JUST-IN-TIME AND JUST-IN-PLACE DEADLOCK RESOLUTION

BY FANCONG ZENG

A Dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Prof. Michael L. Littman

and approved by

New Brunswick, New Jersey

May, 2007

ABSTRACT OF THE DISSERTATION

Just-in-time and Just-in-place Deadlock Resolution

by Fancong Zeng

Dissertation Director: Prof. Michael L. Littman

Deadlocked threads cannot make further progress, and frequently tie up resources requested by still other threads, causing more and more threads to come to a standstill. Thus, a deadlock should not remain undetected and uncorrected for a long time. If deadlock-detection processes are run too frequently, however, valuable system resources may be wasted. Therefore, it is important to choose the right interval between successive deadlock detections.

Deadlock recovery must follow deadlock detection to release held resources in the cyclic wait. In addition to restarting the entire system, it is desirable that programmers be able to implement fine-grained recovery actions such as releasing a resource currently not in use. Such fine-grained recovery actions often require the knowledge of program contexts and deadlock states. Unfortunately, modern programming languages lack language-level support for signaling deadlock conditions and for structuring resolution code.

My thesis is that, under the assumption that the time to the first deadlock in the system (after a system restart) follows an exponential distribution, a reinforcement-learning approach is effective in scheduling deadlock detection for a restart-oriented system, and that runtime exceptions are a programming abstraction that allows programmers to write fine-grained deadlock recovery code.

My approach to deadlock-detection scheduling as reinforcement learning estimates the deadlock rate and then performs an optimization to find the detection interval that maximizes system utility. It is theoretically proved that this technique finds the best tradeoff, and experimental results suggest that it is a reasonable approximation to assume that the time to the first deadlock in the system (after a system restart) follows an exponential distribution.

It is natural to consider deadlock occurrences as runtime exceptions because at runtime it is relatively easy to detect actual deadlock occurrences, which represent not only abnormal states but also fatal errors. Thus, exception handlers can be used to resolve deadlock occurrences based on deadlock states and program contexts. Furthermore, because exceptions are a widely used language concept, the technique of deadlock resolution via exceptions is intuitive and practical.

Acknowledgements

I am grateful to Professor Michael L. Littman for his advice and trust. I also would like to thank the other members of my doctoral committee: Prof. Louis Steinberg, Prof. Marvin Paull, and Prof. Gertrude Levine, for their time and efforts.

Dedication

To my wife, Xi Zhu. For her love and support I work and I enjoy.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is deadlock?

Deadlock, or “Deadly Embrace” as it was called by Dijkstra [13], has been widely studied since the mid 1960s. A collection of tasks become deadlocked when they are involved in a cyclic wait for resources. Deadlocks occur in many different applications such as computer systems, communication networks, and databases. The well-known dining philosophers problem [14, 15], introduced by Dijktra, has been widely used to illustrate deadlocks. Levine gave an insightful definition of deadlocks [29].

There are four necessary conditions for deadlock to exist, and they are also sufficient if all resources are unique.

1. “Tasks claim exclusive control of the resources they require (‘mutual exclusion’ condition)” [10].

2. “Tasks hold resources already allocated to them while waiting for additional resources (‘wait for’ condition)” [10].

3. “Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (‘no preemption’ condition)” [10].

4. “A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain (‘circular wait’ condition)” [10].

In this dissertation, I consider deadlocks in centralized systems with reusable and unique resources based on the one-resource deadlock model [25] in which a task can have at most one outstanding request at one time and blocks until the requested resource is granted. As a practical example, a deadlock in a centralized Java system occurs when “two or more threads block each other in a vicious cycle while trying to access synchronization locks needed to continue their activities" [28]. In this example, tasks are threads, and resources are locks.

1.2 A deadlock example in Java

Listing 1.1 shows a simple deadlocked Java program. This program simulates money transfer between accounts. There are two accounts and two threads. For the transfer to begin, each of the two threads has to acquire the locks for both accounts. In the program showed in Listing 1.1, a deadlock occurs when each thread holds one lock after executing "synchronized (Accounts[f])" ands waits for the other lock at "synchronized (Accounts[t])".

Listing 1.1: A simple deadlockable Java program

```
import java.util.Random;
public class Transfer {
  public static final int NumberOfAccounts = 2;
  public static final int InitialFund = 1000;
  public static final int MaxFund = 1000000;
  public static final int NumberOfThreads = 2;
  private static Object[] Accounts =
    new Object[NumberOfAccounts];
  private static long[] balance =
    new long[NumberOfAccounts];
  public static void main(String[] a){
    for (int i = 0; i < NumberOfAccounts; i++){
          balance[i] = InitialFund;
          Accounts[i] = new Object();}
    for (int i = 0; i < NumberOfThreads; i++){
          TransferThread trans = new TransferThread();
          new Thread(trans).start();
  }}
  public static boolean doTransfer(int f, int t, int a){
```

```
  synchronized(Accounts[f]){
    synchronized(Accounts[t]){
      if (balance[f] < a){
          System.out.println(``Transaction Aborted:
           insufficient funds.'');
          return false; }
      if ((balance[t] + a) > MaxFund){
          System.out.println(``Transaction Aborted: too
           much funds.'');
          return false; }
      balance[f] -= a;
      balance[t] += a;
      System.out.println(``Transaction Completed:
       transferred ''+ a + `` dollars from '' + f + `` to
       '' + t);
  }}
  return true;
}
public static class TransferThread implements Runnable{
  public synchronized void run(){
    while (true){
      int fund =
       Math.abs(new Random().nextInt())% InitialFund;
      int source =
       Math.abs(new Random().nextInt())% NumberOfAccounts;
      int dest = 0;
      do {
       dest = Math.abs(new Random().nextInt())%
        NumberOfAccounts;
      }while (dest == source);
```

```
        doTransfer(source, dest, fund);
}}}}
```

1.3 Accomodating deadlocks in production systems

Due to the state-explosion problem, it is inherently difficult to not introduce deadlocks into system design and implementation, Actually, deadlocks are a well-known multithreaded programming fault despite various traditional debugging and testing tools.

These traditional tools require executing the programs under investigation. Recently a few research groups have developed a number of tools trying to help find deadlocks in programs without actually executing these programs. Three examples of such tools are:

1. **ESC/Java** This tool uses a theorem prover to verify that code matches specifications. Generally programmers supply specifications in terms of annotations to the source code. In some cases, programmers do not need to specify annotations and ESC/Java checks some default properties like deadlocks [16].

2. **JLint** This tool operates on bytecode and exploits inter-procedural dataflow analysis and some syntactical checks to find bugs and coding pitfalls. In particular, Jlint builds a lock graph and signals deadlock warnings if there is a cycle in the graph [2].

3. **FindBugs** This tool works at the bytecode level and relies on bug patterns to find bugs. It favors efficient analyses, so it does not use expensive inter-procedural dataflow analyses. Consequently, FindBugs does not report deadlocks effectively [22].

Suppose the three tools are applied to the sample program in Listing 1.1. FindBugs does not have expensive analyses to support static deadlock detection, so it cannot detect the deadlock in the program. On the other hand, JLint and ESC/Java report some warnings for this deadlock.

If the “if (balance[f] $<$ a)” block were moved to the place between the two synchronized statements, there would still be a deadlock problem in the code, but in this case Jlint would not report a warning. This example is an instance of a false negative.

If the doTransfer method were synchronized, then there would not be any deadlock problem. But, in this case, both Jlint and ESC/Java would still report a deadlock warning. This example shows that false positives are also possible.

The above example illustrates that static bug-finding tools may be helpful, but they suffer from false alarms, false negatives, or both, in particular when they are used for large programs. Actually, without the aid of annotations, ESC/Java often produces too many false positives of deadlocks so that by default it does not report deadlock warnings. I turned on the flag to have ESC/Java report deadlock warnings for the sample program to create the example above. Furthermore, there is a huge amount of legacy code that might deadlock and that may not be ready for debugging or even inspection. In addition, at runtime, an application may dynamically load code from the network that may deadlock. So, in practice these tools are also no "silver-bullet" [17] for guaranteeing deadlock-free code.

1.4 Runtime approaches for handling deadlocks

Traditionally speaking, basic runtime approaches for handling deadlocks include prevention, avoidance, and detection and recovery. The approach of negating one or more of the four necessary conditions is referred to as deadlock prevention, unless it aims to avoid deadlocks by exploiting tasks' future resource requirements and ensuring (via runtime testing) that each resource allocation leads to a safe state, in which there remains at least one way for all tasks to accomplish execution. Levine [30] pointed out that "the classification of deadlock prevention and avoidance is erroneous" because deadlock avoidance also negates a necessary condition. Deadlock instances can be detected by checking the wait-for relationship between tasks; after deadlocks are detected, recovery actions are performed to bring the system back to a working state.

1.5 Resource Allocation Graph and Wait-For Graph

A Resource Allocation Graph (RAG), also known as a reusable-resource graph [21], characterizes the runtime relationship between tasks and resources. A RAG's nodes are partitioned into the set of tasks and the set of resources. Edges directed from resource nodes are called

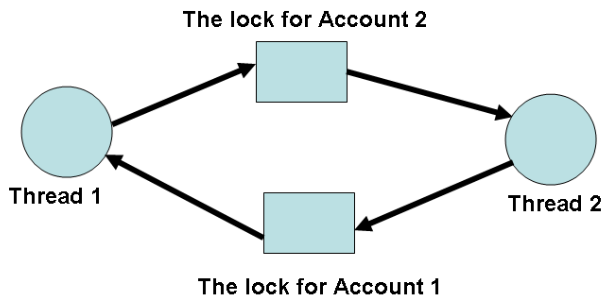

Figure 1.1: A RAG (Resource Allocation Graph) example for 2 deadlocked threads and 2 locks

assignment edges, and edges directed from task nodes are called request edges. Specifically, there is a directed edge from task t_i to resource r_j iff t_i is requesting r_j; there is a directed edge from resource r_m to task t_n iff t_n is holding r_m.

In the case of reusable and unique resources, a RAG can be reduced to a Wait-For Graph (WFG) [1], which describes the wait-for relationship between tasks. Specifically, a WFG is a directed graph, where nodes are tasks, and a directed edge from P to Q, denoted as a wait-for edge, means that P is waiting for a resource currently held by Q. So, the reduction from a RAG to the corresponding WFG in the case of reusable and unique resources is to take out the resource nodes and to collapse the request and assignment edges into wait-for edges. The resulting WFG is always smaller than the RAG. However, the reduction does not lose information *that is needed for deadlock detection*.

Suppose that, in the simple deadlockable Java program as shown in Listing 1.1, Thread 1 and Thread 2 become deadlocked because Thread 1 (resp., Thread 2) holds the lock for Account 1 (resp. Account 2) while waiting for the lock for Account 2 (resp., Account 1). The corresponding RAG is shown in Figure 1.1, and the corresponding WFG is shown in Figure 1.2.

1.6 Deadlock detection via cycle checking

In the case of reusable and unique resources, a cycle in the WFG is both sufficient and necessary for a deadlock assuming the other three conditions (“no preemption”, “mutual exclusion”,

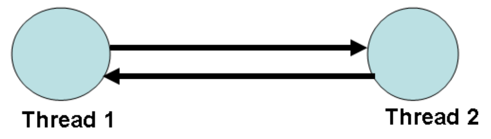

Figure 1.2: A WFG (Wait-For Graph) example for 2 deadlocked threads and 2 locks

and “wait for”) are operative. Deadlock detection in this dissertation work is performed by dynamically building a WFG (based on the runtime relationship between resources and tasks in the program) and checking for cycles in the WFG.

It is not a new idea to find deadlocks by checking for cycles in the WFG. Actually, more than twenty years ago, Agrawal et al. [1] and Chin [8] showed that, for the deadlock model in this dissertation, the complexity of deadlock detection via cycle checking in the WFG is $O(n)$, where n is the number of tasks in the current system, no matter whether detection is continuous or periodic.

Continuous detection and periodic detection are two flavors of deadlock detection scheduling. In continuous detection, deadlocks are checked whenever an edge is added to the WFG graph. In periodic detection, deadlocks are checked periodically either due to some timer timeout or after a certain number of edges are added to the WFG graph.

The contribution of this dissertation is not a new deadlock-detection algorithm. Rather, this dissertation is focused on two emerging research topics in deadlock detection and recovery. As described in the next 2 sections, one is *scheduling deadlock detection to maximize system performability*; the other is *providing a programming abstraction for programmers to resolve deadlocks*.

1.7 Just-in-time deadlock detection

Deadlock detection is associated with a performance overhead. If deadlock detection is invoked too often, the overall detection overhead may significantly impact the normal system performance. On the other hand, if the interval between two consecutive deadlock-detection invocations is too large, then a potential deadlock occurrence left undetected for a long time also may hurt the system performance dramatically.

Reinforcement learning is “learning what to do–how to map situations to actions–so as to maximize a numerical reward signal” [42]. In other words, reinforcement learning allows a software agent to keep learning and adjusting its behavior based on feedback from the environment as time goes by in order to maximizes some well-defined reward. No human domain expert is really needed in this automated learning scheme.

Thus, in order to maximize the “performability” (performance and reliability) [36] of long-running server applications, it is a nice fit to cast the optimal deadlock-detection frequency problem as reinforcement learning.

As stated elsewhere [24], in a standard reinforcement-learning model, a learning agent interacts with its environment via action and perception. The model consists of

1. a set of environment states S;
2. a set of actions A;
3. a set of scalar reinforcement signals.
4. an input function I describing how the agent views the environment state.

Assuming total observability, I is the identity function. At each time t, the learning agent perceives its state s_t in S and chooses an action in $A(s_t)$. It will receive a reward r_{t+1} and perceive a new state s_{t+1}. Based on these interactions, the learning agent must develop a policy P, which maps states to actions to maximize some long-term measure of rewards.

In the deadlock-detection-scheduling setting, since the deadlock time itself is not observable to the agent, I would not be the identity function. Planning and learning in such partial observable domains is notoriously difficult [23]. In Chapter 3, I detail how to establish a parametric utility-centric model for deadlock detection and recovery and how to solve the model to maximize the expected utility.

1.8 Just-in-place deadlock recovery

Once a deadlock is detected, one or more of the four necessary conditions have to be invalidated in order to resolve the deadlock. Recovery actions include killing an offending task, preempting a resource, releasing a resource currently not in use, rolling back to a checkpoint, and even simply restarting the entire system, among many others. It often depends on program semantics and runtime states to determine a fine-grained recovery action that best resolves the current deadlock. Moreover, it is important to enable programmers to implement the (fine-grained) recovery actions they have picked up and to incorporate the implementation into their programs effortlessly.

Goodenough [19] stated that exceptions and exception handling are needed “in general as a means of conveniently interleaving actions belonging to different levels of abstractions.” In programming languages, exceptions are features that “provide the programmer the capability to specify what should happen when unusual execution conditions occur, albeit infrequently” [41].

Because deadlocks are not only abnormal events but also rare yet fatal errors, it is natural to consider deadlocks as exceptions and to exploit exception handling to resolve deadlocks. Furthermore, because exceptions are a widely used language concept, the technique of deadlock resolution via exceptions is intuitive (to learn) to use and is appropriate for real-life large programs. In the dissertation, runtime exceptions are defined and implemented to help programmers resolve deadlocks [51]. In Chapter 4, I describe how to define, implement and use deadlock exceptions.

