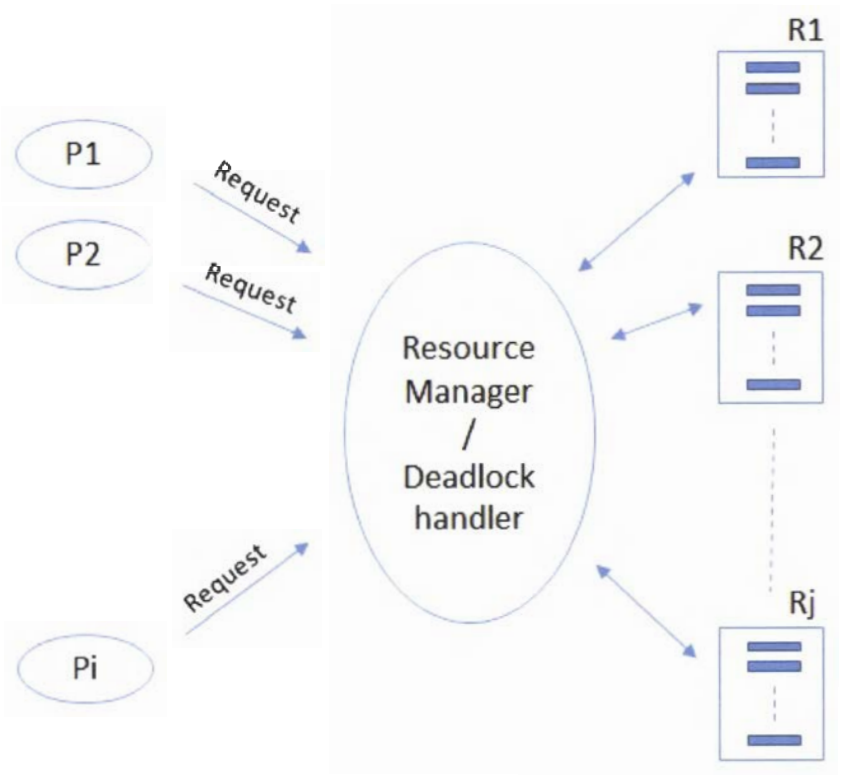

Figure 3.3: Basic System Requirements for Deadlock

The notations of Figure 3.3 are same as Figure 1.1 with additional notation of small, dark rectangles inside resources that represent instances of that resource.

The basic function of RM is to manage resources - grant resources to the processes based on availability of resources and according to the requests they make in a way that the system does not go into a deadlock state. If RM functions properly, deadlock can be avoided but improper functioning of RM can lead the system towards a deadlock situation. The implementation of RM depends on what strategy needs to be implemented to handle deadlock. For deadlock prevention and deadlock avoidance, the logic to handle deadlock can be incorporated into RM. On the other hand, deadlock detection and resolution may require a separate implementation than RM

to handle deadlock.

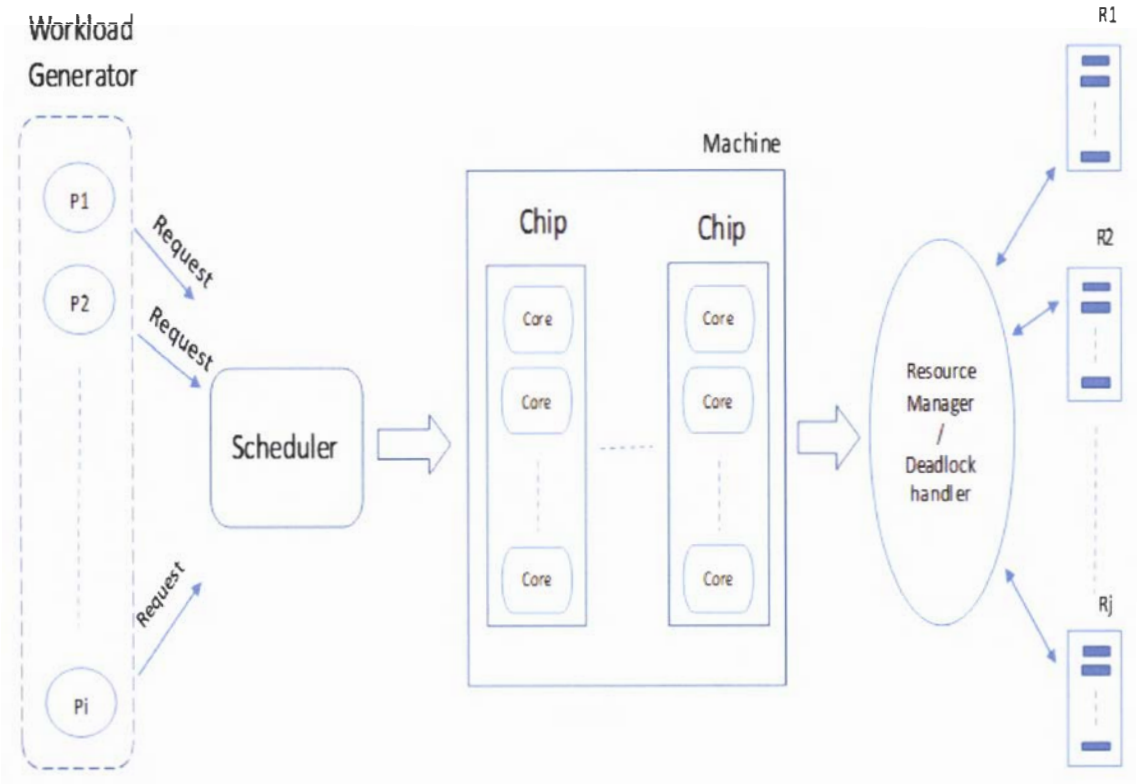

Figure 3.4: Architecture of Simulation Framework for Deadlock in Multicore Systems

The resultant architecture after incorporation of resources and Resource Manager to the core MSS framework of Figure 3.2 is shown in Figure 3.4., which has components from both Figure 3.2 and Figure 3.3. Now, we discuss the additional and modified components of the architecture of Deadlock in Multicore Systems in detail.

3.4.1 Workload Generator

A workload generator is mainly used to generate processes and resources while setting parameters that are necessary for simulation. We can generate three types of

simulation scenarios using the workload generator: (i) General scenario (ii) Scenario for Dining Philosopher's problem (iii) Scenario for Banker's algorithm. Based on the selected scenario, criteria such as limiting resource counts to equal as process counts, instance count per resource, and runtime are set for the given workload. Requests pattern by processes also depends on the scenario type. These scenarios are simulated on Linux Completely Fair Scheduler.

- **General Scenario**: To generate a general scenario workload, required input parameters are: the number of processes, the number of resources, instances per resource type, mean arrival rate and task period range. Also, there are no restrictions on input values of any parameter. The generated workload will have a set of processes and resources. Each process will have a unique id, arrival time, execution time, I/O points, priority, application class, request counts and other parameters required for simulation. Likewise, each resource will have a unique id, instance count and various other parameters. All the parameters that do not require manual input are generated randomly, setting an upper bound on them through the workload generator.

- **Scenario for the Dining Philosopher's problem**: This scenario implements the Dining Philosopher's problem, which can be considered a special case of general scenario. Dining philosopher's problem was introduced by Dijkstra [10]. Five philosophers [$P1$,...,$P5$] sitting around a table as shown in Figure 3.5. The table has a large serving bowl of spaghetti, one small bowl for each philosopher and five forks. A philosopher wishing to eat can take and eat some only if he or she has two forks on either side of the bowl. The problem states that, considering the above scenario, devise a solution that will allow the philosophers to eat by satisfying mutual exclusion while avoiding deadlock and starvation.

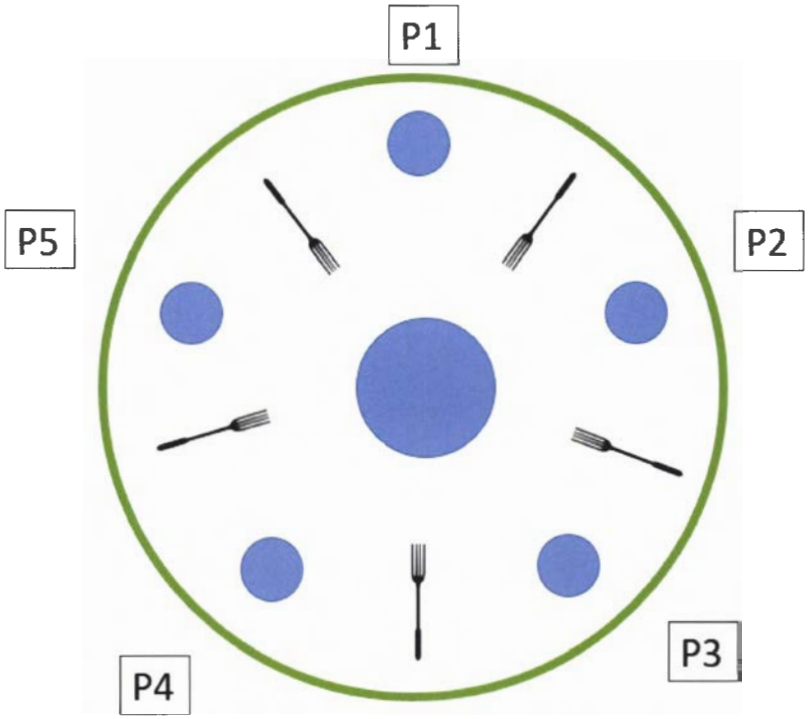

Figure 3.5: Dining Philosopher's Scenario

Here, we limit ourselves only until simulating the scenario and not the solution. Any solution that is implemented for a general scenario can also be used to solve the dining philosopher's problem as it is a special case of general scenario. To simulate this, we put a limit on the number of resources, that is, the number of resources have to be same as the number of processes. Also, we limit the instances per resource type to one. We also have to input runtime beforehand to specify how long the simulation should run. Further requirements to simulate a scenario for the Dining Philosopher's problem include limiting each process to request only its neighbor resources and limiting the maximum count for requests per process to two.

- **Scenario for Banker's Algorithm**: This scenario requires the same input

parameters as the general scenario but parameters such as maximum need for each resource are generated differently than a general scenario to satisfy the requirements of the Banker's algorithm. A Banker's algorithm requires advanced information about processes to execute such as the maximum claimed instances of resources by each process. This information is generated with the workload. The generated workload will have a set of processes and resources. Each process will have the usual parameters set such as a unique id, arrival time, execution time, I/O points, priority, application class, request counts and other parameters required for simulation, with additional parameters such as maximum claim and need. The resources have the same parameters generated as a general scenario with the workload.

There is one more a notable difference between all the scenarios in execution. The general scenario and the scenario for the Banker's algorithm run until all the processes finish their executions. That is, all of the requests are satisfied for all of the processes. For the Dining Philosopher's problem, execution runs until the runtime is reached.

3.4.2 Scheduler and Machine

The implementation of a scheduler is not primary goal for our system. Hence we have used Linux Completely Fair Scheduler, implemented by Manickam V. [3]. Simulation of a machine simply reflects the simulation of the multicore system in it, therefore we have used that component with our system.