1.9 Outline

The rest of the dissertation is organized as follows: Chapter 2 discusses and compares related work. Chapter 3 details the approach of formulating deadlock-detection scheduling as reinforcement learning. Chapter 4 presents and discusses the approach of deadlock resolution via exceptions. Chapter 5 concludes the dissertation. Chapter 6 includes as the Appendices the code listings used for the dissertation.

Chapter 2

Related Work

In this chapter, I compare and discuss related work in various areas.

2.1 Unspecified failure rate

More than 40 years ago, Barlow et al. [40] initially presented the problem of finding optimal inspection policies that minimize the expected cost until a failure is detected. Since then, a few researchers have devised new models based on different assumptions and/or object functions, assumed a specification of the failure distribution shapes and parameters, and proposed various approximation algorithms [34, 39, 12].

I assume that the time to the first deadlock follows an exponential distribution, but I do not require a specification of the deadlock rate. Rather, in this dissertation, I discuss an on-line reinforcement-learning algorithm that keeps learning the deadlock rate and calculating the detection intervals so as to maximize the system performability.

Performability is “performance and reliability” [36]. Reinforcement learning is “learning what to do–how to map situations to actions–so as to maximize a numerical reward signal” [42]. In other words, reinforcement learning allows a software agent to keep learning and adjusting its behavior based on feedback from the environment as time goes by in order to maximize some well-defined reward. Thus, in order to maximize the performability of long-running server applications, it is a nice fit to formulate the optimal deadlock-detection-interval problem as a reinforcement-learning problem.

2.2 Learning and estimation techniques

The deadlock-detection-scheduling problem is a special case of a continuous-time partially observable Markov decision process (POMDP) problem. While it is known that discrete-state POMDP problems are difficult or impossible to solve in the worst case [35], computational approaches have been proposed and applied [23]. The even-more-challenging continuous-time POMDPs have received almost no attention from computationalists at all.

Q-learning [45] is a general model-free reinforcement-learning algorithm. It does not directly handle the partial observability problem, which is a key element of the deadlock-detection scheduling problem: The system may have already deadlocked but until the detection is performed, this information is not available to the decision maker.

Bayesian inference [3] is also used for parameter estimation in reliability analysis. When using Bayesian inference, people need to define the prior distribution of the parameter to be estimated and often have to transform the original estimation problem into a simpler one in order to avoid complex computations. Without exploiting Bayesian inference, I directly solve the formulated problem without assuming any range of the concrete distribution.

2.3 Deadlock-detection scheduling

Chen [7] performed a Petri-net-based analysis of deadlock-detection scheduling in centralized translation database systems with dynamic locking. Specifically, Chen compared periodic detection with continuous detection, and reported: 1) there existed an optimal deadlock detection time interval for performance maximization; 2) the optimal deadlock-detection interval was a function of a few parameters such as workloads, transaction sizes and locking policies; and 3) periodic detection was better than continuous detection when deadlocks are rare, although the performance improvement was often small.

Ling et al. [32] studied scheduling distributed deadlock detection, and assumed that “deadlock formation follows a Poisson process” without performing empirical studies. They aimed to “schedule deadlock detections so as to minimize the long-run mean average cost of deadlock handling”, and they devised some formulae relating the deadlock-formation rate and the detection-scheduling frequency.

Java currently does not support continuous deadlock detection. My work is focused on periodic deadlock-detection scheduling, and I try to maximize a system's average productive time. In particular, I discuss the impact of an undetected deadlock on system productivity rather than the number and/or sizes of deadlocks that affect deadlock resolution.

In practice, two (distributed) deadlock occurrences are likely to be related. Thus, I use a more realistic assumption that the time to the *first* deadlock follows an exponential distribution. Furthermore, it is difficult to know the deadlock formation rate beforehand in practical applications. So, I propose a reinforcement-learning algorithm that continuously estimates the exponential distribution rate (λ) and calculates the scheduling frequencies accordingly.

In addition to performing a simulation study in which experimental data was generated that closely fits my assumption, I have applied my work in a Java experiment with a simple yet sufficiently realistic sample application. The experiment not only validated that it is a reasonable approximation that the time to the *first* deadlock follows an exponential distribution but showed that the algorithm has low overhead and can adjust the detection interval for better system performance in response to the system deadlock behaviors.

If deadlock rates change over time, my approach can be used keeping only the most recent data. Because it learns so quickly, such an approach will remain accurate.

2.4 Complementary techniques to deadlock exceptions

Williams, Thies, and Ernst [47] exploited static analyses to find Java library deadlocks. Despite the false positives reported by the static analyses, I believe such static analyses [47] help programmers use deadlock exceptions by letting them focus on program points that may deadlock.

Ilc et al. exploited a roll-back mechanism that allows locks to be transparently preempted from Java threads [46]. They use this mechanism to avoid the priority-inversion problem. Priority-inversion happens when a low-priority task holds a resource required by a high-priority task. In my work, I separate mechanisms from policies: user-defined exception handlers are used to resolve deadlocks, signaled as exceptions. On the other hand, it would be interesting to investigate using exceptions to resolve priority inversion.

2.5 Extended application scope

A practical solution to the deadlock-detection-scheduling problem has an extended application scope. Failure detection is a key element to success in the emergent "self-healing" tools and systems area [26]. My approach can be adapted for detecting failures whose distribution can be approximated by an exponential distribution.

Deadlock resolution via system restarting is investigated in this dissertation. System restarting has been used in practice to work around Heinsenbugs [20] and to reclaim stale resources like leaked memory. Recently, researchers have been looking into building recursively-restartable systems [4] and optimizing restart strategies [44]. Thus, knowing when to restart is becoming a core problem in several systems areas.

Checkpointing, a technique for periodically saving enough information so that a task can be started from the last point at which information was saved, has been widely used to avoid restarting a task from the beginning [6]. A few references in the literature [33, 18, 49] discuss optimal checkpoint placement. To the best of my knowledge, they all assume that failures such as deadlocks are detected as soon as they occur. My work can be adapted to this work to remove that assumption.

Different polling policies were studied [9] in order to keep "fresh" local copies of remote data sources for web search engines. A Poisson process was proposed as the change model of a data source, and experimental data was used to support the proposal. The learning algorithm in this paper fits well into learning the λ's for the Poisson processes, thus potentially making search engines more responsive.

In summary, a number of related investigations have looked at finding optimal detection/inspection policies and deadlock recovery techniques, but two critical problems remain unsolved. Ons is finding the optimal deadlock-detection interval without knowing the deadlock rate beforehand assuming the system is restarted as soon as a deadlock is detected; the other is providing language-level abstractions for authorizing and structuring fine-grained deadlock resolution code. These dissertation seeks a solution to these two problems.

Chapter 3

Deadlock-Detection Scheduling as Reinforcement Learning

3.1 Overview

In today's programming practice, multithreaded programming is error prone. Deadlocks are a well-known multithreaded programming fault. Moreover, due to the state-explosion problem, it is essentially hard to produce deadlock-free code only. Thus, it is not uncommon for deadlocks to occur in production systems.

Deadlocked tasks not only cannot make further progress, but also frequently tie up resources requested by still more threads, causing more and more tasks to come to a standstill. Thus, a deadlock should not remain undetected and uncorrected for a long time. However, deadlock detection is associated with performance overhead. If deadlock detection is performed too frequently, valuable system resources may be wasted.

Therefore, it is important to choose the right interval between successive deadlock detections. This chapter [52] provides a decision-theoretic learning approach to scheduling deadlock detection.

Specifically, I learn a utility-based model for deadlock occurrence, and solve the model to maximize the expected utility. The detection interval in the solution depends on the deadlock rate, which is normally not in the system specifications. However, I provide a learning algorithm for estimating the deadlock rate. Thus, the deadlock-detection scheduling approach includes an effective method for figuring out the unknown deadlock rate and applies it within an automated procedure for obtaining the current optimal detection interval.

The rest of the chapter is organized as follows. Section 3.2 formulates the problem of deadlock-detection scheduling. Section 3.3 discusses the issue of reward maximization. Section 3.4 details the procedure for estimating the deadlock rate. Section 3.5 presents and discusses an online algorithm for determining the current optimal detection frequency. Section 3.6 uses a simulation to investigate the convergence behavior of the algorithm. Section 3.7 reports my empirical findings by applying the algorithm to detect deadlocks in a sample Java application. Section 3.8 concludes this chapter

3.2 Problem formulation

System restarting has been used in practice to work around Heinsenbugs [20] and to reclaim stale resources like leaked memory. Recently, researchers have been looking into building recursively-restartable systems [4] and optimizing restart strategies [44]. In this chapter, I focus on exploiting restarting to resolve deadlocks. In today's programming practice, system restarting is the only working solution for resolving Java deadlocks involving monitor locks—Chapter 4 will discuss programming abstractions that enable other deadlock resolution solutions.

A deadlock detection is associated with cost D, and a system restart is associated with cost R. When the first deadlock is detected, the system is restarted. In this chapter, I consider the initial system start as the first restart.

I assume that the time to the first deadlock follows an exponential distribution; the Java experiment results in Section 3.7 have been consistent with this standard statistical modeling assumption. In the rest of the chapter, I use λ to denote the hazard function (for the exponential distribution), that is, the deadlock rate.

I assume deadlocks do not happen during a deadlock detection, and I define the time interval w that the system waits until the next deadlock detection as the detection interval. In practice, a system administrator may place both a lower bound and an upper bound on w. The lower bound w_l constrains the biggest fraction of cycles that can be dedicated to deadlock detection, and the upper bound w_u constrains the worst case of how long a deadlock can go undetected. I assume that any w within $[w_l, w_u]$ does not essentially affect the deadlock rate λ, the system-restart cost R, or the deadlock-detection cost D. Both R and D are defined in terms of time units.

I formulate the problem in a reinforcement-learning setting [42] as one of learning to make decisions to maximize a reward. I consider that the system keeps receiving $+1$ reward (for doing useful work) when it is deadlock free and it is not doing a deadlock detection or system restart. Because, for any time point t_0 and interval t, the probability that the system starting at t_0 is deadlock free until after time point $t_0 + t$ is $e^{-\lambda t}$, the reward that the system receives during the detection interval w is:

$r(w) = \int_{t_0}^{t_0+w} e^{-\lambda(t-t_0)} dt = (1 - e^{-\lambda w})/\lambda.$

If a deadlock is detected, then a system restart is performed automatically. The probability of a system deadlock within time interval w is $1-e^{-\lambda w}$, thus the average reward that the system receives is:

$a(w) = r(w)/(w + D + (1 - e^{-\lambda w})R).$

I am trying to optimize the average performance within these bounds. Thus, the problem that I want to solve is: *Choose a detection interval w^* to maximize $a(w^*)$ subject to $0 < w_l \leq w^* \leq w_u$ where w_l and w_u are constants* potentially defined by a system administrator. I call this problem the *Utility-Maximization Problem*.

Note that the deadlock-occurrence time point itself is not observable to the agent, making the problem akin to a partially observable Markov decision process [23] in that decisions have to be made without complete state information. Planning and learning in such domains are notoriously difficult, but I show that the modeling assumptions provide sufficient structure for creating a practical learning algorithm.

3.3 Utility maximization

The critical parameters describing the system are the delay imposed by deadlock detection D, the deadlock recovery time R, and the deadlock rate λ. In this section, I show how to select a detection interval w given D, R, and λ that maximizes the average utility $a(w)$. I assume $\lambda > 0$ and $w > 0$.

The derivative of $a(w)$ is $a'(w) = h(w)g(w)$ where $h(w) = e^{-\lambda w}(w + D) - (1 - e^{-\lambda w})/\lambda$ and $g(w) = 1/(w + D + (1 - e^{-\lambda w})R)^2$. Note that $\forall w > 0, g(w) > 0$.

The derivative of $h(w)$ is $h'(w) = -\lambda e^{-\lambda w}(w + D)$. Because $\forall w > 0, h'(w) < 0$, $h(w)$ is a strictly decreasing function. Since $\lim_{w \to 0} h(w) = D > 0$ and $\lim_{w \to +\infty} h(w) = -1/\lambda$, $h(w)$ has one and only root, denoted as *root*. Since $a'(w) = 0$ iff $h(w) = 0$, *root* is a maximizer of $a(w)$. Thus, *root* is the solution to the *Utility-Maximization Problem* if $w_l \leq$ *root* $\leq w_u$. If $w_u <$ *root* (resp., $w_l >$ *root*), then w_u (resp., w_l) is the solution to the *Utility-Maximization Problem*.

Note that $h(w)$ does not depend on R, meaning that the system restart cost (which is not affected by w) has no impact on the optimal choice of w. Thus, the optimal w only depends on λ and D.

To apply this optimization, it is important to have an accurate model—values of D and λ. It is straightforward to calculate the average value of D via experience with the running system by simply averaging the times needed for deadlock detection. However, there is need to estimate λ, a method for which is described next.

3.4 Lambda estimation

The problem of λ estimation would be simple if we could know when the deadlock occurred. In particular, we could average the time to first deadlock and take the reciprocal.

In fact, we only observe whether the system has deadlocked or not by particular time points—those for which deadlock detection was run. As such, the deadlock-occurrence time is only partially observable.

However, it can be estimated in a maximum likelihood sense, described next.

Consider a series of detection intervals $w_1, w_2, \ldots, w_i, \ldots$. The probability that the system fails within an interval w_i is $1 - e^{-\lambda w_i}, \forall i > 0$.

For $\lambda > 0$, define a component of the log-likelihood function as:

$$f_i(\lambda) = \begin{cases} -\lambda w_i & \text{if no deadlock in } w_i \\ \log(1 - e^{-\lambda w_i}) & \text{otherwise.} \end{cases}$$

The derivative of $f_i(\lambda)$ is:

$$f_i'(\lambda) = \begin{cases} -w_i & \text{if no deadlock in } w_i \\ e^{-\lambda w_i} w_i / (1 - e^{-\lambda w_i}) & \text{otherwise.} \end{cases}$$

The second derivative of $f_i(\lambda)$ is:

$$f_i''(\lambda) = \begin{cases} 0 & \text{if no deadlock in } w_i \\ -e^{-\lambda w_i} w_i^2/(1-e^{-\lambda w_i})^2 & \text{otherwise.} \end{cases}$$

The log-likelihood function of the probability of the observed data is $l(\lambda) = \sum_i f_i(\lambda)$. As is common in machine-learning applications, I seek the maximum log-likelihood solution [5]. That is, I am interested in the value of λ that makes the observed deadlocks maximally likely.

If there is an interval in which the system deadlocks, and if there is another interval in which the system does not deadlock, then $l''(\lambda) < 0, \lim_{\lambda \to 0} l'(\lambda) = +\infty, \lim_{\lambda \to +\infty} l'(\lambda) < 0, \forall \lambda > 0$. So, in this case, $l'(\lambda)$ has one and only one root, which is the maximizer of $l(\lambda)$ and the maximum likelihood estimator (MLE) for λ.

These calculations can be used to select a series of detection intervals by performing deadlock detection at the end of each interval, and if a deadlock is detected, a system restart is also performed. If, so far, there is an interval during which a deadlock is detected and an interval during which no deadlock is detected, the MLE for λ is obtained by numerically finding the root of $l'(\lambda)$. If no deadlock has been detected so far, $\lambda = 0$. If deadlock is detected at the end of every interval seen so far, $\lambda = +\infty$.

For time and space efficiency, for a long-running system, an online learning algorithm cannot always use all detection intervals since the first system start. In particular, a numerical method to solve $l'(\lambda)$ has to compute, during every iteration, $e^{-\lambda w_i} w_i/(1 - e^{-\lambda w_i})$ for every deadlocked interval w_i. A deadlocked interval is a detection interval during which a deadlock occurs. I describe in the next section a practical online learning algorithm.

3.5 An online learning algorithm

The overall algorithm *ALG*(w_0, w_l, w_u, k) for estimating λ and computing the optimal detection interval is as follows:

1. Initialize w to some value w_0 between w_l and w_u.

2. After waiting for w time units, perform deadlock detection.

3. Update D, the average deadlock-detection cost over all detections so far during the last k restarts.

4. If no deadlocks have been detected so far, set $w = \min(2w, w_u)$ and go to Step 2.

5. If every deadlock detection so far reports a deadlock, set $w = \max(w/2, w_l)$ and go to Step 10.

6. Use the detection intervals during the last k restarts to find the MLE for λ by numerically finding the root of $l'(\lambda) = \sum_i f_i'(\lambda)$, where $f_i'(\lambda)$ is defined in Section 3.4.

7. Set w to the root of $h(w) = e^{-\lambda w}(w + D) - (1 - e^{-\lambda w})/\lambda$ as defined in Section 3.3. The average deadlock detection cost D and the MLE for λ are used to numerically find the root of $h(w)$.

8. If $w < w_l$, set w to w_l.

9. If $w > w_u$, set w to w_u.

10. If a deadlock is detected, perform a system restart.

11. Go to Step 2.

The optimal w depends on λ and D as does the dynamics of the learning process. Note that no explicit exploration is performed, nor is any needed. That is, *any* w used for deadlock detection provides information about the true value of λ, due to the assumption of exponentially distributed times to the first deadlocks.

The parameter k determines an upper bound of the number of detection intervals used for λ estimation in practice. However, with $k = *$ meaning using all detection intervals thus far, I show that the algorithm *ALG*$(w_0, w_l, w_u, *)$ will converge on optimal behavior in the limit.

Proposition Let λ_i (resp. w_i) denote the sequence of λs (resp. ws) computed by the algorithm. The value of w_i converges to w^*, which is the solution to the *Utility-Maximization Problem*. Moreover, if $w^* \neq w_l$ and $w^* \neq w_u$, the value of λ_i converges to λ^*, which is the true constant hazard function.

Proof sketch: Suppose w_i does not converge to w_l or w_u. Since $0 < w_l \leq w \leq w_u$, we can break up the values from w_l to w_u into δ-sized blocks for any $\delta > 0$. For the block from w to $w + \delta$, look at the set of j such that $w \leq w_j < w + \delta$. If this set is finite, it will not have an effect on the converged value of λ_i . If it is infinite, then the fraction of deadlock-free trials—no deadlock is detected within the corresponding intervals—will be between $e^{-\lambda^* w}$ and $e^{-\lambda^*(w+\delta)}$. As $\delta > 0$ was arbitrary, this fraction approaches $e^{-\lambda^* w}$, for which λ^* becomes the MLE. So, no matter what the sequence of detection intervals is, the procedure will estimate the constant hazard function as λ^* in the limit. Since w^* is the optimal detection interval for λ^*, the procedure will converge to this choice of detection interval as well.

3.6 A simulation study

In this section, I show that the algorithm quickly finds near-optimal detection intervals in a simulation study, and that the initial value w_0 has little impact on the convergence to near-optimal detection intervals.

3.6.1 Theoretical optimal values

I generated 500 deadlocks according to an exponential distribution of $\lambda = 1E{-}6$ for the simulation study. Assuming each time unit is one second, deadlocks occur exponentially with a mean of 277.8 hours (after each restart). Other parameters are listed in Table 3.1.

D	R	w_l	w_u	w_0	k
30	300	120	10800	600	1000

Table 3.1: Parameters for the simulation study

Given 500 deadlocks in total in the simulation study, parameter $k = 1000$ means that, every time the algorithm estimates a λ, it uses all detection intervals so far (since the beginning of the simulation study).

Given the true $\lambda = 1E{-}6$ and other parameters, the theoretical optimal detection interval $w^* = 7736$, which corresponds to 2.15 hours. The theoretical peak average reward $a(w^*) = 99.2\%$, where $a(w)$ is defined in Section 3.2. The value of R does not influence w^*, but it affects $a(w^*)$. If R were 300000 in the simulation study, for example, $a(w^*)$ would have been 76.47%.

The average detection cost D includes both the average cost of a single deadlock detection and the average cost of a single execution of the algorithm. The average cost of a single execution of the algorithm is considered to be the deadlock-detection scheduler overhead. A constant D means that the variance of the algorithm overhead is ignored; I will discuss the algorithm overhead in detail in Section 3.7.

I define the optimality ratio as $p(w) = a(w)/a(w^*)$, and take $p(w) \geq 99.95\%$ as the definition of near-optimality for w.

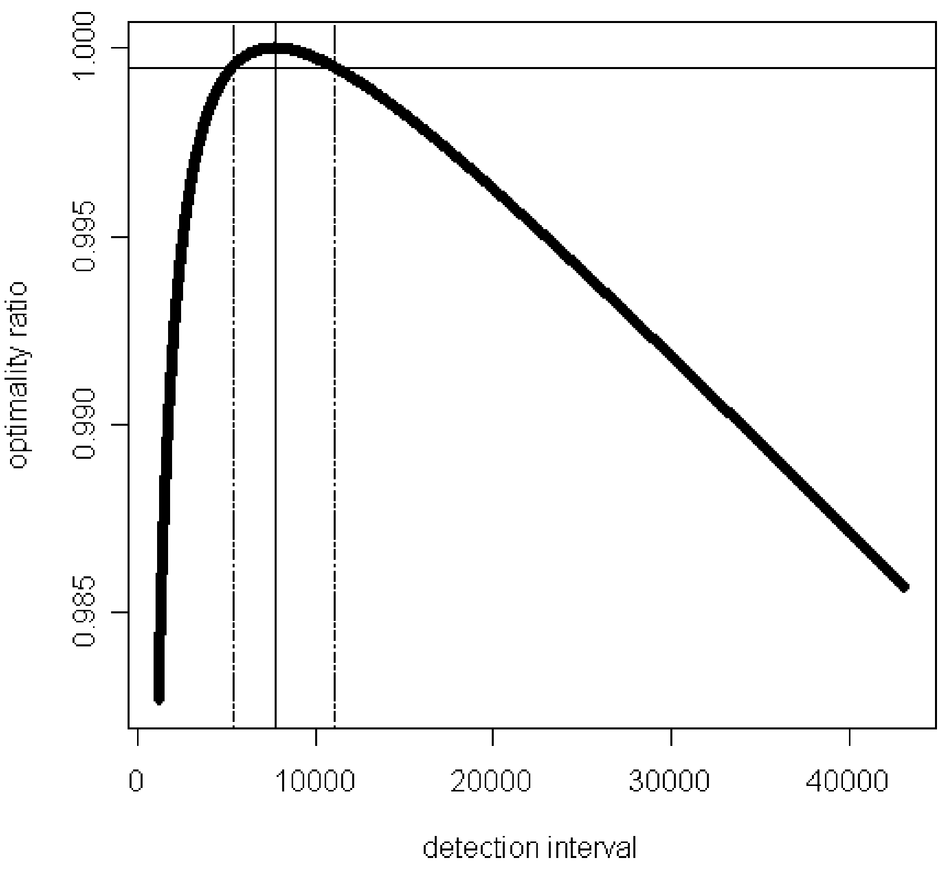

Figure 3.1: Detection interval versus optimality ratio for the simulation study

Figure 3.1, an inverted U-shaped curve, shows the relationship between the detection interval w (from 1200 to 43200) and the optimality ratio $p(w)$.

If $w = 1200$, simulating a detection interval of 2 hours, the optimality ratio is 98.26%. If $w = 43200$, simulating a detection interval of 12 hours, the optimality ratio is 98.56%. If w is between 5406 and 11069, the optimality ratio is over 99.95%.

3.6.2 Lambdas and detection intervals

Table 3.2 shows the simulation results after several deadlocks. The field *Deadlocks* and *Detections* in the table represent cumulative data from the beginning of the simulation. Values λ and w in each row are the λ and detection intervals estimated/calculated right after the corresponding number of *Deadlocks* have occurred.

Deadlocks	Detections	λ	w
1	7	2.897E-05	1429
3	287	2.083E-06	5356
4	535	1.303E-06	6776
9	860	1.759E-06	5830
10	894	1.882E-06	5636
25	3206	1.232E-06	6969
50	6188	1.194E-06	7078
100	12372	1.151E-06	7211
250	30957	1.121E-06	7307
500	64739	1.060E-06	7515

Table 3.2: Simulation data after varying numbers of deadlocks

After 500 deadlocks have occurred, 64739 detections have been performed. The first detection interval $w_0 = 600$ is given/calculated before any detection, and then an interval is calculated after each detection. So, there are 64740 calculated detection intervals in total.

Figure 3.2 shows the dynamics of the calculated detection intervals. As consistent with mathematical analysis, the figure shows that the calculated detection interval size keeps increasing in the absence of a deadlock and that, once a deadlock occurs, the next calculated detection interval size decreases, often noticeably.

After 293 detections have been performed, all calculated detection-intervals are near optimal. If the initial w_0 was $w_l = 120$ (resp., $w_u = 10800$), I find by additional simulations that there would be 64741 (resp., 64730) detections. However, the estimated λ after 500 deadlocks would still be 1.060E-06, the calculated w after 500 deadlocks would still be 7515, and all calculated intervals after 4 deadlocks would be near optimal. In general, a different initial value w_0 makes a small change to the number of detections, but it does not have significant impact on the estimated λ and calculated w in the long run.

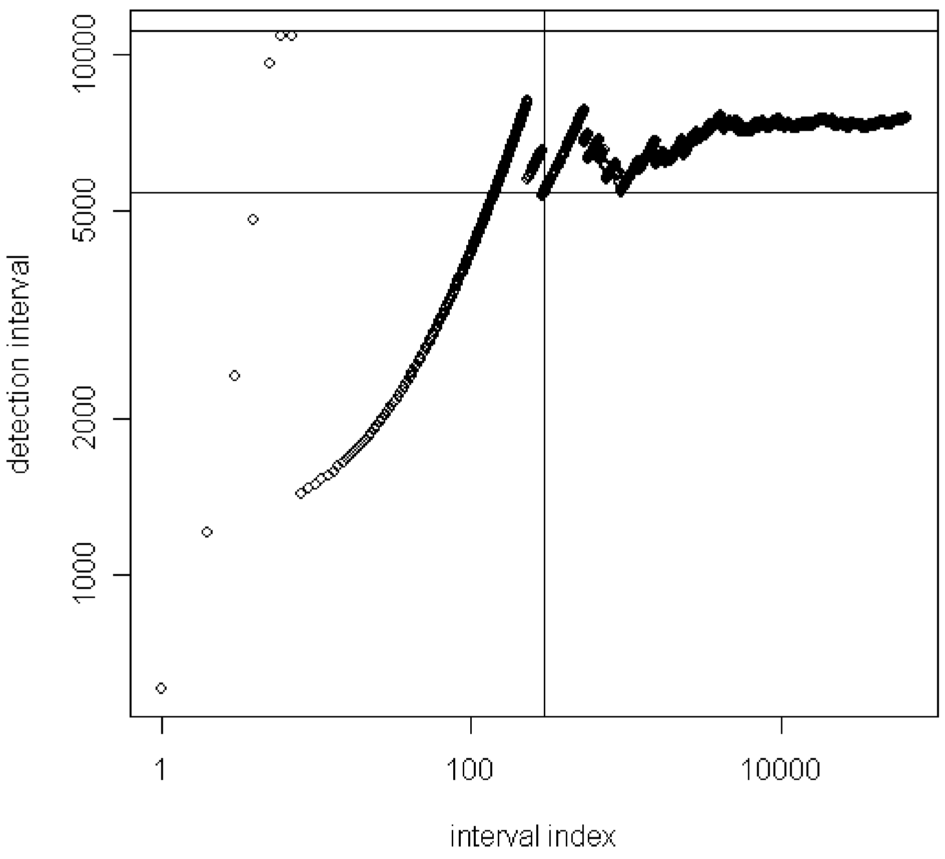

Figure 3.2: Learning curve for the simulation study (Log-Scaled Axes)

Interestingly, the sequence of λ and w (after 4 deadlocks have occurred) does not maintain a relative error of no more than 0.05%. After 500 deadlocks have occurred, $\lambda = 1.060E-06$ has a relative error of 6%, and $w = 7515$ has a relative error of 3%. Recall that the true $\lambda = 1E{-}6$ and the theoretical optimal detection interval $w^* = 7736$. So, although the simulation study shows that it takes quite a few failures to estimate λ for an exponential distribution—a finding consistent with [38]—the average reward $a(w)$ is at a low level of insensitivity to variations in λ and w.

Therefore, the simulation study suggests that, under the assumption of an exponential distribution of the time to first deadlocks, the algorithm can find near-optimal ws quickly in terms of the number of detections in the presence of a few deadlocks.

In next section, I show 1) The online algorithm has an insignificant overhead with a reasonable k; and 2) The online algorithm can compute tiny λs.

3.7 A Java Experiment

I have implemented the algorithm described in Section 3.5 using J2SE 5.0. The implementation contains fewer than 1000 lines of code. I have integrated the implementation with an example Java application, and performed a case study of applying the algorithm with $k = 100$ to Java deadlock detection in a real multithreaded environment.

3.7.1 About Java deadlocks

For software components implemented in Java, a deadlock occurs when "two or more threads block each other in a vicious cycle while trying to access synchronization locks needed to continue their activities" [28]. In such a deadlock case, tasks are threads and resources are reusable locks. Before Java 5.0, Java only provided monitor locks. Since Java 5.0, Java has provided a package "java.util.concurrent.locks" in addition to monitor deadlocks. I focus the discussion of this section on monitor locks only. Thus, in the rest of this chapter, a lock means a monitor lock in Java. Java deadlocks involving only monitor locks can be detected by a Java API: findMonitorDeadlockedThreads. In J2SE 5.0, the Java Doc for findMonitorDeadlockedThreads in ThreadMXBean says: "It might be an expensive operation"[1]. In any case, all Java threads should be stalled when deadlock detection is in progress. So, if deadlock detection is performed too frequently, valuable system resources may be wasted.

Deadlocked threads not only cannot make further progress, but also frequently tie up resources requested by still more threads, causing more and more threads to come to a standstill. Thus, a deadlock should not remain undetected and uncorrected for a long time. Java's approach for handling deadlocks is deadlock detection and recovery. J2SE 1.4.1 has introduced a command-line deadlock-detection utility, and J2SE 5.0 has provided thread-management beans

[1]http://java.sun.com/j2se/1.5.0/docs/api/.

to facilitate writing customized deadlock-detection utilities. Once deadlocks are detected, recovery actions are often required. Java currently does not support fine-grained deadlock recovery actions such as killing an offending thread; the API to kill a thread, Thread.stop(), is now deprecated and does not always function properly. A working solution to Java deadlock recovery is to restart the Java Virtual Machine (JVM).