3.4.3 Resource Manager

As explained earlier, management of resources can be done in this part of the system so we call it Resource Manager (as shown in Figure 3.4). Solution strategies for deadlocks can also be implemented here. There are three types of solutions to a deadlock problem: deadlock prevention, deadlock avoidance or deadlock detection and resolution. *Deadlock prevention* works by preventing one of the four Coffman conditions [9] from occurring in the system, as discussed in Chapter 1. *Deadlock avoidance* makes sure that for every resource request, the system does not enter into an unsafe state and grants the requests that will lead to safe states. An unsafe state is a state that could result in deadlock. *Deadlock detection and recovery* allows deadlocks to occur, on occurrence of deadlock the state of the system is examined and the deadlock is resolved either by termination of one or more processes or by preempting resources.

We have implemented a deadlock avoidance technique, a deadlock detection technique, and deadlock recovery techniques in our system. The avoidance technique is the Banker's algorithm and the implemented detection technique is Dreadlocks [23], whereas we use different criteria to select and terminate one or more processes involved in deadlock.

3.5 Algorithms Implemented

In this section, we explain the Banker's algorithm and Dreadlocks in detail. The banker's algorithm is deadlock detection technique and Dreadlocks is used for deadlock detections.

3.5.1 Deadlock Avoidance: Banker's Algorithm

The Banker's deadlock avoidance algorithm was developed by Dijkstra in mid-1960s [11]. This algorithm requires all processes to declare their maximum requirements for all the resources at the start of execution. A process need not request all of its required resources at the beginning as the requests are made sequentially. Once a process requests the maximum declared resources, it will acquire them for a finite time and then release them. Now, these released resources are also available for allocation to other processes. At every new request, the algorithm checks whether the system stays in a safe state or not. A *safe state* of a system is when all the processes can complete their execution without forming a deadlock [17]. It is determined by simulating the allocation of maximum possible instances of all resources to a process and then by finding a sequence of such processes that can finish their execution without forming a deadlock. A request is granted if on allowing the request, the system stays in a safe state, otherwise the request is denied. Further, the system also determines a dynamic order of processes to execute in order to avoid deadlock.

Table 3.2: System in a Safe State

(a) $Max[i, j]$

	$R1$	$R2$	$R3$
$P0$	7	5	3
$P1$	3	2	2
$P2$	9	0	2
$P3$	2	2	2
$P4$	4	3	3

(b) $Alloc[i, j]$

	$R1$	$R2$	$R3$
$P0$	0	1	0
$P1$	3	0	2
$P2$	3	0	2
$P3$	2	1	1
$P4$	0	0	2

(c) $Avail[j]$

$R1$	$R2$	$R3$
2	3	0

Now we explain the Banker's algorithm with an example. Consider a resource allocation system with five processes $P0$, $P1$, $P2$, $P3$ and $P4$ and three resources $R1$, $R2$ and $R3$. The maximum need for resources j for processes i is represented by $Max[i, j]$. $Alloc[i, j]$ represents the current allocation of resource j to process i and $Avail[j]$ represents currently available instances of resource j. The current safe state of the system is shown in Table 3.2 for (a) $Max[i, j]$, (b) $Alloc[i, j]$ and (c) $Avail[j]$. The system is in safe state because all the processes can finish their execution in order of $P1$, $P3$, $P4$, $P0$ and $P2$. Once completed, $P1$ can release the acquired resources. These resources are then added to the available resources, which in turn can be used by process $P3$. In the same way processes $P4$, $P0$ and $P2$ can finish their executions.

Table 3.3: System in An Unsafe State

(a) $Max[i, j]$

	$R1$	$R2$	$R3$
$P0$	7	5	3
$P1$	3	2	2
$P2$	9	0	2
$P3$	2	2	2
$P4$	4	3	3

(b) $Alloc[i, j]$

	$R1$	$R2$	$R3$
$P0$	0	3	0
$P1$	3	0	2
$P2$	3	0	2
$P3$	2	1	1
$P4$	0	0	2

(c) $Avail[j]$

$R1$	$R2$	$R3$
2	1	0

Now, suppose the system grants request (0, 2, 0) of process $P0$. The resultant state of the system is shown in Table 3.3. This is an unsafe state as there is no such sequence available for processes that can finish their executions. In this case the request (0, 2, 0) of $P1$ is denied by the Banker's Algorithm.

The downside of the Banker's algorithm is that it is quite expensive as it may take $O(n^2m)$ time for each execution, where n is number of processes and m is number of resources [17]. It also needs some information in advance such as the maximum number of required resources. In most of the current systems, this information is not available.

3.5.2 Deadlock Detection: Dreadlocks

Dreadlocks is a deadlock detection technique for shared memory multiprocessors [23]. In Dreadlocks, each process maintains a digest of the waits-for graphs. A *digest* of Process P, denoted $\mathcal{D}_P$, is the set of other processes upon which P is waiting, directly or indirectly. If a process is not trying to acquire any resources then the digest of that process contains only itself. If a process is trying to acquire a resource, then the digest of that process includes itself as well as the digest of the processes that are currently holding the requested resources. Changes to this digest of waits-for graphs are propagated as processes acquire and release resources, and a process detects deadlock when it finds own appearance the digest. Due to this approach, a probing mechanism and the maintenance of explicit waits-for graphs can be avoided. To explain this with an example, consider Figure 3.6.

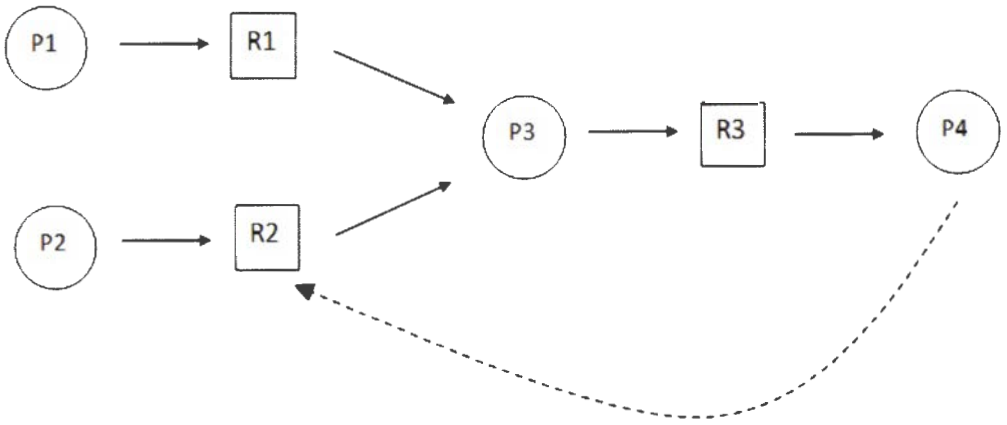

Figure 3.6: Example of Dreadlocks

The notations for Figure 3.6 are the same as Figure 3.3 and Figure 1.1, with addition of a dashed arrow, which indicates that a process is trying to acquire a resource. Process P1 is requesting for resource $R1$, held by process $P3$, that is requesting re-

sources $R3$ which is held by process $P4$. Also, process $P2$ is requesting resource $R2$, held by process $P3$ that is requesting resources $R3$ which is held by process $P4$. In this case, digests for these processes are as follows:

$$\mathcal{D}_{P1} = \{P1,P3,P4\}$$

$$\mathcal{D}_{P2} = \{P2,P3,P4\}$$

$$\mathcal{D}_{P3} = \{P3,P4\}$$

$$\mathcal{D}_{P4} = \{P4\}$$

In the next state, the dotted line, from process $P4$ to resource $R2$, indicates that $P4$ is trying to acquire $R2$. But according to the algorithm, $P4$ finds itself in the digest of $P3$ which is currently holding $R2$, detects a deadlock and aborts. Once a deadlock is detected, the next step is to resolve the deadlock. We used process termination techniques to resolve the deadlock situation. This procedure is explained next.

3.5.3 Deadlock Recovery: Process Termination

Deadlock recovery can be achieved by aborting one or more processes involved in the deadlock. To select these processes, we have certain criteria described as follows:

Process Termination Criteria

Once the existence of the deadlock is detected by Dreadlocks, victim selection is required to select one or more processes to terminate. We select either of the two processes - the process that is requesting resources or the process whose digest contains the requesting process, to be terminated in order to resolve deadlock. The criteria to select which process to terminate are as follows:

i *Random*: The selection criteria is random. Any process can selected to be aborted randomly.

ii *First In First Out*: The process which entered the system the first is selected to be terminated.

iii *Least Recently Used*: The process that has utilized the system the last is selected to be terminated.

iv *Minimum Runtime*: The process that has spent the least time in the system is selected to be aborted.

v *Maximum Runtime*: The process that has spent the most time in the system is selected to be aborted.

vi *Smallest Digest*: The process with the smallest digest is selected for termination.

vii *Largest Digest*: The process with the largest digest is selected for termination.

User can enable or disable option for the algorithm to detect the deadlock. Process termination option is available only if deadlock detection is enabled for the particular simulation run. If users select scenario for the Banker's algorithm then the detection

and termination options are not made visible. Next, we show how to operate the system by showing and explaining the user interface of it.

3.6 User Interface

Our simulator has two main windows: a Performance Parameter Setting Window and a Performance Observation Window as shown in Figure 3.7.

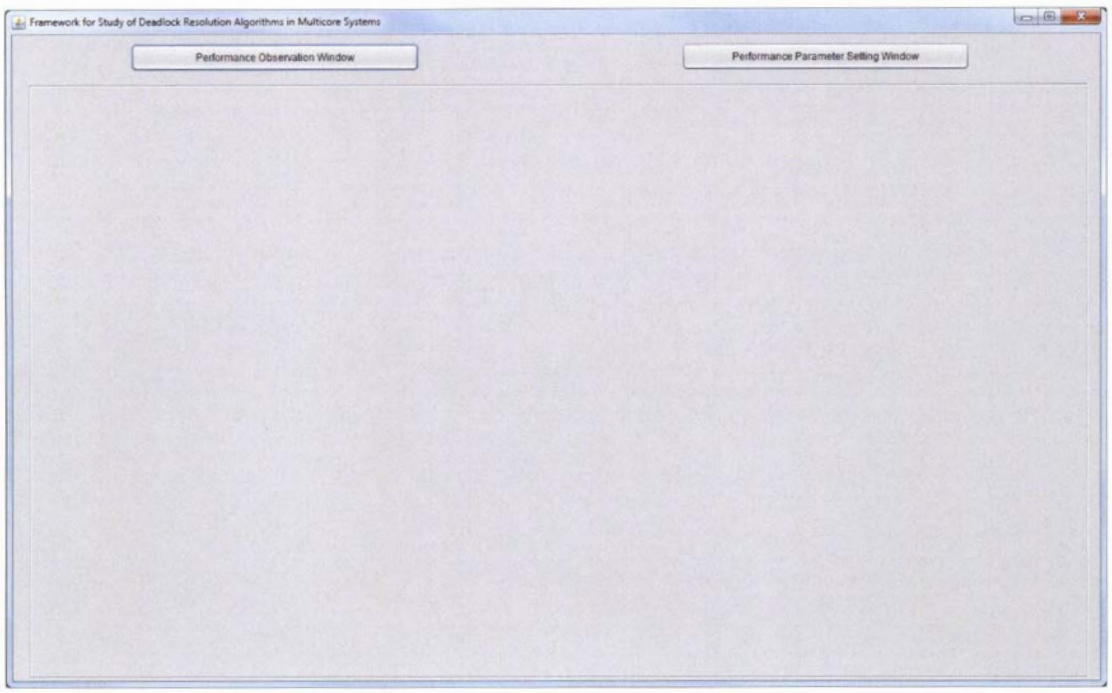

Figure 3.7: Main Window of the Simulator

A user can click on either the 'Performance Parameter Setting Window' or 'Performance Observation Window'. Each of the windows has different panels in them, to set different parameters for simulation. We now explain these windows.