3.7.2 Experiment setup

The machine used to perform the experiment had 2.00 GB of RAM and one 2.00 GHz processor. The operating system was Windows XP Professional, SP2. The JDK was J2SE 5.0 update 6.

The example application used in the experiment defines two classes: class Account and class Experiment. The code listing for a simplified version of the two classes is in the Appendices (Section 6.1 and Section 6.2) at the end of the dissertation. Class Account has a "transfer" method to transfer money from one account to the other. In some cases, both accounts need to be locked for the transfer to be accomplished. When several threads are executing the "transfer" method, it is possible that two threads become deadlocked when trying to lock the destination account while holding the lock of the source account. Class Experiment defines 2 accounts and 4 threads. The run method of each of the 4 threads executes a loop with its iteration transferring some randomly selected amount of money between two different accounts, which are also randomly selected.

Deadlock detection and recovery is performed by the main thread, which has the highest thread priority. Once a deadlock is found, deadlock recovery via system restart is performed to keep the experiment running until 500 restarts have occurred.

3.7.3 Experiment parameters and data

w_l	w_u	w_0	k
0.1 s	30 min	15 s	100

Table 3.3: Parameters for the Java experiment

Deadlocks	R (ms)	D (ms)	S (ns)
10	102.6	13.2	162162
25	236.6	17.7	208827
50	121.3	15.3	307462
100	133.4	15.6	520522
150	118.5	17.3	832319
200	127.4	18.0	930881
250	162.3	18.7	922265
300	149.3	19.0	918374
400	178.8	17.5	910223
500	219.5	18.1	912333

Table 3.4: Detection and recovery costs from the Java experiment

Deadlocks	Detections	λ	w (ns)
1	3	1.412E-11	3.217E9
2	49	1.030E-11	2.021E9
3	50	1.549E-11	1.636E9
4	62	1.889E-11	1.411E9
5	73	2.217E-11	1.232E9
6	82	2.548E-11	1.163E9
7	88	2.906E-11	1.077E9
8	100	3.159E-11	1.076E9
9	101	3.560E-11	1.009E9
10	204	2.786E-11	9.697E8
11	273	2.550E-11	1.071E9
25	893	2.188E-11	1.267E9
50	1772	2.275E-11	1.157E9
100	3692	2.276E-11	1.166E9
200	7305	2.325E-11	1.239E9
250	8968	2.515E-11	1.213E9
300	10519	2.573E-11	1.210E9
400	14215	2.272E-11	1.235E9
500	17352	2.753E-11	1.140E9

Table 3.5: Experimental data after varying numbers of deadlocks

Table 3.3 lists the parameters used by the algorithm in the experiment. It took about 6 hours to finish a run of the experiment. Table 3.4 reports the detection and recovery costs from the experiment.

The average cost of deadlock detection (D) includes two items: One is the average cost of invoking a Java deadlock-detection API, and the other is the average overhead of the algorithm for computing λ and w. The average algorithm overhead is also denoted as the average deadlock-detection scheduler overhead (S). Note that S is part of D. The computational cost in this experiment is measured in terms of nanoseconds.

In the experiment, D (resp., S) is the average deadlock-detection cost (resp., deadlock-detection scheduler cost) over all detections so far in the last $k = 100$ restarts. The experiment finds a sequence of small average deadlock-detection cost D. It is not surprising that the cost of a single detection is small, because the system uses only 4 fixed Java working threads competing for 2 Java locks. As shown in Table 3.4, S keeps increasing when the number of deadlocks is no more than 200. When the number of deadlocks is more than 200, S is around 1 ms.

The average scheduler overhead S is not sensitive to the number of threads or locks, and it is still relatively small compared to the single detection cost. Moreover, in this experiment and the simulation study in Section 3.6, a generic bisection method was used to find the numerical roots required by Step 6 and Step 7 of the algorithm; the efficiency of the scheduler could be further improved with a more customized numerical method.

The restart cost R in this experiment is also small. It includes the cost to restart a Java Virtual Machine (JVM) and the cost to save and fetch a small amount of data—the algorithm needs some data, whose size is bounded by k, and the rest of the data is for keeping a record of the experiment execution. There is no checkpoint for the experiment application. Again, R does not impact the optimal choice of w.

Table 3.5 shows the experiment results after several deadlocks. The field *Deadlocks* and *Detections* in the table represent cumulative data from the beginning of the experiment. Values λ and w in each row are the λ and detection intervals estimated/calculated right after the corresponding number of *Deadlocks* have occurred.

As the computational cost is measured in terms of nanoseconds, the algorithm computes tiny λ's. The experiments involved 17352 detections and ended up with λ = 2.753E-11 and

w=1.140 second.

In the next section, I present a practical approach to evaluating the estimated λ's and detection intervals.

3.7.4 A practical evaluation

After each restart, the online algorithm keeps waiting and then detecting until the first deadlock occurs. For the ith restart, I recorded the start time-point $s(i)$ and the end time-point $e(i)$ of the deadlocked interval, that is, the detection interval in which the first deadlock had occurred.

Suppose the algorithm spent detection cost $c(i)$ before the deadlocked interval during the ith restart, the lower bound of the ith productive time period $p(i)$ is $x(i) = s(i) - c(i)$ and the upper bound of productive time is $y(i) = e(i) - c(i)$. The ith deadlocked interval size is $y(i) - x(i) = e(i) - s(i)$.

Due to the JVM thread-scheduling overhead and the Java timer API invocation overhead, the recorded deadlocked interval size is a few milliseconds larger than the corresponding detection-interval size calculated by the algorithm.

There are 500 deadlocked intervals; 499 of them are below 3 seconds. As shown in Table 3.6, 90% of the deadlocked intervals are below 1247.9 ms. The largest deadlocked interval (60006.3 ms) belongs to the first productive time period.

Table 3.6 also shows that the lower bounds on the productive time period are broader and larger than the deadlocked interval sizes. For some productive time periods, the deadlocked interval is the first detection interval, thus the corresponding lower bounds are 0.

As a consequence of the exponential distribution assumption of the time to the first deadlock, I assume the exact time-point in which the first deadlock occurred during the ith system lifetime follows a uniform distribution on the interval ($s(i)$,$e(i)$]. For the ith restart, the average productive time period is $\bar{p}(i) = (x(i) + y(i))/2$, and the average total lifetime period, assuming a constant detection interval of w time units, is $\bar{k}(w, i) = R + n(D + w)$, where the number of detections $n = (\int_{x(i)}^{y(i)} \lceil z/w \rceil dz)/(y(i) - x(i)) = w((2y(i)/w - \lceil y(i)/w \rceil + 1)\lceil y(i)/w \rceil - (2x(i)/w - \lceil x(i)/w \rceil + 1)\lceil x(i)/w \rceil)/(2y(i) - 2x(i))$.

Define $A(w, m, n) = \sum_{i=m}^{n} \bar{p}_i / \sum_{i=m}^{n} \bar{k}(w, i)$ to be an empirical estimate of the average reward using the productive time periods. According to $A(w,m,n)$, once again, R affects the maximal value of $A(w,m,n)$, but does not affect the value of its maximizer.

Define $A(w^*(m,n),m,n)$ to be the peak average reward. For $A(w,1,500)$ using the average $R = 194.3$ ms (over the 500 restarts) and the last $D = 18.1$ ms, the peak average reward $A(w^*(1,500),1,500) = 96.65\%$ for $w^*(1,500) = 1244$ ms.

Define $P(w,m,n) = A(w,m,n)/A(w^*(m,n),m,n)$ to be the estimated optimality ratio. Figure 3.3, another inverted-U-shaped curve very similar in shape to Figure 3.1, shows the relationship between the detection interval (from 100 ms to 10000 ms) and the optimality ratio.

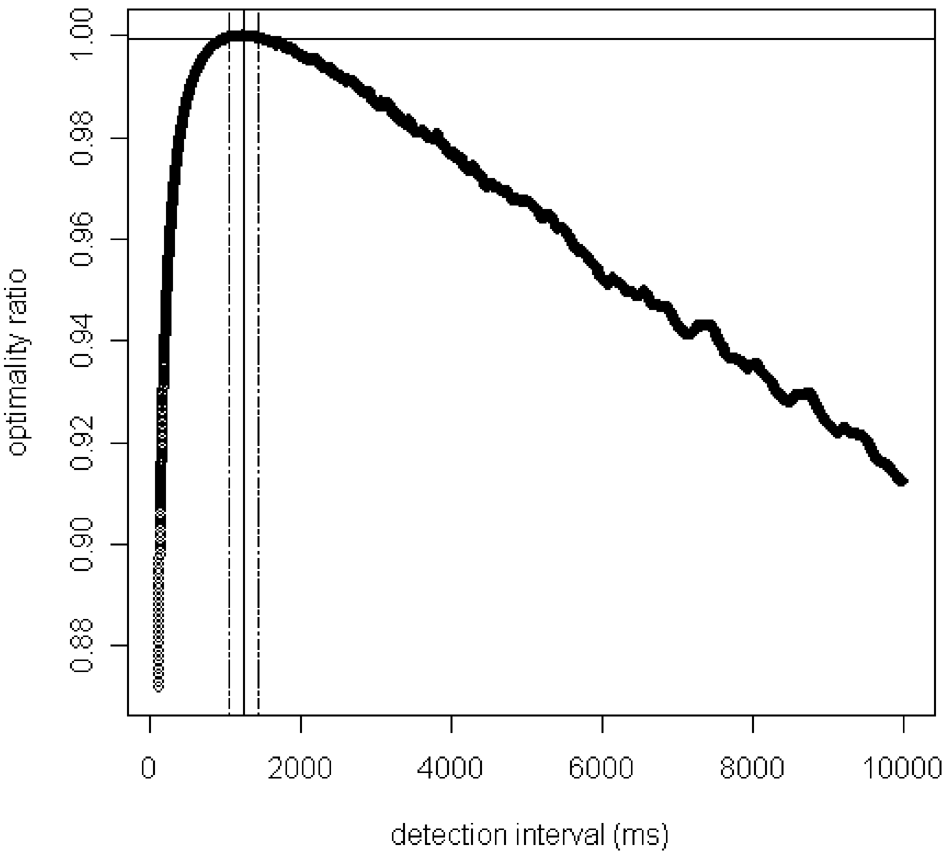

Figure 3.3: Detection interval versus optimality ratio for the Java experiment

I take $P(w,m,n) \geq 99.95\%$ as the definition of near optimality for w. If w is *not* in [1009,1455], then $P(w,1,500) < 99.95\%$ and w is therefore not near optimal. On the other hand, w in [1027,1420] is near optimal because $P(w, 1, 500) \geq 99.95\%$ for w in [1027,1420]. In fact, the final calculated interval is 1140 ms.

3.7.5 The dynamics of calculated detection intervals

I use the last 250 productive time period ranges to test all the 8968 detection intervals used during the first 250 restarts. For $A(w, 251, 500)$ with the average $R = 194.3$ ms (over the 500 restarts) and the last $D = 18.1$ ms, the peak average reward $A(w^*(251, 500), 251, 500) = 96.58\%$ for $w^*(251, 500) = 1152$ ms.

If w is *not* in [997,1454], then $P(w, 251, 500) < 99.95\%$. If w is in [1024,1403], then $P(w, 251, 500) \geq 99.95\%$.

Like Figure 3.2, Figure 3.4 shows that the size of the detection interval calculated right after a deadlock occurrence drops compared to the previous detection interval. After the average detection cost D has been stabilized, detection interval sizes generally increase in the absence of a deadlock occurrence. It takes a few detections to stabilize the average detection cost D in the experiment.

The experimental study uses an algorithm instance with $k = 100$. More generally, if k is $100 \times j$ where j is a positive integer, S will be bounded by $j \times q$ where q is a constant in terms of time units. In this experiment setting, according to Table 3.4, q would be around 1 ms. A larger k, the average scheduler overhead, would use more detection intervals for learning, thus it will estimate λ and w with less variance.

However, as consistent with the simulation study, the experiment suggests that the algorithm has an insignificant overhead and can find near-optimal w's quickly in terms of the number of detections in the presence of a few deadlocks. Figure 3.4 shows that for $i \geq 232, P(w_i, 251, 500) \geq 99.95\%$. That is, after *232* detections, all detection intervals calculated by an algorithm instance with $k = 100$, which has an insignificant overhead, are near optimal.

I take $P(w, m, n) \geq 99.95\%$ as the definition of near optimality for w; in practice, a system administrator can redefine near optimality as needed. It is also worthwhile to note the following use scenario. In practice, for the purpose of load balance and fault tolerance, there are often multiple server-application instances running similar code and balancing workload in a cluster.

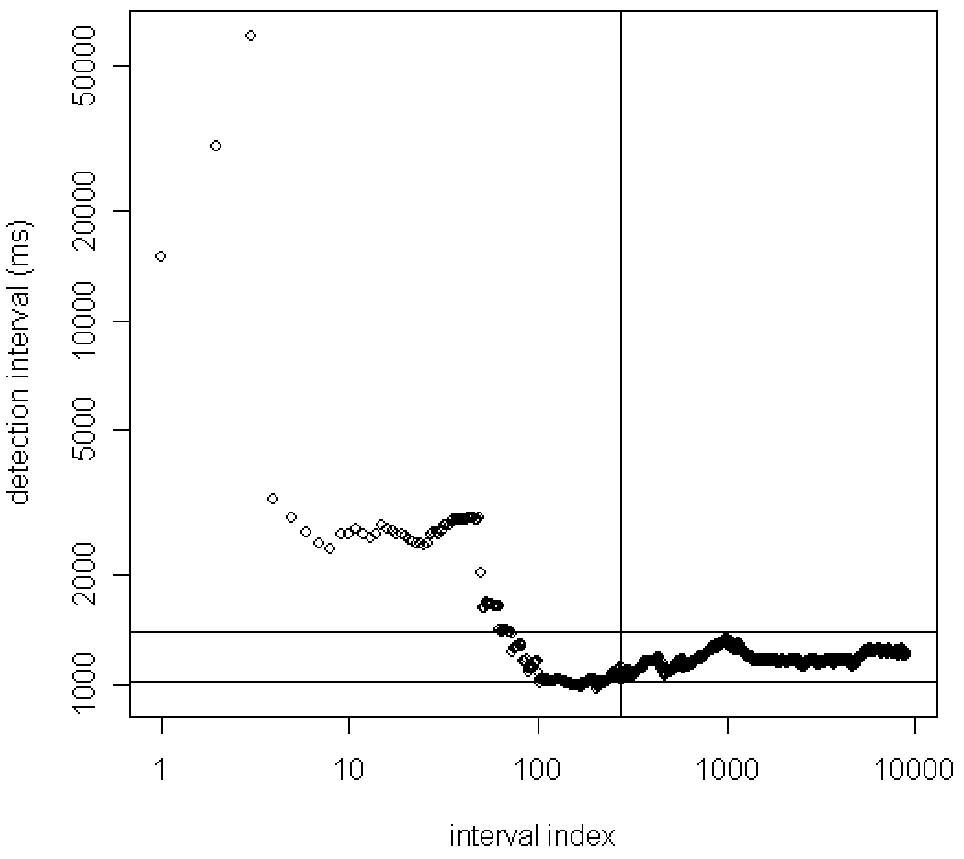

Figure 3.4: Learning curve for the Java experiment (Log-Scaled Axes)

In this case, the scheduling algorithm can take (resp. apply) detection-intervals from (resp. to) all running server instances within the cluster, and still it is likely that only a few hundreds of detections in the presence of a few deadlocks *in total* are needed for the algorithm to approach near-optimal detection intervals.

3.8 Summary

In this chapter, I provided a decision-theoretic learning approach to scheduling deadlock detection for Java, described not only a simulation study but also a case study using a simple yet sufficiently realistic Java program, and showed that the approach of deadlock-detection scheduling as reinforcement learning would be practical and promising for restart-oriented systems.

Percentage %	$x(i)$ s	$y(i) - x(i)$ ms
0%	0	1025.1
10%	3.5	1145.7
20%	8.1	1160.3
30%	14.2	1171.0
40%	20.4	1190.1
50%	27.5	1209.7
60%	38.2	1221.6
70%	48.7	1230.4
80%	67.3	1239.1
90%	96.2	1247.9
100%	221.6	60006.3

Table 3.6: Productive time period lower bound $x(i)$ and deadlocked-interval size $y(i) - x(i)$

Chapter 4

Deadlock Resolution via Exceptions

4.1 Overview

Due to the difficulty of the state-explosion problem, it is inherently hard to find and remove deadlocks in multithreaded programs. There are some tools to help find deadlocks in multithreaded Java programs, but they are not widely used in industry for various reasons. One technical reason is that these tools cannot efficiently handle large real-life programs, which may dynamically load classes from networks, without generating too many spurious warnings. Moreover, although some of the tools can show in a conservative way the absence of deadlocks in some small programs not using certain Java features, they cannot be used to certify large real-life programs for deadlock freedom. Furthermore, it is possible to write deadlock-free code using well-known prevention methods such as linearly ordering resources for unique resource, but it is not practical to apply these methods to dynamically created resources in real-life programs. Consequently, it is difficult for programmers to write deadlock-free code only, and most existing class libraries do not bear a certificate for deadlock freedom.

Nowadays when building truly dependable multithreaded applications, programmers cannot use or produce code not guaranteed to be deadlock free. Thus, the productivity of dependable applications containing deadlock-free-only code is quite unsatisfactory. To improve software productivity and quality, it would be a necessary breakthrough to provide a systematic and programmable approach for incorporating code that is not deadlock free into dependable applications. Because at runtime it is relatively easy to detect actual deadlock occurrences, which represent not only abnormal states but also fatal errors, it is natural to consider deadlock occurrences as runtime exceptions. Thus, exception handlers associated to deadlock-able code can be exploited to resolve potential deadlock occurrences during the execution of code.

In addition, because exceptions are a widely understood language construct supporting forward recovery [37, 43, 11], the approach of deadlock resolution via exceptions is intuitive for programmers (to learn) to use and is appropriate for real-life large programs. Furthermore, exception objects contain rich and useful information about the deadlock occurrences, and the exception handlers can access local program states. Thus, the approach allow programmers to select and implement suitable fine-grained resolution actions.

This chapter [50, 51] describes an approach of deadlock resolution via exceptions. The approach is not restricted to Java. Rather, it applies to any programming language that supports both exceptions and multi-threading. However, for presentation purposes, I use Java as the programming language to discuss the design, implementation and application of deadlock exceptions.

The rest of this chapter is organized as follows. Section 4.2 describes an approach to representing deadlocks as exceptions and discusses two types of deadlock-exception handlers. In Section 4.3, I restrict resources to monitor locks and analyze a JVM-based implementation of deadlock exceptions and their handlers. In Section 4.4, I focus on user-defined resources and exploit a class library to implement deadlock exceptions and their handlers. Section 4.5 further illustrates the utility of the deadlock exceptions and their handlers in programming practice. Section 4.6 concludes this chapter.

4.2 Design

I first briefly introduce exception handling in Java, then present a design for encoding various deadlock states into exceptions, and then discuss two types of handlers for deadlock exceptions.

4.2.1 Exception handling in Java

As part of its runtime support, Java provides an exception-handling mechanism to help programmers write reliable and robust programs in a structured and controlled manner.

Java exceptions are first-class objects representing runtime errors, and they contain rich information about the exception state for the sake of exception handling. Like other types of objects, exceptions can be created, passed to methods as arguments, and garbage collected.

Unlike other types of objects, exceptions can be thrown by throw statements in program code or by the JVM.

When exceptions are thrown, they are passed to their handlers, the closest dynamically enclosing catch clauses that can handle the thrown exceptions, unless the handlers are unavailable. Catch clauses are associated to “try blocks”, which represent code that needs to be protected against exceptions. There can be several catch clauses for a try block, as long as they catch different types of exceptions.

Upon receiving an exception object, an exception handler begins to execute. If there is not an exception handler for an exception, the uncaughtException method of this thread’s UncaughtExceptionHandler is invoked. If this thread does not have an UncaughtExceptionHandler, its ThreadGroup object is considered as its UncaughtExceptionHandler.

Programmers can define their own exception classes by extending the existing exception hierarchy. Java exceptions are objects of the predefined class Throwable or its subclasses. RuntimeException is a subclass of Throwable. Deadlock exceptions are defined as new subclasses of RuntimeException.

4.2.2 A base class for deadlock exceptions

After a deadlock is detected, it is represented and signaled by an exception. *An exception for representing a deadlock should contain rich and helpful information to support deadlock resolution.* In particular, it should provide access to the following information:

- The number of threads involved in the cycle
- For each thread involved in the cycle:
    1. The thread object
    2. The resource that this thread holds and that is involved in this deadlock
    3. The resource the thread is waiting for

The “number of threads involved” gives programmers an intuitive knowledge of how complex the deadlock is. The encoding of the cyclic wait provides useful information for deadlock resolution. As will be shown in a use case study in Subsection 4.3.6, even the names of deadlocked thread objects can help deadlock resolution.

The exception class, denoted as *DeadLock*, which contains the aforementioned fields is considered as the base class for deadlock exceptions. Users can customize their own deadlock exception classes by extending the base class. For example, sometimes it helps to include the stack traces of all deadlocked threads in the deadlock exception. In this case, users can define a subclass containing the stack traces in addition to the aforementioned fields. In the rest of this chapter, the discussion is focused on the base class.

A deadlock exception, which represents a deadlock occurrence, is supposed to be handled by a well-designed handler that can resolve the deadlock occurrence. I discuss two types of deadlock exception handlers in the next subsection.

4.2.3 Deadlock exception handlers: global versus local

Deadlock exception handlers can be installed for an application thread that may deadlock. These deadlock handlers are classified as *local deadlock handlers*. One approach to make use of local deadlock handlers is to have the JVM runtime throw a deadlock exception to a thread that would otherwise be about to deadlock. This approach was partially implemented around Summer 2002 [50]. Local deadlock handlers can exploit threads' local states and program semantics to perform fine-grained recovery actions like releasing a resource currently not in use and picking up a possibly deadlock-free execution path.

Because it is hard to know beforehand which threads will get involved in a deadlock in which order, in most cases local deadlock handlers have to be installed for all potentially deadlocked threads in order not to miss a deadlock exception. Furthermore, this time-consuming task is not even always feasible in the presence of unchangeable and invisible code. In addition, even if all potentially deadlocked threads have local deadlock handlers installed, without application knowledge it is difficult to know which thread to throw the deadlock exception to results in the most cost-effective way to resolve the current deadlock.

To overcome the shortcomings of local deadlock handlers, when a deadlock is detected, it is desirable to get the deadlock exception thrown to a special thread, referred to as the deadlock resolver. The *global deadlock handler* is used to refer to the deadlock exception handler (for *DeadLock* instances) installed for and executed by the deadlock resolver. The deadlock resolver is set to have the highest thread priority and should be started before any other threads in order not to miss some deadlocks.

The global deadlock handler is suitable for performing coarse-grained recovery actions such as killing a thread. However, unlike local deadlock handlers, it cannot perform some fine-grained recovery actions based on deadlocked thread states. To exploit the benefits of local deadlock handlers, the global handler can exploit application knowledge to select a deadlocked thread with local deadlock handlers installed, and delegate the deadlock exception object to this deadlocked thread.

The two complementary deadlock handler types enable effective deadlock recovery in programming practice. In terms of implementation, local deadlock handlers are in the form of catch clauses in order to take the advantage of the exception-handling mechanism of the language, but the global deadlock handler does not need to be, especially when the thread performing deadlock detection and the deadlock resolver are the same thread. However, it is assumed that, when handling a deadlock exception, both global and local deadlock handlers actually break the cycle in the current WFG, thus resolving the corresponding deadlock.

There are synchronization issues between the thread that performs deadlock detection, the deadlock resolver, and threads that have local deadlock handlers installed. The next subsection discusses these synchronization issues.

4.2.4 Synchronization issues

Suppose

1) Thread A performs a deadlock detection, and finds $N > 0$ concurrent deadlocks. In the case of periodic detection, these N deadlocks do not share threads; in the case of continuous detection, $N = 1$ and the detected deadlock contains the thread whose current outstanding request initiated the deadlock detection. Thread A then constructs N deadlock exceptions and reports the exceptions to the deadlock resolver,

2) Thread B is the deadlock resolver; the global deadlock handler associated with Thread B handles M out of N deadlock exceptions, and delegates the rest of the deadlock exceptions to local deadlock handlers,

and

3) Thread C_1,C_2,...,C_{N-M} execute the local deadlock handlers to handle the delegated deadlock exceptions.

Thread A and Thread B may be the same, but other threads, which are deadlocked threads, are different from each other. Lack of proper synchronization between these different threads may result in unexpected behaviors.

Consider the following scenario: Thread A sends N deadlock exceptions to Thread B. The relationship between Thread A and Thread B is like that between a producer and a consumer. So, a buffer can be used to store deadlock exceptions in order not to miss any deadlock exception. Further, it is important to ensure that every deadlock exception gets processed by Thread B.

Consider another scenario: Thread A detects the same deadlock for the second time before the deadlock exception gets handled by a local handler, and Thread A reports the deadlock exception for the second time (to Thread B) after the deadlock exception has already been handled by the local handler. In this scenario, it is possible that the deadlock exception is delegated to the local handler for the second time but unfortunately gets uncaught. If a deadlock exception is guaranteed to be reported exactly once before it is handled, this scenario does not come into being.

So, to address such synchronization issues, it is sufficient that an implementation ensures the *Synchronization Property* that a deadlock exception is reported to the deadlock resolver once and exactly once before it is handled and every deadlock exception is handled by the global deadlock handler (and a local deadlock handler in the presence of delegation) once and exactly once.

To achieve the *Synchronization Property*, it is necessary that neither the deadlock detection thread or the global deadlock handler blocks forever. Actually, if the deadlock detection thread or the deadlock resolver blocks forever such as getting involved in a deadlock, the system may get stuck since future deadlocks will not be detected or resolved.

It is worthwhile to note that currently-resolved deadlocks may repeat themselves in the future. So, if deadlocks are detected after the corresponding deadlock exceptions have been handled, the deadlock exceptions have to be reported again in order to be handled again (potentially by different local handlers).

In the next two sections, I describe two implementations of the approach of deadlock resolution via exceptions. One implementation is within a Java Virtual Machine (JVM), and the other is outside any JVM.

4.3 Implementation within a JVM

Consider a common Java deadlock case in which “two or more threads block each other in a vicious cycle while trying to access synchronization locks needed to continue their activities” [28]. In such a deadlock case, the resources are reusable locks. Before Java 5.0, Java only provided monitor locks. Since Java 5.0, Java provides a package “java.util.concurrent.locks” in addition to monitor deadlocks. Without loss of generality, I focus the discussion of this section on monitor locks only. Thus, in the rest of this chapter, a lock means a monitor lock in Java.

To address deadlocks in the above deadlock scenario, I constructed an initial implementation of the deadlock exception approach into a modified Latte 0.9.1 JVM (Java Virtual Machine). Latte [48] is a Java Virtual Machine that can execute Java bytecode. In addition, Latte provides a just-in-time compiler that dynamically translates Java bytecode into native code, an on-demand exception-handing mechanism, and a lightweight monitor implementation [48]. Currently, Latte runs on Solaris 2.5+ on top of UltraSPARCs, and it has its own thread package implemented inside the JVM.

Below I first briefly introduce Java monitors, then discuss four implementation issues: deadlock exception, deadlock detection, deadlock delegation, and deadlock resolver. At the end of this section, I describe a use case study.

4.3.1 Monitors in the Java language

Java adopts Mesa-style monitors for thread communication and synchronization [27]. Java monitors are in the form of synchronized methods or synchronized statements. A thread has to acquire a lock associated with a monitor in order to enter it. When the thread leaves the monitor, the thread releases the lock. Every object has a lock.

Java provides condition variables in the form of the methods of wait(), notify() and notifyAll() on class Object. For a clear presentation, in this dissertation I assume wait() is invoked without a timeout value. A thread can wait in a monitor by invoking wait(). Specifically, the thread is blocked on the condition variable of the monitor after it invokes wait() and before it is awakened.

A thread that has invoked wait() releases the lock associated to the monitor, and it is disabled from scheduling until the JVM sends it a notification, which is produced by another thread via an invocation of notify(), notifyAll() or interrupt(). Java allows a thread with adequate permission to interrupt another thread blocked on a condition variable by invoking interrupt() for the blocked thread. Java provides other methods for thread communication and manipulation. For example, a thread can wait for the termination of another thread via join(), and a thread can kill another thread via stop().

However, the stop() API is deprecated because it is inherently unsafe. Specifically, invoking stop() on a thread will cause the thread to release all locks it holds thus leaving the objects protected by those locks potentially in inconsistent states. Therefore, stop() is now not guaranteed to always function correctly.

Thus, once a Java thread is blocked due to waiting for a monitor lock, there is no effective programming API to effectively change the thread to the ready state. So, the JVM has to be modified in order to get a deadlocked thread to execute local deadlock handlers.

4.3.2 Deadlock exception

The exception for deadlocks, DeadLock, is a subclass of RuntimeException. When a deadlock is detected, some native code is used to construct a DeadLock object within the Latte JVM. It has 4 fields:

Listing 4.1: Fields in DeadLock exception in a Latte-based implementation

```
int size;
Thread[] waiters;
Object[] locks_held_in_deadlock;
Object[] locks_waiting;
```

The first field *size* is the number of deadlocked threads in this deadlock. The following three fields are arrays of size *size*. The array *waiters* stores the deadlocked threads. The element *locks_held_in_deadlock*$[i]$ stores the lock that *waiters*$[i]$ holds and that is being waited for by *waiters*$[(i - 1 + \textit{size}) \bmod \textit{size}]$. The element *locks_waiting*$[i]$ stores the lock that *waiters*$[i]$ is waiting for and that is being held by *waiters*$[(i + 1) \bmod \textit{size}]$.

4.3.3 Deadlock detection

I adopt a continuous deadlock-detection method that is easily implemented inside Latte. The detection method is based on finding a new cycle in the WFG (Waits-For Graph), which is locked during deadlock detection. Nodes in the WFG represents the threads, and there is an edge from the node representing thread T_1 to that representing thread T_2 if T_1 is waiting for a lock held by T_2.

Only a contended lock request, which means a request for a lock already held by a thread other than the requesting thread, will trigger deadlock detection. The detection is performed by taking a directed walk in the WFG starting from the node R representing the requesting thread: if the node R is encountered again during the walk, then a deadlock that needs to be reported is found. Otherwise, either the system currently has no deadlock or the requesting thread is transitively blocked by a thread in a deadlock that has been reported via an exception but that has not been resolved yet.