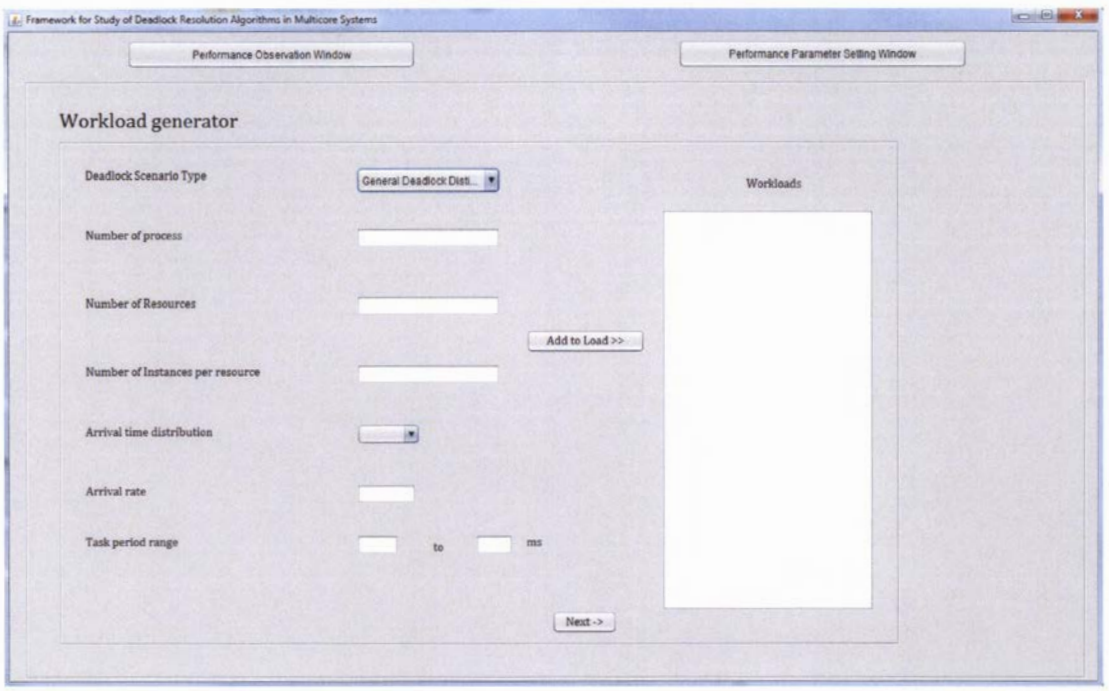

Figure 3.8: Workload Parameter Setting Window

3.6.1 Performance Parameter Setting Window

The Performance Parameter Setting Window has three panels in it. One panel is used to input parameters that are required to generate workloads. The second panel is used to set configuration of multicore machine and deadlock detection algorithm parameters. The third panel is used to start a simulation after all the required parameters are set.

The workload generator panel shown in Figure 3.8 takes the scenario type, the number of processes, number of resources, instances per resource type, arrival type, arrival rate and the task period range as input. Users can also create more than one workload at the same time. The input parameters depend on the selected scenario type, for example, in a general deadlock scenario and a scenario involving the Banker's

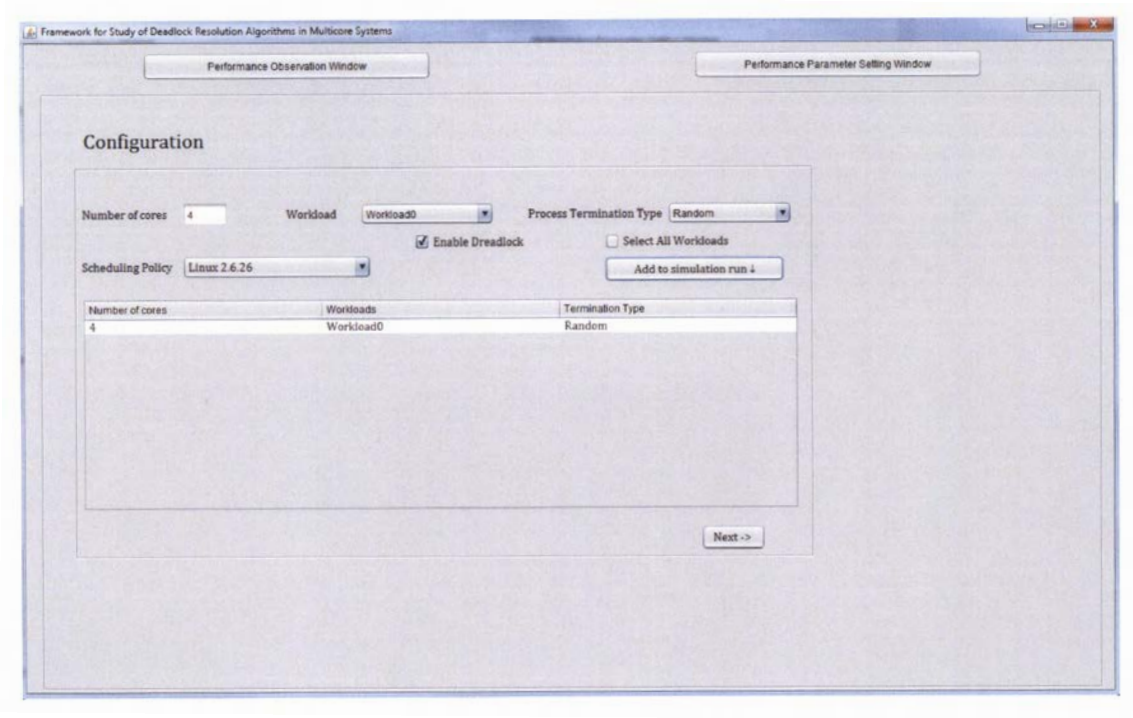

Figure 3.9: Configuration Parameter Setting Window

algorithm, users have to enter a value for instances per resource count, and there is no restriction for the number of processes or resources. On the other hand, for a scenario involving the Dining Philosopher's problem, users have to enter runtime instead of instances per resource count. Also, the number of processes and resources have to be the same to satisfy the scenario criteria. After entering the above mentioned parameters for each workload, users have to click on 'Add to Load' button. The added workload can be seen on the right side of the panel. If more than one workload is added, the additional workload is appended to the existing list of workloads. After all the workloads are added, by clicking on the 'Next' button at the bottom of the panel, users can go to the configuration panel.

The configuration panel in the Performance Parameter Setting Window is shown in Figure 3.9. This panel is used to configure the multicore machine and deadlock

Figure 3.10: Simulation Run Window

algorithm to create simulation runs. Several simulation runs can be configured using this panel. The parameters required to configure the machine include setting the number of cores for each workload. To set parameters for the deadlock algorithm, users can either enable or disable Dreadlocks' deadlock detection using different workloads. Furthermore, if Dreadlocks is enabled, users can set a criteria to select processes for termination from those involved in the deadlock. These options are not visible if users have selected the Banker's algorithm scenario from the Workload Generator window. After entering all the parameters, by clicking the 'Add to simulation run' button, the simulation run is added using the selected workload. If the 'Select All Workloads' option is enabled, multiple simulation runs are added using all the workloads. Once all the simulation runs are added, pressing the 'Next' button at the bottom of the panel will bring the next panel to start the simulation.

The third panel is shown in Figure 3.10. By clicking on the 'Start Simulation' button, users can start simulation for all the simulation runs that are added in the previous panel.

3.6.2 Performance Observation Window

Using the Performance Observation Window, as shown in Figure 3.11, various performance metrics can be represented in charts. To select a performance metric, the Chart Type drop-down menu can be used.

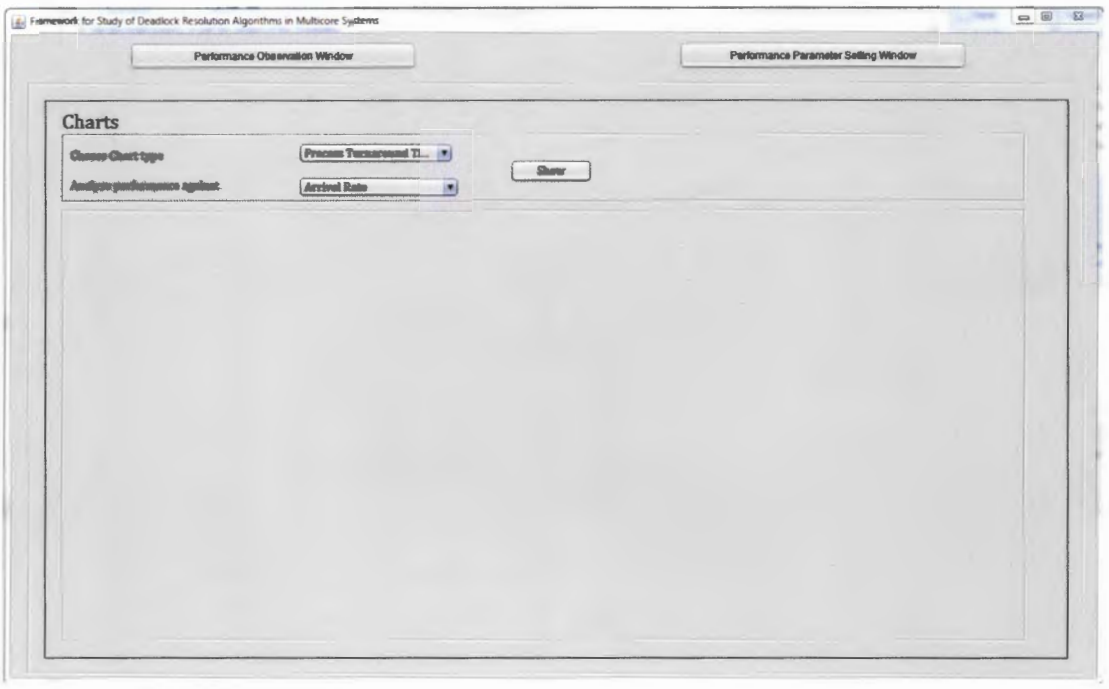

Figure 3.11: Performance Observation Window

Also, the results can be analyzed with respect to the arrival rate or number of cores,

depending on the different parameters selected while creating workloads or configuring simulation runs. After selecting the appropriate choices, pressing the 'Show' button will display a results chart in the chart display area.

3.7 Summary

First, we explained the core components of the framework for simulation of the multicore scheduling algorithms, developed by Manickam and Aravind. We, then presented an expanded MSS framework for the simulation of deadlock handling algorithms in multicore systems. The framework is flexible and competent to incorporate different deadlock handling algorithms for multicore systems in it. In the next chapter, we discuss traces generated during execution and the computation of performance metrics from traces.

Chapter 4

Execution Trace and Performance Evaluation

The previous chapter explained the entire procedure from operating the framework for deadlock in multicore systems to starting the simulation. During the simulation, we record different types of traces for the purposes of performance evaluation. Here, we discuss traces and performance evaluation in detail.

4.1 Traces

There are four types of traces in our system. They are execution trace, I/O trace, request trace and deadlock (DL) trace. Execution trace has format of a quintuplet: **<core id, thread id, execution start time, scheduling end time, status>**. The status can have a value between 0-3. Value 0 means that thread is preempted by quanta expiration. Value 1 means the thread is preempted by other higher priority thread. The value of status as 2 indicates that the thread is going for I/O and 3 means that the thread is completed. The I/O trace has formate of a triplet: **<thread id, I/O start time, I/O end time>**. The execution trace and I/O trace are retained same as described in the framework of the MSS framework. This was done in order to

preserve multicore properties and to calculate performance metrics that are related to multicore systems but are affected by deadlock algorithms, such as Turnaround time and Throughput.

Request Trace: The format of the request trace is **<rid, req-wait, req-grant, req-release>**. The request id is denoted by rid, the time a request started to wait is denoted by req-wait, the time a request is granted is denoted by req-grant and the time when a request ends or a process releases occupied resources is denoted by req-release.

The request trace helps in observing the performance metrics of the whole system which includes the effects of the additional components such as Resource Manager and resources as well as the implemented deadlock algorithm. The deadlock trace particularly helps in computing performance metrics pertaining to implemented deadlock algorithms. From this trace type we can calculate performance metrics such as request wait times which are eventually used in computing the total wait time for a process or a system.

Deadlock Trace: The deadlock (DL) trace has a quintuple format as shown in Figure 4.1. This includes the process id (pId), deadlock id (dId), deadlock start time (DLStart), deadlock end time (DLEnd) and status (Status). The process id refers to the id of the process that is involved in the deadlock. The deadlock id is set to distinguish different deadlocks, that occur in the system. The deadlock start time and deadlock end time correspond to the time a deadlock is started and the time a deadlock ends respectively. The deadlock status can have values either 0 or 1. If the involved process is selected to abort, then this value is set to 1. All other processes involved in deadlock will have this value set to 0.

DL trace helps us in calculating very important performance metrics directly

<pId, dId, DLStart, DLEnd, Status>

<27, 2, 86, 109, 0>

<34, 4, 258, 475, 1>

.

.

.

<100, 9, 2486, 6971, 0>

Figure 4.1: Sample DL Trace

related to the implemented deadlock algorithm such as Deadlock count, Duration of Deadlock etc.. In the next section, we explain the calculation of these performance metrics in detail.

4.2 Performance Evaluation

The performance metrics related to deadlocks are mainly aimed at measuring the cost (computation and communication) of deadlocks and the techniques to handle them. A deadlock handling technique could be avoidance, prevention or detection followed by recovery. The cost of deadlock detection and recovery depends on the kind of strategy used. We compute performance metrics mainly from two perspectives: (i) From the perspective of an individual process; and (ii) From the perspective of the system. First we will review the metrics used in the distributed systems context, where processes communicate through messages [8, 13, 24, 25, 33, 35].

Prevention of deadlocks is the most desirable option. However, due to the complexity involved, distributed deadlock prevention methods are rarely considered and studied for distributed systems. Maintaining up to date information is almost impossible, due to the absence of a global clock and unpredictable message delays and/or losses. Even with centralized approaches to maintaining the global state with reasonable consistency, the message cost would be prohibitively high, and on most systems such message traffic could easily choke their underlying communication networks. Therefore, prevention is not a popular technique in distributed systems. In any case, the most interesting performance metrics for prevention are the number of times deadlock is predicted and avoided, and the overhead of the algorithm. Next, we look at the metrics suitable for deadlock detection, and resolution techniques in distributed systems.

From a process perspective, the key metrics are: *wait time* during deadlock, *turnaround time* (also sometimes referred to as *response time*), *number of times* involved in deadlocks, and *number of restarts* due to deadlocks. System level performance metrics vary depending on the type of technique employed, whether it is *centralized* or *distributed*. For centralized techniques, the metrics are: *number of deadlocks* that occurred in a specified period of time[1], *number of processes* involved in a deadlock, *number of processes* needed to be aborted to resolve a deadlock, *duration* of a deadlock, *number of processes* completed in a specified period of time, *overhead* of deadlock processing, and *number of messages* used to handle a deadlock. In a distributed case, more than one process could initiate deadlock detection. Therefore, in addition to the above metrics, the number of deadlock detection invocations could be a performance metric when a distributed deadlock handling technique is used. Based on these metrics, several secondary metrics such as the *abort/restart ratio*, *deadlock*

[1]Here the specified period is normally the simulation time.

overhead (time to handle deadlock), *percentage* of processes involved in deadlocks, etc. are also used. Also, for several metrics, *minimum, maximum, average*, and *variance* would be desirable.

In distributed systems, due to message delay and/or loss, and the absence of global knowledge, several kinds of deadlocks such as phantom deadlocks, transient deadlocks, and true deadlocks are possible. These complications are not present in a shared memory system. Multicore systems are typically shared memory systems and therefore deadlock handling in a multicore system does not involve messages. Therefore, counting the number of messages involved in handling a deadlock is not applicable here. Also, we don't deal with the possibility of phantom or transient deadlocks.

A deadlock handling technique in multicore systems could be centralized or distributed. For a centralized implementation, one dedicated process could be assigned to do the job, or the responsibility could be rotated among several processes, but one process is responsible at any particular time. As indicated earlier, in distributed deadlock handling, several processes could initiate deadlock detection independently. If more than one process initiates a deadlock check, then for simplicity and efficiency, one must be elected to do the job. So, in a distributed case, in addition, the cost of an election could be a performance metric. When to initiate a deadlock check also influences the performance.

4.3 List of Performance Metrics

We now summarize the key performance metrics of deadlock handling techniques for multicore systems. We assume that the deadlock handling technique maintains process states (executing, waiting for a resource), resource status (free, used by a process), timestamp of key events (arrival, start of waits, exit, deadlock resolved (prevented), etc.). We refer to the timestamp function as ts.

4.3.1 Performance Metrics (for Process i)

1. *Wait time ($W\_Time_i(R)$)*: The duration between the times i started waiting for resource(s) R and the time the resources R is/are granted access to i. That is, $W\_Time_i(R) = ts(R \text{ granted access to } i) - ts(i's \text{ request for } R)$.

2. *Deadlock wait time ($DW\_Time_i(d_k)$)*: The duration between the times i started waiting due to its involvement in the deadlock d_k and it is resolved. That is, $DW\_Time_i(d_k) = ts(d_k \text{ resolved}) - ts(i \text{ joined } d_k)$.

3. *Turnaround time/response time (TR_Time_i)*: The duration between the times i entered and exited the system. That is, $TR\_Time_i = ts(i's \text{ exit}) - ts(i's \text{ entry})$.

4. *Deadlock count (D_Count_i)*: Number of times process i was involved in deadlock.

5. *Abort/restarts count (AR_Count_i)*: Number of times i is aborted to resolve deadlock.

4.3.2 Performance Metrics (for System)

1. *Deadlock count (D_Count)*: The number of deadlocks occurred in the system in a specified period of time.

2. *System wait time (SW_Time)*: The time processes spend to wait for the requests to be granted and the deadlock d_k to be resolved. That is, $SW\_Time = W\_Time_i(R) + DW\_Time_i(d_k)$, for all i.

3. *Deadlock encounter count (DP_Count)*: The number of times deadlocks are predicted in the system in a specified period of time.

4. *Deadlock size ($D\_Size(d_k)$)*: The number of processes involved in the deadlock d_k.

5. *Deadlock strength ($D\_Strength(d_k)$)*: The number of processes needed to be aborted to resolve the deadlock d_k.

6. *Degree of Deadlock ($DD(d_k)$)*: Degree of deadlock represents the complexity of the deadlock, and we define it as the product of size and strength. That is, $DD(d_k) = D\_Size(d_k) * D\_Strength(d_k)$.

7. *Duration of deadlock ($D\_Duration(d_k)$)*: The duration between the times the deadlock d_k is formed and resolved. That is, $D\_Duration(d_k) = ts(d_k \text{ resolved}) - (dk \text{ formed})$. The value of $ts(d_k \text{ resolved})$ is easy to obtain, and $(dk \text{ formed})$ could be computed from the $ts(request)$ of all the processes involved in the deadlock d_k.

8. *System Throughput (ST)*: The number of processes completed in a specified period of time. The number processes completed per unit time may be computed by dividing it by the entire duration.

9. *Algorithm Overhead ($D\_O(d_k)$)*: The read/write count required to process the deadlock (prevention, avoidance or detection and recovery) d_k.

10. *Total Overhead (SO)*: The total read/write count of system including algorithm overhead to process the deadlock (prevention, avoidance or detection and recovery) d_k.

11. *Overhead Ratio (OHR)*: The ratio between the algorithm overhead to process the deadlock (prevention, avoidance or detection and recovery) d_k to total total overhead in terms of read/write count of the system. That is, $OHR = \frac{Deadlock\ algorithm\ overhead}{Total\ overhead}$

12. *Degree of Concurrent Invocations ($DCCI(d_k)$)*: The number of invocations of deadlock detection of processes involved in the deadlock d_k.

13. *Abort/Restart Ratio (ARR)*: The ratio between the number processes aborted due to deadlock and the total number of processes in the system. That is, $ARR = \frac{Number\ of\ processes\ aborted}{Number\ of\ processes\ in\ the\ system}$.

14. *Deadlock Ratio (DR)*: The ratio of processes involved in deadlocks in comparison to the total number of processes. That is, $DR = \frac{Number\ of\ processes\ involved\ in\ deadlocks}{Number\ of\ processes\ in\ the\ system}$.

Due to the solution technique, current system has $W\_Time_i(R)$, $DW\_Time_i(d_k)$, TR_Time_i, D_Count_i and AR_Count_i implemented for Processes and D_Count, SW_Time, $D\_Size(d_k)$, ST, ARR, SO, DR, $D\_O(d_k)$ and OHR implemented for System. Average, and variance are computed for the metrics related to processes.

4.4 Summary

We discussed different types of traces generated and collected during execution. We also presented different performance metrics and the computation of those metrics from the traces. In the next chapter, we present the analysis of the results from executions for the values of different input parameters. They are analysed and compared based on the performance metrics.

Chapter 5

Simulation Study

We were interested in studying the performances of the Banker's algorithm for deadlock avoidance and Dreadlocks for deadlock detection with process termination as a deadlock recovery, with respect to the metrics discussed in the previous chapter. We present two sets of experiments for Dreadlocks and one set of experiments for the Banker's algorithm, in order to study them and demonstrate the operation of the proposed simulator.