The complexity of detecting a deadlock in this case is $O(n)$, where n is the number of threads in the current system. Liang and Viswanathan [31] claimed that lock contention is rare in well-tuned programs since lock contention is usually due to "multiple threads holding global locks too long or too frequently." Further, they reported that during one run of mtrt, the only multi-threaded program in the SPECjvm98[1] benchmark suite, 11 out of 715244 lock requests are contended requests.

Each continuous deadlock detection finds at most one deadlock that needs to be reported, and every deadlock that needs to be reported is detected as it occurs. So, it is safe to create a deadlock exception for this deadlock and to report it to the deadlock resolver, described next.

[1]http://www.spec.org/osg/jvm98/.

4.3.4 Deadlock resolver

Programmers can choose to deploy a thread as the deadlock resolver; the deadlock exception handler (for DeadLock instances) to be executed by the deadlock resolver is the global deadlock handler. To be deployed as the deadlock resolver, a thread should have a specific name so that the Latte JVM can recognize it as the deadlock resolver. Currently, a deadlock resolver should have "NoTimerResolver" as its thread name. The deadlock resolver is set to have highest thread priority, and should be started before any other threads in order to avoid missing some deadlocks.

In the Latte-based implementation, the thread performing deadlock detection is different from the deadlock resolver. The former produces a deadlock exception, and the latter consumes deadlock exceptions. So, they have the producer-consumer relationship, and share a First-In-First-Out (FIFO) buffer.

The deadlock resolver invokes join() for itself without a timeout value. In regular programs under standard JVMs, an invocation of join() for the current thread without a timeout value makes the current thread blocked forever. However, in the Latte-based implementation that supports deadlock exceptions, the implementation of join() is customized for the deadlock resolver, which has "NoTimerResolver" as its name, so that the deadlock resolver behaves as a producer.

Usually the invocation of join() is contained in a loop for the sake of continuous deadlock resolution. Every time the deadlock resolver performs a join() for itself, it checks if there is any deadlock exception in the FIFO buffer. If yes, it removes the first exception from the buffer, wakes up any thread that is waiting for the FIFO buffer to be not full, and throws it to the global deadlock handler, which is in the form of a catch clause. Otherwise, it blocks until exceptions arrive at the FIFO buffer.

The thread that has made a contended lock request performs a deadlock detection. If it detects a deadlock occurrence, it creates an exception for this deadlock occurrence, saves the exception in a First-In-First-Out (FIFO) buffer, and wakes up the deadlock resolver (if it is currently blocked).

4.3.5 Deadlock delegation

The global deadlock handler is suitable for performing coarse-grained recovery actions such as killing a thread. However, unlike local deadlock handlers, it is not able to perform some fine-grained recovery actions based on deadlocked thread states. To exploit the benefits of local deadlock handlers, the deadlock resolver can select a deadlocked thread with local deadlock handlers installed, and delegate the deadlock exception object to this deadlocked thread. No new API is needed for delegation; the deadlock resolver just invokes interrupt() for the thread which the deadlock exception is to be delegated to. When executing interrupt() for a deadlocked thread invoked by a deadlock resolver, the JVM runtime will restore the deadlocked thread to the state right before it got deadlocked and then throw the current deadlock exception to it.

In sum, the JVM-based implementation does not require programmers to learn new APIs. Rather, it only asks programmers to use some easy-to-follow programming conventions when using existing Thread APIs. In the next subsection, I describe a use case of this implementation.

4.3.6 A use case

The use case in this subsection shows how to resolve deadlocks involving locks in a system of two money-transfer transactions. The two simultaneous transactions are as follows: one is to transfer some money from a savings account *s* to a checking account *c*, the other is to transfer some money from *c* to *s*. The full code listing (including the definitions of all classes to be discussed in this subsection) is in the Appendices (Section 6.3).

Suppose class Account is unchangeable. The transfer method, as shown in List 4.2, in class Account specifies how to perform a money-transfer transaction. The method contains a locking order bug in two phase locking. Specifically, this bug causes a potential deadlock: the two threads may hold a lock and wait for the lock held by the other thread.

Listing 4.2: A locking-order bug

```
public synchronized void transfer(Account to, int amount){
try {
    Thread.sleep(100);
}catch (InterruptedException e) {}
synchronized (to) {
   if (value >= amount) {
      to.value = to.value + amount;
      value = value-amount;
}}}
```

Class S2CTransfer (C2STransfer, resp.) defines the run() method used by thread S2C (thread C2S, resp.), which implements the transaction that transfers money from the savings (checking, resp.) account to the checking (savings, resp.) account.

Suppose class C2STransfer is changeable, but class S2CTransfer is unchangeable. There is no local deadlock exception handler installed for thread S2C, since class S2CTransfer is unchangeable. A local deadlock handler is plugged into the run() method of class C2STransfer. When a DeadLock exception is caught by this local handler, the current thread has already released the lock it owned. Thus, as shown in the code fragment below in Listing 4.3, this local handler just lets the current thread, i.e., thread C2S, waits for a while so that the other thread, i.e., thread S2C, can get a chance to finish.

Listing 4.3: A local handler

```
while (!successful){
  try {
    a1.transfer(a2,amount);
    successful = true;
  }catch (DeadLock e) {
    try {
      Thread.sleep(200);
    }catch (InterruptedException e1) {}
}}
```

Class DeadlockResolver defines how the deadlock resolver (NoTimerResolver) works. As shown in the code fragment in Listing 4.4, NoTimerResolver invokes join() for itself. The global deadlock handler installed for NoTimerResolver is in the form of a catch clause. When a DeadLock exception is caught, the deadlock exception is delegated to thread C2S, which installs a local handler for DeadLock exceptions.

Listing 4.4: A global handler

```
while (cont){
  try {
        Thread.currentThread().join();
  }catch (InterruptedException e0){
        cont = false;
  }catch (DeadLock e1){
        if (e1.waiters[0].getName().equals(``S2C''))){
           e1.waiters[1].interrupt();
        }else {
           e1.waiters[0].interrupt();
}}}
```

Class Driver describes creation of NoTimerResolver, thread S2C and thread C2S. NoTimer-Resolver is a thread with the name “NoTimerResolver” in the thread group with the name “DeadlockResolverGroup.” It is set to have the highest priority and is started before thread S2C and thread C2S in order not to miss any deadlock exceptions.

With the help of the 2 deadlock handlers, the potential deadlock involving thread S2C and thread C2S can be resolved and both threads can accomplish their money-transfer transactions.

4.4 Implementation outside a JVM

In this dissertation, I consider deadlocks in centralized systems with unique and reusable resources, based on the one-resource deadlock model [25] in which a task can have at most one outstanding request at one time and blocks until the resource is granted.

Locks are not the only interesting resources in user applications. In this section, I first present a general resource type that works with deadlock exceptions. Then, I go on to discuss an out-of-the-JVM (OOTJ) implementation of the deadlock-exception approach, and illustrate it with a use case.

4.4.1 A general resource type

The full code listing for this resource type is in the Appendices (Section 6.4). Below, I describe some important methods.

The resource type provides a *request* method and a *release* method. A thread can request a resource via the *request* method. After a thread finishes using a resource, it can return the resource via the *release* method.

Both method *request* and *release* try to acquire lock *l* first. Lock *l* is used to protect the WFG; the thread performing deadlock-detection also needs to acquire lock *l* before executing code that manipulates the WFG. The use of lock *l* guarantees that the WFG is not modified during the process of deadlock detection.

After acquiring lock *l*, if the resource is free, the thread will get the resource. If the resource is owned by some other thread, the requesting thread is put to a waiting table. Once the resource becomes free again, a waiting thread for this resource is picked up to get this resource.

The code fragment implementing the request and release methods is shown below:

Listing 4.5: A general resource

```
private Thread owner = null;
private Thread thrower = null;
private DeadLock exception = null;
public static Object l = new Object();
public resource request() {
  while (true) {
    synchronized (l) { //protect WFG
      synchronized (this) { //protect current resource
        if (owner==null || owner==Thread.currentThread()) {
          h.remove(Thread.currentThread());
          owner = Thread.currentThread();
          return this;
        }
        if (thrower==Thread.currentThread()) {
          thrower = null;
          DeadLock e = exception;
          exception = null;
          throw e;
        }
        h.put(Thread.currentThread(),this);
    }}
    synchronized (this) {
      try {
        while ((thrower!=Thread.currentThread()) && (owner!=
          null))
          this.wait();
      }catch (InterruptedException e) {}
}}}
public void release() {
  synchronized (l) { //protect WFG
    synchronized (this) { //protect current resource
      if (owner==Thread.currentThread()) {
        owner = null;
        this.notifyAll();
}}}}
```

The *request* method also contains code to throw a deadlock exception. This piece of code works with the *setThrower* method, as described below, to delegate a deadlock exception. Specifically, after a deadlock is detected, the deadlock resolver can exploit *setThrower* to delegate a deadlock exception *e* to some local handler installed for a deadlocked thread *t*.

Listing 4.6: The setThrower method

```
public synchronized void setThrower(Thread t,DeadLock e) {
  thrower=t;
  exception=e;
```

```
    this.notifyAll();
}
```

4.4.2 Deadlock exception

Below, I describe a class for deadlock exceptions. The full code listing for this deadlock exception class is in the Appendices (Section 6.5).

Listing 4.7: Fields of DeadLock exception in an OOTJ implementation

```
public class DeadLock extends RuntimeException{
    public int size = 0;
    public Thread[] waiters;
    public resource[] resources_held;
    public resource[] resources_waiting;
    //constructors and methods are omitted here
  }
```

The first field *size* is the number of deadlocked threads in this deadlock. The following three fields are arrays of size *size*. The array *waiters* stores the deadlocked threads. The element *resources_held*$[i]$ stores the resource that *waiters*$[i]$ holds and that is being waited for by *waiters*$[(i-1+\textit{size}) \bmod \textit{size}]$. The element *resources_waiting*$[i]$ stores the resource that *waiters*$[i]$ is waiting for and that is being held by *waiters*$[(i+1) \bmod \textit{size}]$.

So, the OOTJ implementation of deadlock exceptions and the Latte-based implementation contain similar fields.

4.4.3 Deadlock detection

The deadlock detection is performed at the application level. I adopt a periodic detection method in this OOTJ implementation. As shown in other work [1, 8], the complexity of detecting a deadlock in this case is $O(n)$, where n is the number of threads in the current system.

The periodic detection method may find multiple cycles in the WFG. In this case, there are multiple concurrent deadlocks. Given the deadlock model in this dissertation, a thread can be involved in at most one deadlock. That is, multiple concurrent deadlocks do not share a thread.

When one or more deadlocks are detected, corresponding deadlock exceptions are reported to the deadlock resolver. To ensure that deadlock exceptions are reported to the deadlock resolver once and exactly once before they are handled, the next deadlock detection is not performed until all deadlock exceptions for the concurrent deadlocks are handled by the deadlock handlers.

4.4.4 Deadlock resolver

The deadlock resolver and deadlock detection can share the same thread. This thread does not need to use a special name, because the implementation of deadlock detection and deadlock resolver is at the application level.

The global exception handler in this case does not need to use a catch clause to catch deadlock exceptions. For example, the deadlock exceptions constructed during deadlock detection can be saved in an array, and the global exception handler can be in terms of a code sequence that examines the array and then takes appropriate actions.

One action may be to delegate a deadlock exception to a local deadlock handler. Below, I discuss the OOTJ implementation of deadlock delegation.

4.4.5 Deadlock delegation

In the OOTJ implementation, deadlock delegation does not use the interrupt() API. Rather, it exploits the setThrower API provided by the resource type.

Consider the following statement:

Listing 4.8: Deadlock delegation in an OOTJ implementation

```
e.resources_waiting[i].setThrower(e.waiters[i],e);
```

Suppose variable e stores a deadlock exception. What this statement does is to delegate the deadlock exception e to the $(i+1)$th deadlocked thread in the deadlock. So, like the Latte-based implementation, deadlock delegation can also be accomplished by a single statement.

Upon receiving a delegated deadlock exception, a local deadlock handler starts to handle the exception, and will notify the deadlock-detection thread after it finishes handling the deadlock exception. The use case in the next section shows how to do the notification, among other things.

4.4.6 A use case

The use case in this subsection is essentially the same as what is used to illustrate the Latte-based implementation. There are two simultaneous transactions in the system: one is to transfer some money from a savings account *s* to a checking account *c*, and the other is to transfer some money from *c* to *s*. The full code listing is in the Appendices (Section 6.6).

Assume Class Account, a subclass of the class implementing the generic resource type, is unchangeable. The transfer method, as shown in Listing 4.9, in Class Account specifies how to perform a money-transfer transaction. The method contains a resource-request ordering bug. Specifically, this bug causes a potential deadlock: the two threads can hold a resource and wait for the resource held by the other thread.

Listing 4.9: A resource-request ordering bug

```
public void transfer(Account to, int amount){
  this.request();
  try {Thread.sleep(200);} catch (Exception ee){}
  to.request();
  if (value >= amount) {
    to.value = to.value + amount;
     value = value-amount;
  }
  to.release();
  this.release();
}
```

Class S2CTransfer(C2STransfer, resp.) defines the run() method used by thread S2C(thread C2S, resp.), which implements the transaction that transfers money from the saving(checking, resp.) account to the checking(saving, resp.) account.

Suppose class C2STransfer is changeable, but class S2CTransfer is unchangeable. There is no local deadlock exception handler installed for thread S2C, since class S2CTransfer is unchangeable. A local deadlock handler is plugged into the run() method of class C2STransfer. Unlike the local handler in Listing 4.3 in the Latte-based implementation, the local handler as shown in the code fragment in Listing 4.10, upon catching a DeadLock exception, first releases the resource it holds, notifies the deadlock-detection thread that the current deadlock has been handled, and then lets the current thread, i.e., thread C2S, wait for a while.

Listing 4.10: A local deadlock handler for general resources

```
while (!successful){
   try {
      a1.transfer(a2,amount);
      successful = true;
   }catch (DeadLock e) {
      a1.release();
      synchronized (clients.lock) {
         clients.resolved++;
         clients.lock.notify();
      }
      try {Thread.sleep(200);} catch (Exception ee){}
}}
```

The deadlock resolver is shown in Listing 4.11. No naming convention is needed for the thread serving as the deadlock resolver, and the global handler installed for the deadlock resolver is not in the form of a catch clause. In addition, the deadlock resolver also performs a periodic deadlock detection. If a deadlock is found, it creates a deadlock exception, and then delegates the exception to the local handler of thread S2C. Then, the thread is waiting for a notification from the local handler that the deadlock has been resolved. After the notification is received, the thread will continue the periodic deadlock detection.

Listing 4.11: A global deadlock handler for general resources

```
boolean cont = true;
while (cont) {
   try {
     Thread.sleep(5000);
   }catch (InterruptedException e) {
     count=false;
   }
   /*Acquiring the lock resource.l to protect WFG, exploiting
     an O(n) cycle-detection method to find deadlocks,setting
     number_of_deadlocks to be the number of deadlocks found,
     constructing exceptions for the found deadlocks, and
    storing the exceptions in the array: currentDeadlocks.
    Details omitted */
   if (number_of_deadlocks > 0) {
    for (int i=0; i<number_of_deadlocks; i++) {
      DeadLock e=currentDeadlocks[i];
      e.DeadlockPrint();
      for (int j=0;j<e.size;j++)
      if (e.waiters[j].getName().equals(‘‘C2S’’)){
         e.resources_waiting[j].setThrower(e.waiters[j],e);
           break;
    }}}

    if (number_of_deadlocks >0) {
       synchronized (lock) {
         while (resolved < number_of_deadlocks)
           lock.wait();
           resolved = 0;
}}}
```

Again, with the help of the 2 deadlock handlers, the potential deadlock involving thread S2C and thread C2S can be resolved and both threads can accomplish their money-transfer transactions.