5.1 Experimental Setup

The scenarios used in the experiments are the general scenario and the scenario for the Banker's algorithm. The general scenario entails a normal setup of processes, requests and resources. Request generation is randomized with the upper bound provided. There is no other restriction on the generation of requests. In the scenario for the Banker's algorithm, the processes are assigned a maximum claim for each resource in advance while generating workloads. These scenarios are discussed in detail in Chapter 3.

The first two sets of experiments pertain to Dreadlocks: deadlock detection algo-

rithm. We compare the results of termination types with respect to variation in either cores, or mean arrival rate.

Experiment 1: In this experiment we set the workloads with fixed mean arrival rate and number of cores to vary. Table 5.1 shows the simulation parameters used for this experiment.

Table 5.1: Simulation Parameters for Experiment 1

Parameters	**Value**
Scenario Type	General
Number of Processes	700
Number of Resources	70
Number of Instances per Resource	30
Mean Arrival Rate	2.5
Number of Cores	25, 50, 75, 100, 125

Experiment 2: We set fixed number of cores and mean arrival rate of the workloads is varied in this experiment. Table 5.2 shows the simulation parameters used for this experiment.

Table 5.2: Simulation Parameters for Experiment 2

Parameters	**Value**
Scenario Type	General
Number of Processes	700
Number of Resources	70
Number of Instances per Resource	30
Mean Arrival rate	2.5, 3.5, 5, 6.5, 7.5
Number of cores	75

For both experiments we enabled option of detecting deadlock using Deadlocks. In Chapter 3, we discussed selection criteria for processes to abort. We showed that once deadlock is detected, one or more processes can be selected to abort based on different criteria. They are: Random, Least Recently Used (LRU), First In First Out (FIFO), Minimum Runtime (min. runtime), Maximum Runtime (max. runtime), Minimum Digest (min. digest) and Maximum Digest (max. digest). We believe terminating processes with higher runtimes would be more expensive as restarting such processes cost more. Based on this proposition, we made the following hypothesis.

Hypothesis: Terminating processes with the minimum runtime will perform better[1] than terminating processes with the maximum runtime.

Experiment 3: In this experiment, we show the performance of the Banker's

[1]Here we considered overall performance of the system, that is, a termination type that shows better results for most of the metrics. Due to the randomness of the input and the uncertainty of deadlocks, the selected termination type might not always perform well with all of the performance metrics. Hence, if the termination type performs better for the majority of performance metrics, it is considered better than other termination types.

deadlock avoidance algorithm. The simulation parameters set for this experiment are shown in Table 5.3. We set the workload with a fixed mean arrival rate and varied the number of cores. The scenario type chosen was scenario for the Banker's algorithm.

Table 5.3: Simulation Parameters for Experiment 3

Parameters	**Value**
Scenario Type	Scenario for Banker's algorithm
Number of Processes	500
Number of Resources	100
Number of Instances per Resource	10
Mean Arrival Rate	10
Number of Cores	25, 50, 75, 100, 125

Next, we explain our observations and analysis for deadlock detection: Dreadlocks, deadlock recovery: termination with the minimum and the maximum runtime, and deadlock avoidance: the Banker's algorithm. The observation are illustrated and analyses with respect to key performance metrics for the system.

5.2 Analysis

For the first two experiments, simulation results are mainly shown for four performance metrics in each case. The metrics are: (i)Throughput, (ii) System Abort/Restart Ratio, (iii) Deadlock Wait Time, (iv) Overhead Ratio. For the third experiment, the

performance metrics shown are: (*i*) Turnaround time, (*ii*) Throughput, (*iii*) Process Wait Time and (*iv*) Overhead Ratio. Some of the performance metrics from the initial experiments are not shown for the third experiment because of limitations of the the deadlock handling technique. These metrics are discussed in Chapter 4. Now, we show our observations on the above mentioned performance metrics for each experiment.

We conducted several sets of experiments. In each experiment, we observed a common behavior for termination types with respect to each other. Every time, performances of different termination types with respect to each other resulted in a similar pattern, that is, one type of termination always performed better than other type of termination.

Also, on comparing two different executions, we observed changes in patterns of graphs for some performance metrics, that is, when the performances were compared with respect to the change in the number of cores, the graphs had increasing/decreasing lines and when the performances were compared by changing the mean arrival time, the graphs had lines with a zig-zag pattern. The reason behind this is that the implemented deadlock handling techniques are not developed by keeping change in the mean arrival rate in mind. Hence, their performances with respect to the mean arrival rate are arbitrary and they showed a zig-zag pattern in the results.

5.2.1 Observations from Experiment 1

Observation on Throughput: While conducting experiment with various input, we observe that termination of processes based on the minimum runtime has better throughput than terminating process based on the maximum runtime. With the increase in the number of cores, throughput also increases in both cases. There was always a

significant difference between the throughputs for minimum runtime and maximum runtime. The result for the given input parameters is shown in Figure 5.1.

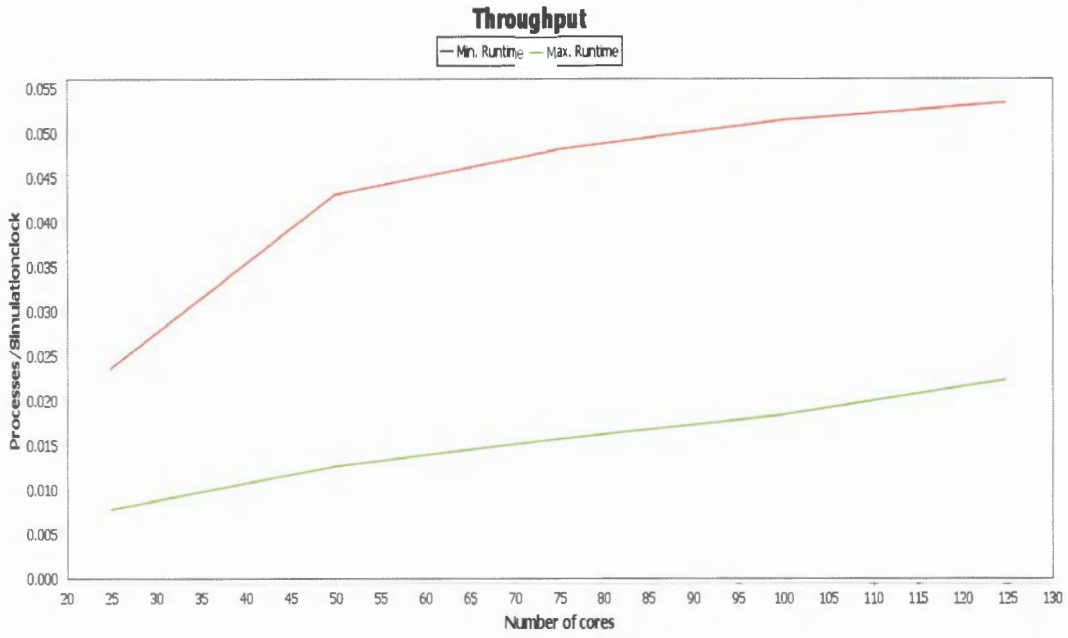

Figure 5.1: Throughput: Minimum Runtime and Maximum Runtime (by varying the number of cores)

The throughput increases with increase in number of cores because with availability of more cores, more processes can execute per unit time. Hence, more processes can complete their executions per unit time.

Observation on System Abort/Restart Ratio (ARR): System ARR increases with the increase in number of cores in both the cases. Termination with maximum runtime always shows better behavior than termination with minimum runtime. The system ARR chart for both termination types is shown in Figure 5.2.

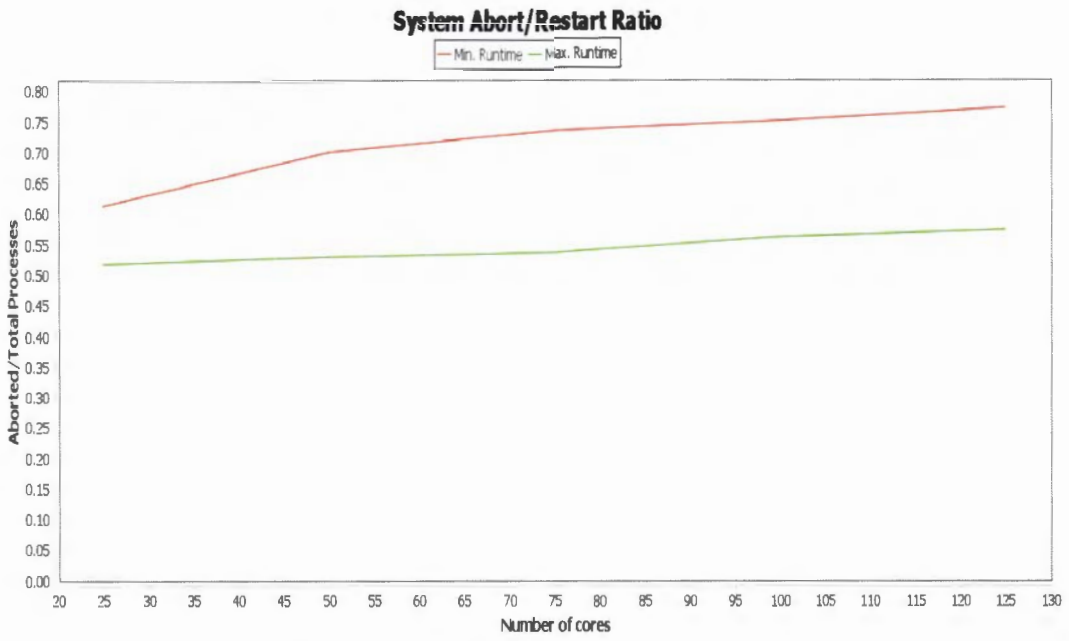

Figure 5.2: System ARR: Minimum Runtime and Maximum Runtime (by varying the number of cores)

The maximum runtime has lower System ARR because less number of processes are terminated when termination criteria is selected as the maximum run time. In the same way, with the minimum runtime, more processes are terminated. Hence, termination with the minimum runtime has a higher System ARR.

Observation on Deadlock Wait Time: The graph for Deadlock wait time for both termination types is shown in Figure 5.3. The Deadlock wait time for termination with the minimum runtime is significantly lower than termination with the maximum runtime. The deadlock wait time increases with an increase in the number of cores for the termination with the maximum runtime, while it remains nearly consistent for termination with the minimum runtime.

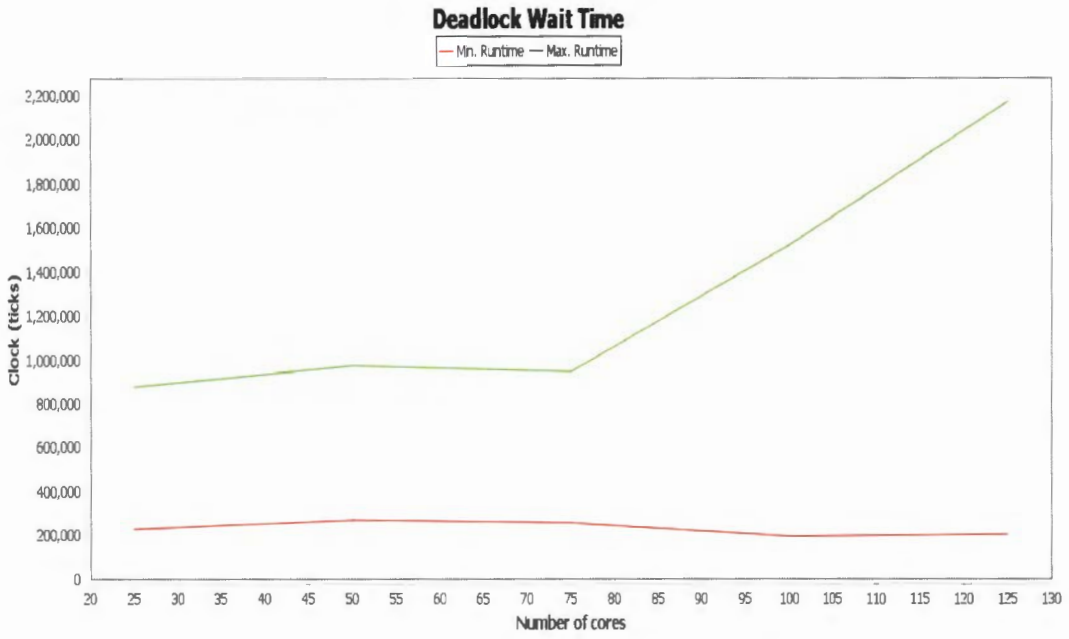

Figure 5.3: Deadlock Wait Time: Minimum Runtime and Maximum Runtime (by varying the number of cores)

With the increase in number of cores, more processes can execute per unit time. This results in more processes holding resources per unit time while waiting for other requested resources. Because of that, more time is required to resolve the deadlock. Hence, the Deadlock wait time increases (for the maximum runtime) or remain nearly consistent (for the minimum runtime) with the increase in number of cores.

Observation on Overhead Ratio: The graph for overhead ratio for both termination types is shown in Figure 5.4. The Overhead ratio decreases with increment in the number of cores. Termination with the minimum runtime always showed a lower overhead ratio than termination with the maximum runtime. For most of the experiments, the overhead ratio varied between 0.02 to 0.4 for Dreadlocks for all termination types.

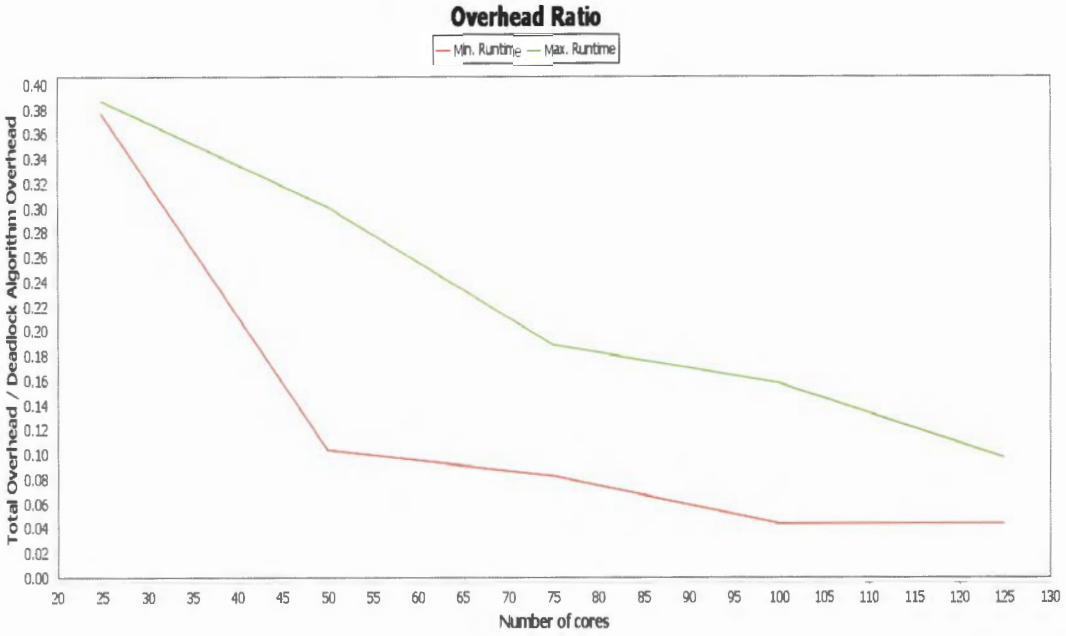

Figure 5.4: Overhead Ratio: Minimum Runtime and Maximum Runtime (by varying the number of cores)

With the decrease in number of cores, less number of processes have to wait for the required resources. Because of that less reads/writes are required to detect the deadlock at every request. Hence, the overhead ratio decreases with the increase in number of cores.

5.2.2 Observations from Experiment 2

Observation on Throughput: We observed that termination based on the minimum runtime always performs better than termination based on the maximum runtime. The graph of throughput for both termination types is shown in Figure 5.5. The graph demonstrates a zig-zag pattern but the values remain between a specific range because the number of cores is same for all the experiments and the implemented

algorithms do not react to the change in the mean arrival rate.

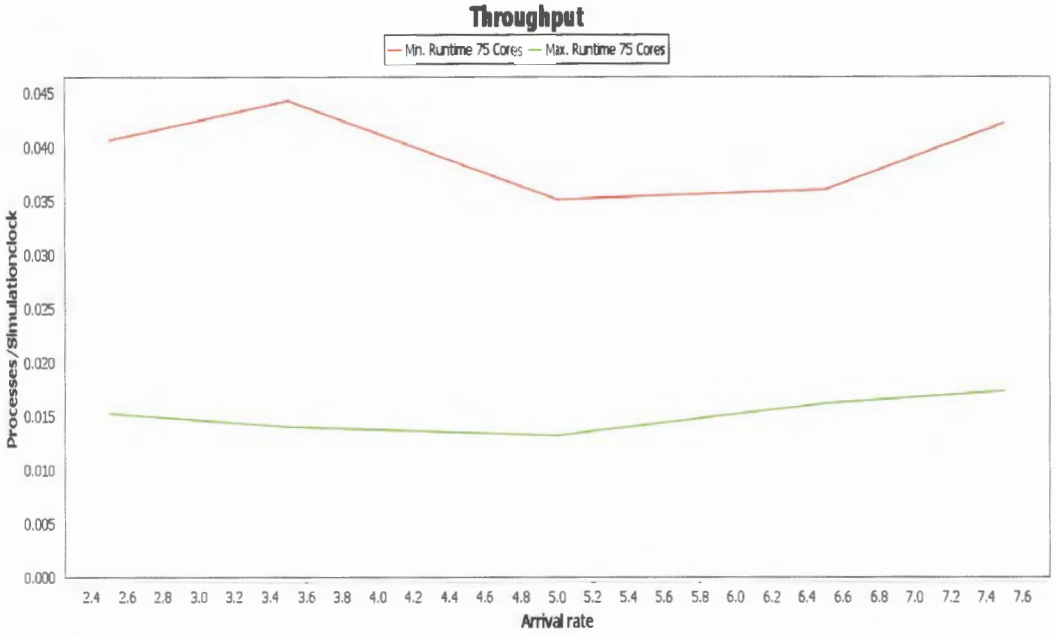

Figure 5.5: Throughput: Minimum Runtime and Maximum Runtime (by varying the mean arrival rate)

Observation on System Abort/Restart Ratio (ARR): For all values of the mean arrival rate, termination with the maximum runtime shows lower value than the minimum runtime. The lines are nearly consistent for all of the values of the mean arrival rate. The system ARR chart for both termination types is shown in Figure 5.6.

Observation on Deadlock Wait Time: The Deadlock wait time for termination with the minimum runtime is always lower than termination with the maximum runtime. The graph demonstrating deadlock wait times for both termination types is shown in Figure 5.7. For most of the experiments of this set, we observed a significant difference between both termination types. Termination with the maximum runtime shows variation while termination with the minimum runtime has almost consistent performance for all values of the mean arrival rate.

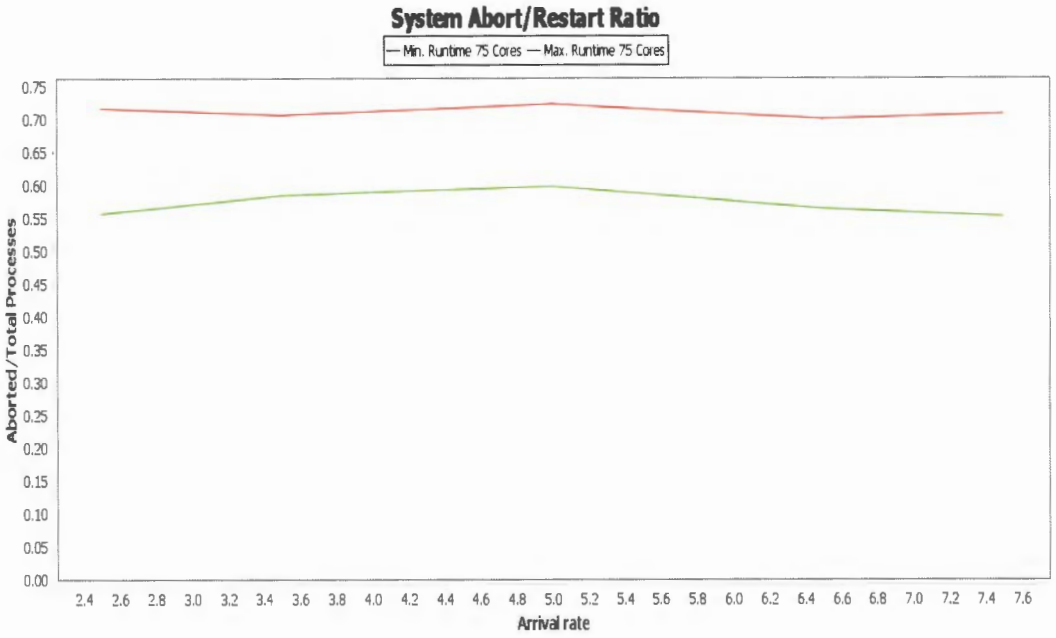

Figure 5.6: System ARR: Minimum Runtime and Maximum Runtime (by varying the mean arrival rate)