If an implementation uses periodic detection, programmers need to write synchronization code between deadlock handlers and any thread that performs deadlock detection. On the other hand, if an implementation uses continuous detection, deadlock detection is triggered by a contended resource request and there is no need for programmers to write extra synchronization code, which could be difficult if the implementation (of deadlock detection) is within a JVM.

The Latte-based implementation requires programmers to use some specific naming conventions, but the OOTJ implementation does not. In addition, the OOTJ implementation supports applying exceptions to deadlocks involving generic resources, which can play the role of locks among many others. Moreover, given that nowadays JVM's are considered as exchangeable commodities, currently programmers are likely reluctant to rely on a customized JVM. While I have shown that it is feasible to implement the approach in a JVM, I will use the OOTJ implementation to illustrate the deadlock exceptions and their handlers in programming practice in the next chapter.

4.5 Application

In practice, users can exploit deadlock exceptions and their handlers to resolve deadlocks in various effective ways. In this section, I use one example to show deadlock resolution via selecting a different forward execution path. Another example shows how to resolve a deadlock by releasing a resource currently not under use. Yet another example describes how to handle multiple deadlocks detected at one time by the periodic detection method. The last but not least example shows deadlock resolution via restarting the system in the global deadlock handler.

4.5.1 Selecting a different execution path

In the use case discussed in the previous section, after the deadlock exception is caught, the recovery action is to have a deadlocked thread release the resource it holds, wait for a while, and then retry the deadlocked operation. Exception handling mechanisms are known to be suitable for forward error recovery [37, 43, 11]. So, besides retrying the previously-deadlocked operation as described in the use case, the deadlock in the use case study can be resolved by selecting a different forward-execution path.

Listing 4.12: Resolving deadlock via a different execution path

```
try {
   a1.transfer(a2,amount);
}catch (DeadLock e) {
   System.out.println(''Caught an exception!'');
   a1.withdraw(amount);
   synchronized (clients.lock) {
      clients.resolved++;
      clients.lock.notify();
   }
   a2.deposit(amount);
}
```

As shown in Listing 4.12, after a deadlock exception is caught, the handler (installed for thread C2S) will withdraw the money from the checking account, notify the deadlock resolver that performs deadlock detection also, and then deposit it to the savings account. That is, the deadlock is resolved by selecting the alternative execution path.

Neither withdraw() nor deposit() requests two or more resources. The two methods are shown in Listing 4.13 below:

Listing 4.13: Method withdraw and deposit

```
public void deposit(int amount){
   this.request();
   value = value+amount;
   this.release();
}
public void withdraw(int amount){
   this.request();
   if (value >= amount) {
      value = value-amount;
```

```
    }
    this.release();
}
```

It is worthwhile to note the notification of the deadlock resolver is done between the invocation of withdraw() and that of deposit(). It is after the invocation of withdraw() because withdraw() releases a resource in the cyclic wait, thus breaking the cycle. It is before the invocation of deposit() because deposit() needs to acquire a resource. More specifically, if the notification of the deadlock resolver is done after the invocation of deposit(), the notification may never be sent out because the invocation of deposit() may make the current thread wait for a resource held by a deadlocked thread.

4.5.2 Releasing a resource currently not under use

A local deadlock handler can choose to resolve a deadlock by releasing a resource that is being waited for by another deadlocked thread and that is not being used by the current deadlocked thread.

Suppose there are 2 threads in a system. Thread AGGRESSIVE requests resource FAX, but does not use it immediately. Then, it requests resource PRINTER followed by requesting resource SCANNER, and will use PRINTER and SCANNER after getting them. Then, it will go back and use FAX. On the other hand, thread LAZY requests resource SCANNER followed by requesting FAX, and uses these 2 resources after getting them.

It is possible that thread AGGRESSIVE and LAZY get involved in a deadlock in which thread AGGRESSIVE holds resource FAX but waits for resource SCANNER and thread LAZY holds resource SCANNER, but waits for resource FAX. In this case, a local deadlock handler associated with thread AGGRESSIVE can choose to release resource FAX. Thread AGGRESSIVE will have to reacquire resource FAX before using it.

Thread AGGRESSIVE executes the code shown in Listing 4.14. The local handler examines the deadlock exception, and releases resource FAX if FAX is involved in the deadlock. If FAX is not involved in the deadlock but PRINTER is, the deadlock handler will release PRINTER and then will wait for a while before trying to reacquire PRINTER—a resolution action similar to what is exploited in the use case study in Section 4.4.6.

Listing 4.14: Releasing a resource not under use

```
resources[FAX].request();
boolean succeeded = false;
while (!succeeded) {
 try {
  resources[PRINTER].request();
  try {Thread.sleep(100);} catch (Exception ee){}
  resources[SCANNER].request();
  succeeded = true;
  System.out.println(``Thread ''+Thread.currentThread().
   getName()+`` is using PRINTER and SCANNER.'');
 }catch (DeadLock e){
  System.out.println(``Caught an exception!'');
  for (int i=0;i<e.size;i++) {
  if (e.resources_held[i].getId()==FAX) {
    resources[FAX].release();
    synchronized (lock) {
      resolved++;
      lock.notify();
    }
    break;
  }
  if (e.resources_held[i].getId()==PRINTER) {
    resources[PRINTER].release();
    synchronized (lock) {
      resolved++;
      lock.notify();
    }
    try {Thread.sleep(5000);} catch (Exception ee){}
    break;
 }}}}
 if (resources[FAX].getOwner() == null)
    resources[FAX].request();
 else if (!resources[FAX].getOwner().getName().equals(''
  AGGRESSIVE''))
    resources[FAX].request();
 System.out.println(''Thread ''+Thread.currentThread().
  getName()+'' is using FAX.'');
 resources[FAX].release();
 resources[PRINTER].release();
 resources[SCANNER].release();
```

Thread LAZY does not need to have any local deadlock handler installed. It executes the code as shown in Listing 4.15.

Listing 4.15: Lazy use of resources

```
resources[SCANNER].request();
try {Thread.sleep(100);} catch (Exception ee){}
resources[FAX].request();
System.out.println(''Thread ''+Thread.currentThread().
 getName()+'' is using FAX and SCANNER.'');
resources[FAX].release();
resources[SCANNER].release();
```

Like the use case in Section 4.4.6, the global deadlock handler delegates the exception to thread AGGRESSIVE. However, the global deadlock handler is programmed in a more general way, exploiting the size of the deadlock, as show in Listing 4.16.

Listing 4.16: Deadlock delegation using the field *size*

```
if (number_of_deadlocks > 0) {
   for (int i=0; i<number_of_deadlocks; i++) {
      DeadLock e=currentDeadlocks[i];
      e.DeadlockPrint();
      for (int j=0;j<e.size;j++)
      if (e.waiters[j].getName().equals(``AGGRESSIVE'')){
          e.resources_waiting[j].setThrower(e.waiters[j],e);
             break;
}}}
```

4.5.3 Resolving multiple deadlocks concurrently

The periodic deadlock-detection method may detect multiple concurrent deadlocks during one detection. These deadlocks do not share threads. So, a simple yet practical approach to resolving multiple concurrent deadlocks is that, for each deadlock, the deadlock resolver selects a deadlocked thread with local handlers installed for each deadlock and then delegates each deadlock exception to the selected deadlocked thread.

Consider the following use case. There are 32 threads and 128 shared resources in a system. Each thread randomly requests 2 resources, one after another, use them for a while, and then release them. As shown in Listing 4.17, two or more threads may get involved in a cyclic wait for resources. A thread may hold some resource after “resources[source].request();” but uses “resources[dest].request();” to request another resource that is held by another thread. Each cyclic wait corresponds to a deadlock. Further, there can be multiple deadlocks concurrently, and these deadlocks do not share threads.

After a deadlock exception is caught, the handler installed for every thread will release the resource held by the current thread, sleep for a while, attempt to request the 2 resources again. Each thread keeps attempting to request the 2 resources until it obtains them.

Listing 4.17: Multiple deadlock resolution

```
static final int N_ACC = 128;
static final int N_TEL = 32;
static resource[] resources = new resource[N_ACC];
```

```
        // Each thread will run the following loop
while (true) {
  Random r = new Random(System.nanoTime());
  int source = Math.abs(r.nextInt()) % N_ACC;
  int dest = 0;
  do {
     dest = Math.abs(r.nextInt()) % N_ACC;
  }while (dest == source);
  resources[source].request();
  boolean succeeded = false;
  while (!succeeded) {
    try {
      resources[dest].request();
      succeeded = true;
    }catch (DeadLock e){
      System.out.println(``Caught an exception!'');
      resources[source].release();
      synchronized (lock) {
        resolved++;
        lock.notify();
      }
      try {Thread.sleep(5000);} catch (Exception ee){}
      resources[source].request();
  }}
  System.out.println(``Thread ''+Thread.currentThread().
   getId()+`` is using resource ''+source+`` and resource ''
   +dest);
  try {Thread.sleep(50);} catch (Exception ee){}
  resources[source].release();
  resources[dest].release();
}
```

Listing 4.18 below shows how to delegate multiple deadlock exceptions. Each deadlock exception is thrown to the first deadlocked thread stored in the deadlock exception. The deadlock resolver also performs deadlock detection. After all deadlock exception are delegated and *handled*, the deadlock resolver continue its periodic deadlock detection.

Listing 4.18: Multiple deadlock delegation

```
boolean cont = true;
while (cont){
  try {
    Thread.sleep(5000);
  }catch (InterruptedException e) {
    cont = false;
  }
  /* Acquiring the lock resource.l to protect WFG,
   exploiting an O(n) cycle detection method to find
   deadlocks,setting number_of_deadlocks to be the number of
    deadlocks found, constructing exceptions for the found
   deadlocks, and storing the exceptions in the array:
   currentDeadlocks. Details omitted */
  if (number_of_deadlocks > 0) { //can be more than 1 in
   this case
    for (int i=0; i<number_of_deadlocks; i++) {
      DeadLock e=currentDeadlocks[i];
      e.DeadlockPrint();
      e.resources_waiting[0].setThrower(e.waiters[0],e);
  }}
  if (number_of_deadlocks >0) {
    synchronized (lock) {
      while (resolved < number_of_deadlocks)
```

```
          lock.wait();
        resolved = 0;
}}}
```

With the help of deadlock exceptions, all 128 threads can keep running code that may result in multiple deadlocks concurrently in an infinite loop!

4.5.4 Restarting the system to resolve deadlocks

The global deadlock handler can choose to resolve the current one or more deadlocks by restarting the system. In this case, the global deadlock handler does not delegate any deadlock exception, but just restarts the system after saving necessary information. The restarted system will continue its execution after picking up the information.

Listing 4.19 below sketches the sequence to restart the system after one ore more deadlocks have been detected. Information to be saved may include large application-specific data. The script “restart.sh” contains the Java command to run the system again. After restarting the system, the global handler gets the current system to exit. The code sequence does not need to be in an exception handler.

Listing 4.19: Restarting the system,

```
if (number_of_deadlocks >0) {
  try {
    /* saving some necessary information*/
    String command = ``sh restart.sh ''+arg1+`` ''+arg2+`` '
     '+argn;
    System.out.println(``Now execute the recovery command: '
     '+command);
    Process child = Runtime.getRuntime().exec(command);
  }catch (IOException e) {}
  System.out.println(``Now exit the current system...'');
  System.exit(1);
}
```

4.6 Summary

This chapter presented an approach of deadlock resolution via exceptions, and showed that this approach is practical and effective in developing dependable applications containing code that may deadlock. In particular, deadlocks as exceptions allow programmers to write fine-grained recovery code in addition to restarting the entire system.

Chapter 5

Conclusions

5.1 Conclusions

I considered deadlock-detection scheduling as a reinforcement-learning problem. Specifically, based on the assumption that the time to first deadlock in the system (after a system restart) follows an exponential distribution, I established a utility model for restart-oriented systems, proposed a learning algorithm to estimate the deadlock rate and to find the detection interval that maximizes system utility.

I have demonstrated that it is a reasonable approximation that the time to first deadlock in the system (after a system restart) follows an exponential distribution. I have proved that this technique finds the best tradeoff in theory, and I have used both a simulation study and a simple yet sufficiently realistic Java program to show this technique is effective in practice.

I considered deadlocks as exceptions. Using this idea in addition to restarting the system, programmers can exploit exception handlers to resolve deadlock occurrences based on program contexts and deadlock states. I proposed a design of a base class for exceptions, distinguished between global and local deadlock handlers, and described a solution to the synchronization issues that should be addressed in any implementation.

I have presented 2 implementations of deadlock exceptions and their handlers. One implementation is based on a modified Latte JVM, and the other is outside any JVM. I have illustrated the use of deadlock exceptions and their handlers by a use case study and various examples. In the use case study and all the applicable examples, all deadlocks, signaled as exceptions, are resolved effectively by corresponding exception handlers performing fine-grained recovery actions.

Therefore, it is a valid thesis that, under the assumption that the time to first deadlock in the system (after a system restart) follows an exponential distribution, a reinforcement-learning approach is effective in scheduling deadlock detection for a restart-oriented system, and that run-time exceptions are a programming abstraction that allows programmers to write fine-grained deadlock-recovery code.