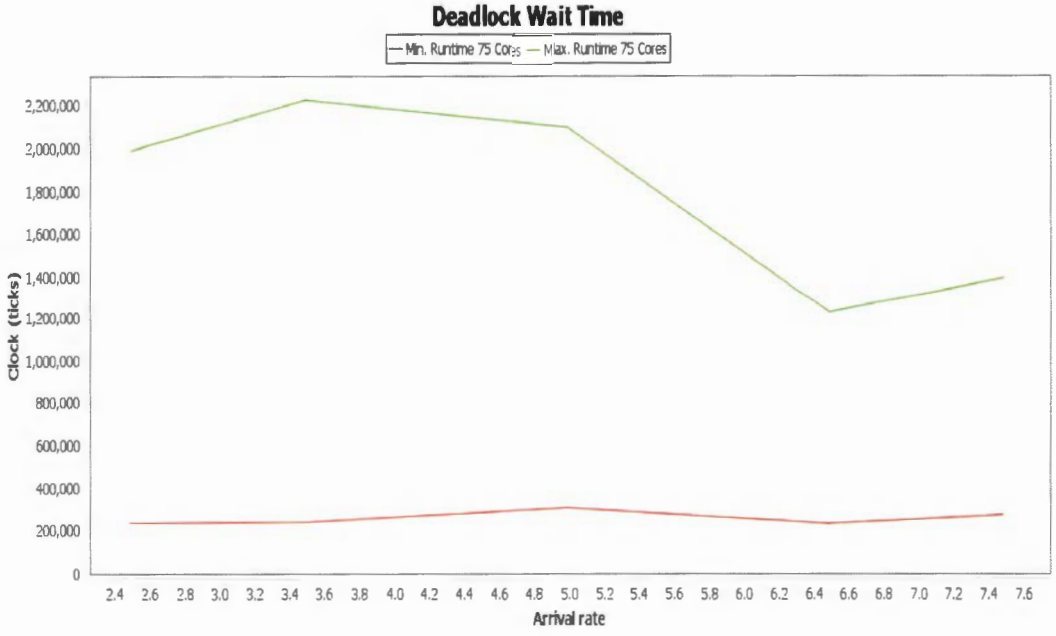

Figure 5.7: Deadlock Wait Time: Minimum Runtime and Maximum Runtime (by varying the mean arrival rate)

Observation on Overhead Ratio: The graph for the overhead ratio for both the termination types is shown in Figure 5.8. The overhead ratio for the termination type with the minimum runtime is always lower than the termination with the maximum runtime. For most of the experiments in this set, the overhead ratio varied between 0.1 to 0.4 for both the termination types.

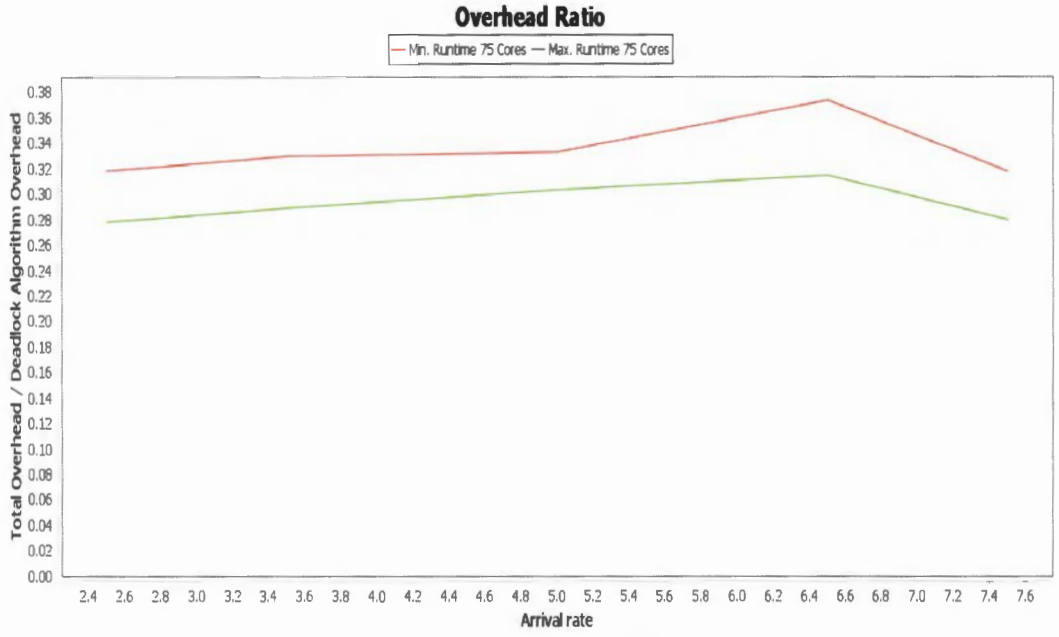

Figure 5.8: Overhead Ratio: Min. Runtime and Max. Runtime (by varying the mean arrival rate)

5.2.3 Observations from Experiment 3

Observation on Turnaround time: We observe that turnaround time decreases with increment in the number of cores for the Banker's algorithm. The rate of decrease of the turnaround time is nearly constant between all the numbers of cores. The graph for turnaround time for the Banker's algorithm is shown in Figure 5.9.

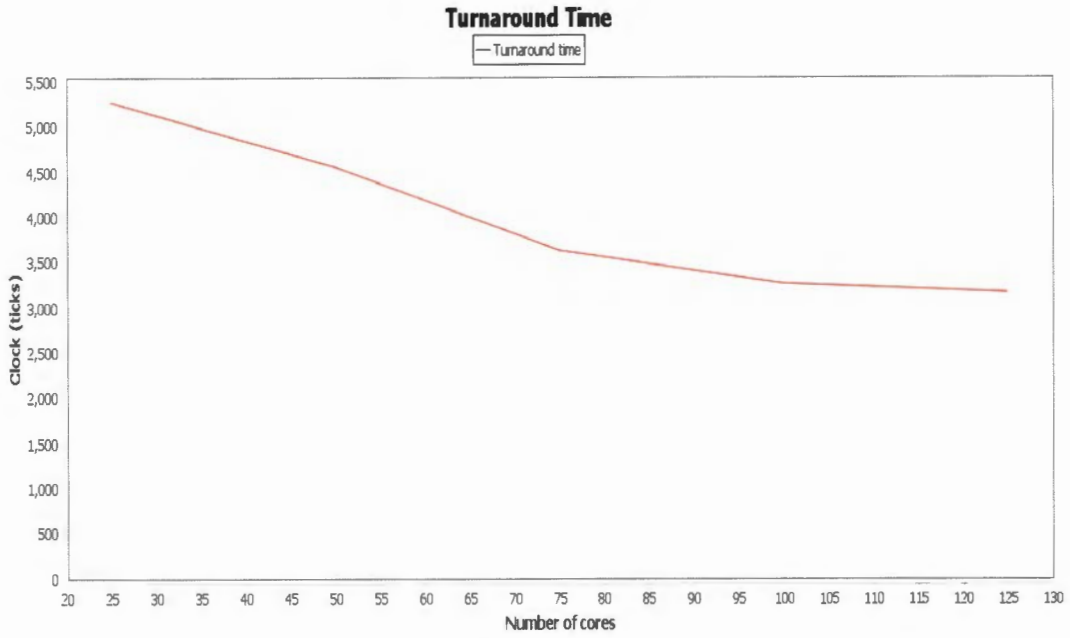

Figure 5.9: Turnaround Time: Banker's Algorithm (by varying the number of cores)

With the increase in number of cores, the processes have more number of cores available per unit time. Which results in less execution time for each process. Hence, the turnaround time decreases with the increase in the number of cores.

Observation on Throughput: We observe that throughput of the system increases with the increase in the number of cores for Banker's algorithm for most of the times. The graph for throughput for the Banker's algorithm is shown in Figure 5.10.

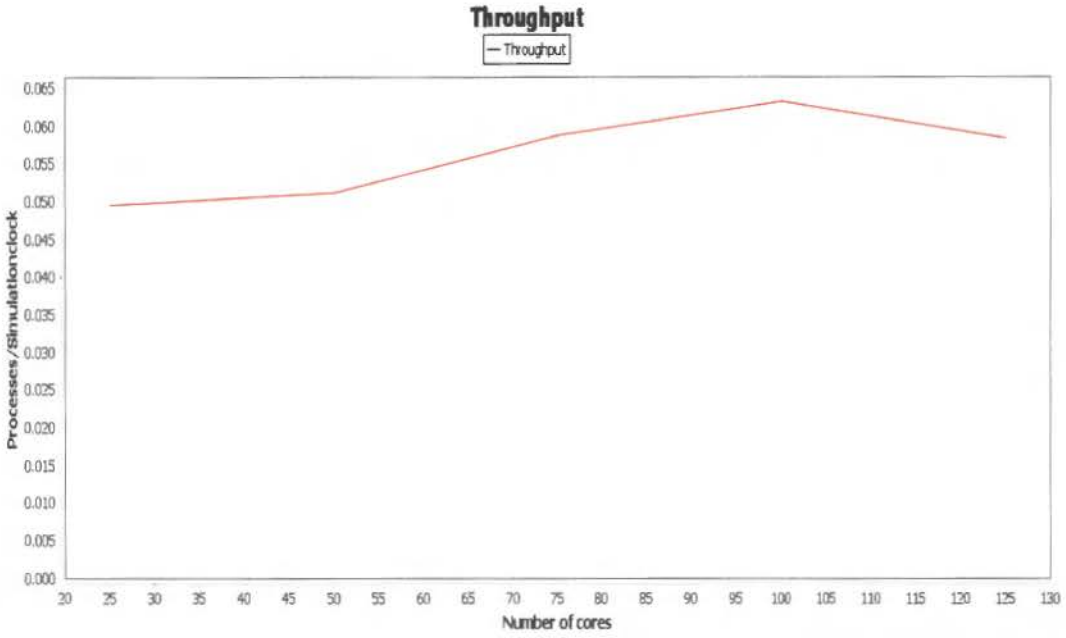

Figure 5.10: Throughput: Banker's Algorithm (by varying the number of cores)

With increase in number of cores, more processes can execute per unit time. Hence, the throughput increases with the increase in number of cores. Though from 100 to 125 cores it slightly decreases, the value of throughput remains higher than other numbers of cores. We can not determine the exact reason behind this strange behavior, randomness of the input can be one of the reasons for that.

Observation on Process Wait Time: The Process wait time decreases with increase in number of cores. We observed consistent results for average and variance for all of the different experiments of this set. The graph for Process wait time for the Banker's algorithm is shown in Figure 5.11.