Chapter 6

Appendices

6.1 A simplified version of class Account for Chapter 3

```
class Account {
  private int value;
  public Account(int v) { value = v;}
  synchronized void transfer(int to, int amount){
    Account toAccount = Experiment.accounts[to];
    if (value<amount) return;
      synchronized (toAccount){
          toAccount.value += amount;
          value -= amount;
}}}
```

6.2 A simplified version of class Experiment for Chapter 3

```
public class Experiment implements Runnable {
 /* Attribute and variable definitions omitted */
   public void run(){
     while (true) {
       /* Do some house keeping work and randomly choose
        source, dest and amount. Details omitted */
       accounts[source].transfer(dest,amount);
     }
   }
```

```
  static double L(double no_deadlock_sum, long n[]){/* to
    compute the MLE for lambda*/}
  static double W(double lambda, double cost) {/* to compute
    the interval*/}
  public static void main(String[] args) throws IOException
    {
/* Create 2 accounts and start 4 threads. Details omitted.
 */
    boolean deadlock_found = false;
    while (!deadlock_found){
        /* Save some intermediate computational results,
         update the interval potentially by invoking L (for
         lambda) and W (for interval), and collect timing
         information. Details omitted. */
        try { Thread.sleep(new Double(interval).longValue())
         ;}
        catch (InterruptedException e) {}
        /*use findMonitorDeadlockedThreads for deadlock
         detection, and collect timing information. If a
         deadlock is found, deadlock_found is set to true.
         Details omitted.*/
    }
/* Do deadlock recovery and pass some data to the next
 restart. Details omitted.  */
  }
}
```

6.3 A bank transfer deadlock example using locks for Chapter 4

```
class Account {
  private int value;
  public String type;
  public Account(int v, String t) {
    value = v;
    type = t;
  }
  public synchronized void transfer(Account to,int amount){
    try {
      Thread.sleep(100);
    }catch (InterruptedException e) {}
    synchronized (to) {
      if (value >= amount) {
        to.value = to.value + amount;
        value = value-amount;
}}}}

class S2C_Transfer implements Runnable {
  private Account a1,a2;
  private int amount;
  public S2C_Transfer(Account a1,Account a2,int amount){
    this.a1=a1;
    this.a2=a2;
    this.amount=amount;
  }
  public void run(){
    a1.transfer(a2,amount);
}}

class C2S_Transfer implements Runnable {
  private Account a1,a2;
  private int amount;
  public C2S_Transfer(Account a1,Account a2,int amount){
    this.a1=a1;
    this.a2=a2;
    this.amount=amount;
  }
  public void run(){
    boolean successful = false;
    while (!successful){
      try {
        a1.transfer(a2,amount);
        successful = true;
      }catch (DeadLock e) {
        try {
          Thread.sleep(200);
        }catch (InterruptedException e1) {}
}}}}

class DeadlockHandler implements Runnable {
  private Account s,c;
  private int s2c,c2s;
  public DeadlockHandler(Account s,Account c,int s2c,int c2s
   ) {
    this.s = s;
    this.c = c;
    this.s2c = s2c;
    this.c2s = c2s;
  }
  public void run(){
    boolean cont = true;
    while (cont){
        try {
          Thread.currentThread().join();
        }catch (InterruptedException e0){
          cont = false;
        }catch (DeadLock e1){
          if (e1.waiters[0].getName().equals(''S2C'')){
             e1.waiters[1].interrupt();
          }else {
             e1.waiters[0].interrupt();
}}}}}

public class Driver {
  public static void main(String[] args){
    ThreadGroup HG = new
    ThreadGroup(''DeadlockHandlerGroup'');
    Account s = new Account(1500,''saving'');
    Account c = new Account(1000,''checking'');
    int s2c = 500;
    int c2s = 600;
    S2C_Transfer trans_1 = new
    S2C_Transfer(s,c,s2c);
    C2S_Transfer trans_2 = new
    C2S_Transfer(c,s,c2s);
    DeadlockHandler DH=new DeadlockHandler(s,c,s2c,c2s);
    Thread resolver = new Thread(HG,DH,''NoTimerHandler'');
    resolver.setPriority(Thread.MAX_PRIORITY);
    resolver.start();
    new Thread(trans_2,''C2S'').start();
    new Thread(trans_1,''S2C'').start();
}}
```

6.4 A general resource type for Chapter 4

```
import java.util.Hashtable;

public class resource {
  private Thread owner = null;
  private Thread thrower = null;
  private DeadLock exception = null;
  private static Hashtable h = new Hashtable();
  private int id = 0;
  public static Object l = new Object();
  public resource() {}
  public resource(int i) { id = i;}

  public resource request() {
   while (true) {
    synchronized (l) { //protect WFG
      synchronized (this) { //protect current resource
        if (owner==null || owner==Thread.currentThread()) {
          h.remove(Thread.currentThread());
          owner = Thread.currentThread();
          return this;
        }
        if (thrower==Thread.currentThread()) {
          thrower = null;
          DeadLock e = exception;
          exception = null;
          throw e;
        }
        h.put(Thread.currentThread(),this);
  }}
  synchronized (this) {
    try{
       while ((thrower!=Thread.currentThread()) && (owner
        !=null))
          this.wait();
    }catch (InterruptedException e) {}
}}}

public synchronized void setThrower(Thread t,DeadLock e) {
  thrower=t;
  exception=e;
  this.notifyAll();
}

public synchronized Thread getThrower() {
  return thrower;
}

public synchronized Thread getOwner() {
  return owner;
}

public synchronized int getId() {
  return id;
}

public static Hashtable getWRTable() {
  return h;
}
```

```

  public void release() {
    synchronized (l) { //protect WFG
      synchronized (this) { //protect current resource
        if (owner==Thread.currentThread()) {
          owner = null;
          this.notifyAll();
}}}}}
```

6.5 A deadlock exception class for Chapter 4

```
import java.util.Hashtable;

public class DeadLock extends RuntimeException{
  public int size = 0;
  public Thread[] waiters;
  public resource[] resources_held;
  public resource[] resources_waiting;
  public DeadLock(int number, Thread last){
    size = number;
    waiters = new Thread[number];
    resources_held = new resource[number];
    resources_waiting = new resource[number];
    Hashtable WRTable = resource.getWRTable();
    resource currentHR = (resource)WRTable.get(last);
    for (int i=0;i<size;i++) {
      Thread currentT = currentHR.getOwner();
      resource currentWR=(resource)WRTable.get(currentT);
      waiters[i]=currentT;
      resources_held[i]=currentHR;
      resources_waiting[i]=currentWR;
      currentHR = currentWR;
  }}
  public int DeadlockPrint(){
    for (int i=0; i<size; i++) {
      System.out.println(``For deadlocked thread ''
                  +i+`` :'');
      System.out.println(``Current Thread ID is: ''
                  +waiters[i].getId());
      System.out.println(``Resource Held is: ''
                  +resources_held[i].getId());
      System.out.println(``Resource Waiting is: ''
                  +resources_waiting[i].getId());
    }
    return size;
}}
```

6.6 A bank transfer deadlock example using general resources for Chapter 4

```
import java.util.Hashtable;

class Account extends resource{
  private int value;
  public String type;
  public Account(int v, String t) {
    value = v;
    type = t;
  }
  public void transfer(Account to,int amount){
    this.request();
    try {Thread.sleep(200);} catch (Exception ee){}
    to.request();
    if (value >= amount) {
      to.value = to.value + amount;
      value = value-amount;
    }
    to.release();
    this.release();
}}

class S2C_Transfer implements Runnable {
  private Account a1,a2;
  private int amount;
  public S2C_Transfer(Account a1,Account a2,int amount){
    this.a1=a1;
    this.a2=a2;
    this.amount=amount;
  }
  public void run(){
    a1.transfer(a2,amount);
}}

class C2S_Transfer implements Runnable {
  private Account a1,a2;
  private int amount;
  public C2S_Transfer(Account a1,Account a2,int amount){
    this.a1=a1;
    this.a2=a2;
    this.amount=amount;
  }
  public void run(){
    boolean successful = false;
    while (!successful){
      try {
        a1.transfer(a2,amount);
        successful = true;
      }catch (DeadLock e) {
        System.out.println(``Caught an exception!'');
        a1.release();
        synchronized (clients.lock) {
            clients.resolved++;
            clients.lock.notify();
        }
        try {Thread.sleep(200);} catch (Exception ee){}
}}}}
public class clients{
  static final int N_TEL = 2;
  static Thread[] clients_threads = new Thread[N_TEL];
  static DeadLock[] currentDeadlocks = new DeadLock[N_TEL];
  static int resolved = 0;
  static Object lock = new Object();
  private int id = 0;
  public clients (int id) {this.id=id;}
  public static void main(String[] args) throws Exception {
    int number_of_deadlocks = 0;
    Account sa = new Account(1500,``saving'');
    Account ch = new Account(1000,``checking'');
    int s2c = 500;
    int c2s = 600;
    S2C_Transfer trans_1 = new S2C_Transfer(sa,ch,s2c);
    C2S_Transfer trans_2 = new C2S_Transfer(ch,sa,c2s);
    clients_threads[0]=new Thread(trans_2,``C2S'');
```

```
clients_threads[1]=new Thread(trans_1 ,''S2C'');
clients_threads[0].start();
clients_threads[1].start();

Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
int deadlocked_threads = 0;
boolean cont = true;
  while (cont) {
    try {
      Thread.sleep(5000);
    }catch (InterruptedException e) {
       count=false;
    }

  /* Acquiring the lock resource.l to protect WFG,
   exploiting a O(n) cycle detection method to find
   deadlocks, setting number_of_deadlocks to be the
   number of deadlocks found, constructing exceptions
   for the found deadlocks, and storing the exceptions
   in the array: currentDeadlocks. Details omitted */

  ///////////////////////////////////////////////
  //Following code serves as a deadlock resolver
  //performing deadlock delegation
   /////////////////////////////////////////////
  if (number_of_deadlocks > 0) {
     for (int i=0; i<number_of_deadlocks; i++) {
        DeadLock e=currentDeadlocks[i];
        e.DeadlockPrint();
      for (int j=0;j<e.size;j++)
```

```
                if (e.waiters[j].getName().equals(``C2S'')){
                    e.resources_waiting[j].setThrower(e.waiters[
                     j],e);
                    break;
        }}}
        if (number_of_deadlocks >0) {
            synchronized (lock) {
            while (resolved < number_of_deadlocks)
                lock.wait();
            resolved = 0;
}}}}}
```

References

[1] Rakesh Agrawal, Michael J. Carey, and David J. DeWitt. Deadlock detection is cheap. *SIGMOD Rec.*, 13(2):19–34, 1983.

[2] C. Artho. *Finding faults in multi-threaded programs*. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[3] William M. Bolstad. *Introduction to Bayesian Statistics*. John Wiley, 2004.

[4] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 125, Washington, DC, USA, 2001. IEEE Computer Society.

[5] George Casella and Roger L. Berger, editors. *Statistical Inference*. Duxbury Press, CA, USA, 1990.

[6] K.M. Chandy. A survey of analytic models of roll-back and recovery strategies. *IEEE Computer*, 8(5):40–47, May 1975.

[7] Ing-Ray Chen. Stochastic Petri net analysis of deadlock detection algorithms in transaction database systems with dynamic locking. *The Computer Journal*, 38(9):717–733, September 1995.

[8] W. N. Chin. Some comments on "deadlock detection is cheap" in SIGMOD record Jan. 83. *SIGMOD Rec.*, 14(1):61–63, 1983.

[9] Junghoo Cho and Hector Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Trans. Database Syst.*, 28(4):390–426, 2003.

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

[11] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Trans. Computers*, 31(6):531–540, 1982.

[12] J. Rodrigues Dias. New approximate solutions per unit of time for periodically checked systems with different lifetime distributions. *Journal of Applied Mathematics and Decision Sciences*, 2006:Article ID 34506, 11 pages, 2006. doi:10.1155/JAMDS/2006/34506.

[13] Edsger W. Dijkstra. Co-operating sequential processes. In *Programming Languages*, pages 43–112. F. Grnuys, Ed., Academic Press, New York, NY, USA, 1968.

[14] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[15] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. Circulated privately, 1977.

[16] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, 2002.

[17] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[18] Erol Gelenbe and Marisela Hernández. Optimum checkpoints with age dependent failures. *Acta Informatica*, 27(6):519–531, May 1990.

[19] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[20] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[21] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[22] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[23] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[24] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

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

[26] Phil Koopman. Elements of the self-healing system problem space. In *Workshop on Architecting Dependable Systems / WADS03*, May 2003.

[27] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[28] Doug Lea. *Concurrent Programming in Java: Design Principles and Pattern*. Addison-Wesley, Reading, Mass., 1997.

[29] Gertrude Neuman Levine. Defining deadlock. *SIGOPS Oper. Syst. Rev.*, 37(1):54–64, 2003.

[30] Gertrude Neuman Levine. The classification of deadlock prevention and avoidance is erroneous. *SIGOPS Oper. Syst. Rev.*, 39(2):47–50, 2005.

[31] Sheng Liang and Deepa Viswanathan. Comprehensive profiling support in the Java virtual machine. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS ’99)*, pages 229–240, 1999.

[32] Yibei Ling, Shigang Chen, and Cho-Yu Jason Chiang. On optimal deadlock detection scheduling. *IEEE Transations On Computers*, 55(9):1178–1187, September 2006.

[33] Yibei Ling, Jie Mi, and Xiaola Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Transations On Computers*, 50(7):699–708, July 2001.

[34] Hanan Luss and Zvi Kander. Inspection policies when duration of checkings is non-negligible. *Operational Research Quarterly (1970-1977)*, 25(2):299–309, Jun., 1974.

[35] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *AAAI '99*, pages 541–548, Menlo Park, CA, USA, 1999.

[36] J. F. Meyer. Performability evaluation: where it is and what lies ahead. In *IPDS '95: Proceedings of the International Computer Performance and Dependability Symposium*, pages 334–343, Washington, DC, USA, 1995. IEEE Computer Society.

[37] Ali Mili. Towards a theory of forward error recovery. *IEEE Trans. Softw. Eng.*, 11(8):735–748, 1985.

[38] K.E. Murphy, C.M. Carter, and S.O. Brown. The exponential distribution: the good, the bad and the ugly. a practical guide to its implementation. In *Proceedings of Reliability and Maintainability Symposium*, pages 550–555, 2002.

[39] T. Nakagawa and K. Yasui. Approximate calculation of optimal inspection times. *The Journal of the Operational Research Society*, 31(9):851–853, Sep., 1980.

[40] R.E.Barlow, L.C.Hunter, and F.Proschan. Optimum checking procedures. *Journal of Society for industrial and Applied Mathematics*, 11(4):1078–1095, 1963.

[41] Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.*, 14(4):431–477, 2005.

[42] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[43] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite web services. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS)*, pages 167–176, Florence, Italy, 2003.

[44] A.P.A. van Moorsel and K. Wolter. Analysis of restart mechanisms in software systems. *IEEE Transactions on Software Engineering*, 32(8):547–558, August 2006.

[45] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[46] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-based avoidance of priority inversion for Java. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 529–538, Washington, DC, USA, 2004. IEEE Computer Society.

[47] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005*, July 2005.

[48] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[49] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.

[50] Fancong Zeng. Exploiting runtime exceptions and static analyses to detect deadlock in multithreaded Java programs. *Ph.D. qualification talk presented at Department of Computer Science at Rutgers university*, August 2002.

[51] Fancong Zeng. Deadlock resolution via exceptions for dependable Java applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, June 2003.

[52] Fancong Zeng and Michael L. Littman. A decision-theoretic approach to scheduling deadlock detection for Java. *DCS-TR-592*, December 2005.

Curriculum Vita

EDUCATIONAL EXPERIENCES:

09/1999–present Ph.D. Candidate
Department of Computer Science
Rutgers University (thesis advisor: Michael L. Littman)

09/1998–05/1999 Ph.D. Student
Department of Computer Science
Florida International University

09/1996–07/1998 Teaching Assistant/Instructor
Department of Computer Science
Nanjing University

09/1993–07/1996 M.S. Student (M.S. in Computer Science, 1996)
Department of Computer Science
Nanjing University

09/1989–07/1993 B.S. Student (B.S. in Computer Science, 1993)
Special Class for Gifted Young
Nanjing University

SELECTED PUBLICATIONS:

1. Fancong Zeng and Michael L. Littman: ”A Decision-theoretic Approach to Scheduling Deadlock Detection for Java”, DCS-TR-592, Rutgers University (2005)

2. Fancong Zeng: ”Deadlock Resolution via Exceptions for Dependable Java Applications”, DSN 2003: 731-740 (2003)

3. Xudong He, Fancong Zeng, and Yi Deng: ”Specifying Software Architectural Connectors in SAM”, SEKE99: 144-151 (1999)

4. Manwu Xu, Jianfeng Lu, Fancong Zeng, and Jingwen Dai: ”Agent Language NUML and Its Reduction Implementation Model Based on HOpi”, SIGPLAN Notices 29(5): 41-48 (1994)