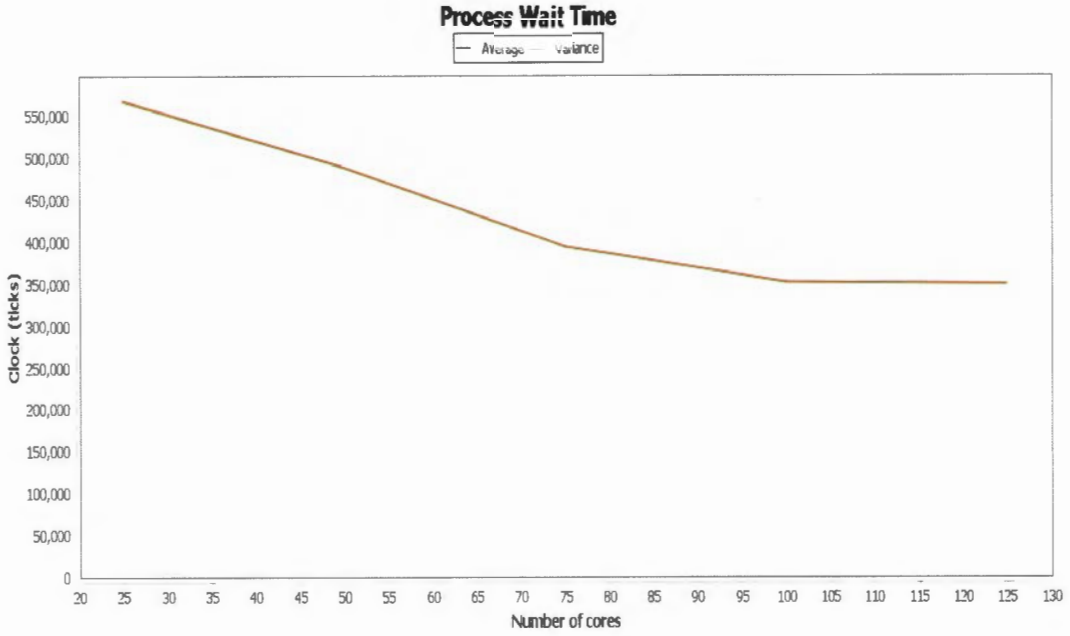

Figure 5.11: Process Wait Time: Banker's Algorithm (by varying the number of cores)

With the increase in number of cores, more processes can execute per unit time. This results in more processes holding resources per unit time while waiting for other requested resources. Because of that, processes have to wait less for other requested resources. Hence, the Process wait time decreases with the increase in number of cores.

Observation on Overhead Ratio: The graph for overhead ratio for Banker's algorithm is shown in Figure 5.12. The overhead ratio increases with increases in the number of cores. The overhead ratio is higher than 0.80 for the given input parameters for all cores. We observed that for all experiments of this set, the overhead ratio for the Banker's algorithm has always been higher than 0.75.

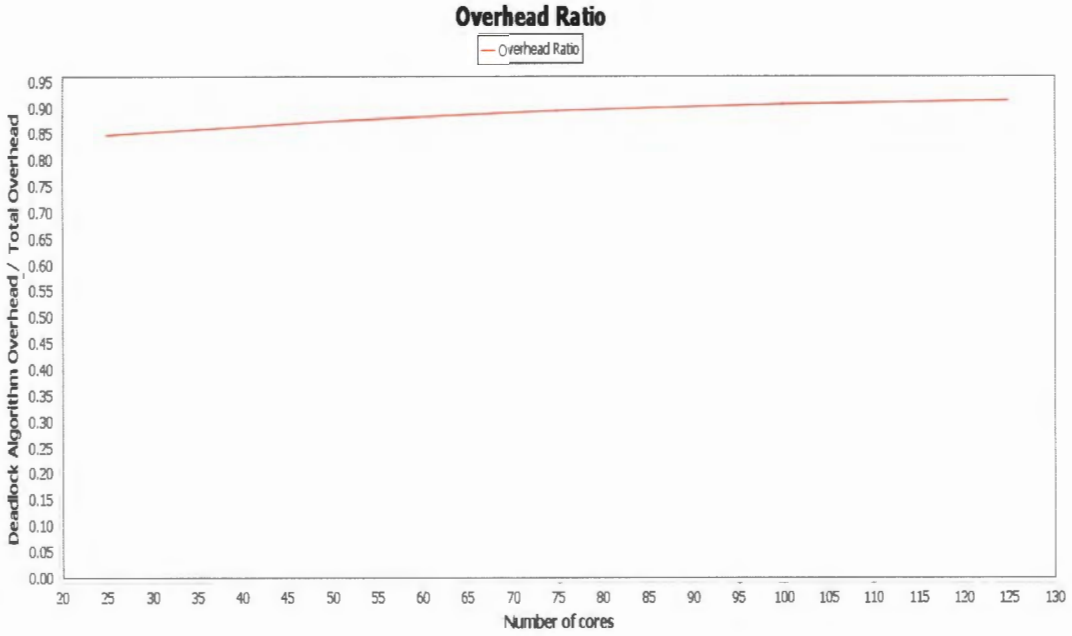

Figure 5.12: Overhead Ratio: Banker's Algorithm (by varying the number of cores)

For the Banker's algorithm, safe state is checked for each request. With the increase in number of cores, more processes execute per unit time. Hence, more processes can request resources per unite time. The algorithm determines a safe sequence while checking the safe state and it assumes that the processes will finish their execution sequentially according to the safe sequence. This idea is not helped by increasing the number of cores. In some cases, the increase in number of cores make the performance worse. Hence, the overhead ratio increases with the increase in number of cores for the Banker's algorithm.

5.3 Summary

We presented the simulation experimentation for termination with the minimum runtime and the maximum runtime, in this chapter. As hypothesized, termination with the minimum runtime outperforms termination with the maximum runtime in most of the cases. We also presented simulation experiments for the Banker's deadlock avoidance algorithm. These experiments showed valid results for deadlock avoidance as well as deadlock detection and recovery. Hence we are confident that our simulation implementation of deadlock in multicore systems is fairly sound.

Chapter 6

Conclusions and Future Work

Deadlock is a very common bug in software applications, yet, it is ignored by most of the operating systems [32]. With the advent of multicore processors, the problem of deadlock has gained renewed attention in research. Many approaches that have been developed to handle deadlock in multicore systems were discussed in Chapter 2. In this chapter, we conclude the thesis with providing possible future work that can be conducted to extend the research carried our in this thesis.

6.1 Conclusion

The primary contributions of this thesis are: (*i*) The design and implementation of a flexible framework to simulate deadlock in multicore systems and (*ii*) Three scenarios implemented to incorporate different deadlock handling techniques. The scenarios are: general scenario, scenario for the Dining Philosopher's problem and scenario for the Banker's algorithm. Two deadlock handling strategies, deadlock avoidance and deadlock detection & recovery, are simulated. The deadlock avoidance strategy is the Banker's algorithm and the deadlock detection strategy is Dreadlocks. Deadlock recovery follows deadlock detection, which is done by terminating one of the processes involved in the deadlock. The experience gained by developing this simulator involves

software design, implementation and performance analysis. (*iii*) Demonstration of the soundness and validity of the proposed framework by presenting the result analysis of the experiments.

The proposed simulator has the basic components required to simulate a deadlock handling algorithm in multicore systems. More deadlock handling techniques can be easily added to this simulator to study, which can furnish initial insights on behavior of the algorithms in practical multicore systems. These insights can really be helpful to study the performances of the algorithms as well as to identify their shortcomings. Based on the shortcomings, necessary guidelines can be offered for improvements of the algorithm.

6.2 Future Directions

The proposed framework to incorporate deadlock handling algorithms in multicore systems is just the beginning of research into deadlock handling algorithms in multicore systems. There are many directions in which the work presented in this thesis can be expanded. We outline some of them next.

- The simulation can be made more sophisticated by refining and improving components such as user interrupts. Current system simulates only I/O interrupts, more real system interrupts can be introduced to the proposed framework.
- Cache memory can also be simulated to the system to make the framework's behaviour closer to real systems.
- More scheduling algorithms can be added to observe performance of deadlock

handling algorithms with them.

- More deadlock handling algorithms can be incorporated.
- Modelling resources and Resource Manager can be refined and improved by adding different types of resources with different properties. Also, function of Resource Manager can be improved to deal differently with various resources.
- The current performance study has exposed some irregularities in the implemented algorithms as well as some limitations of them. For example, for Dreadlocks, digest propagation is not accurate and it considers all the processes with equal priority. These limitations can be overcome to make the algorithms more efficient. For example, to overcome limitation of digest propagation, a centralized schema can be introduces to update digests for all the processes at regular intervals.

I would like to continue working on some of these directions in the future.

Bibliography

[1] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 51–60, New York, NY, USA, 2006. ACM.

[2] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, HVC'05, pages 191–207, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Alex A. Aravind and Viswanathan Manickam. A flexible simulation framework for multicore schedulers: Work in progress paper. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 355–360, New York, NY, USA, 2013. ACM.

[4] Ferenc Belik. An efficient deadlock avoidance technique. *IEEE Trans. Comput.*, 39(7):882–888, July 1990.

[5] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 41–50, New York, NY, USA, 2006. ACM.

[6] Saddek Bensalem and Klaus Havelund. Scalable dynamic deadlock analysis of multithreaded programs. In *IN PARALLEL AND DISTRIBUTED SYSTEMS:*

TESTING AND DEBUGGING (PADTAD - 3), IBM VERIFICATION CONFERENCE, 2005.

[7] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *SIGPLAN Not.*, 37(11):211–230, November 2002.

[8] Omran A. Bukhres. Performance comparisons of distributed deadlock detection algorithms. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 210–217, Washington, DC, USA, 1992. IEEE Computer Society.

[9] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[10] Edsger W. Dijkstra. The origin of concurrent programming. chapter Hierarchical Ordering of Sequential Processes, pages 198–227. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[11] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[12] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, October 2003.

[13] Luca Ferrarini, Luigi Piroddi, and Stefano Allegri. A comparative performance analysis of deadlock avoidance control algorithms for fms. *Journal of Intelligent Manufacturing*, 10(6):569–585, 1999.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[15] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[16] A. N. Habermann. Prevention of system deadlocks. *Commun. ACM*, 12(7):373–ff., July 1969.

[17] Sibsankar Haldar and Alex Aravind. *Operating Systems.* Pearson Education, 2010.

[18] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, January 2006.

[19] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, UK, 2000. Springer-Verlag.

[20] Richard C. Holt. Some deadlock properties of computer systems. In *Proceedings of the Third ACM Symposium on Operating Systems Principles*, SOSP '71, pages 64–71, New York, NY, USA, 1971. ACM.

[21] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, 44(6):110–120, June 2009.

[22] Edgar Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, December 1987.

[23] Eric Koskinen and Maurice Herlihy. Dreadlocks: Efficient deadlock detection. In

Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08, pages 297–303, New York, NY, USA, 2008. ACM.

[24] Natalija Krivokapić, Alfons Kemper, and Ehud Gudes. Deadlock detection in distributed database systems: A new algorithm and a comparative performance analysis. *The VLDB Journal*, 8(2):79–100, October 1999.

[25] Soojung Lee and Junguk L. Kim. Performance analysis of distributed deadlock detection algorithms. *IEEE Trans. on Knowl. and Data Eng.*, 13(4):623–636, July 2001.

[26] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, March 2008.

[27] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Onur Özgün and Yaman Barlas. Discrete vs. continuous simulation: When does it matter. In *Proceedings of the 27th international conference of the system dynamics society*, volume 6, pages 1–22, 2009.

[29] Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. Multicoresdk: A practical and efficient data race detector for real-world applications. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 5:1–5:11, New York, NY, USA, 2009. ACM.

[30] M. Sfinghal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, November 1989.

[31] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[32] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[33] I. Terekhov and T. Camp. Time efficient deadlock resolution algorithms. *Inf. Process. Lett.*, 69(3):149–154, February 1999.

[34] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 602–629, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] Chim-fu Yeung, Sheung-lun Hung, and Kam-yiu Lam. Performance evaluation of a new distributed deadlock detection algorithm. *SIGMOD Rec.*, 23(3):21–26, September 1994.