Speculative Lock Acquisition for
Fault-Tolerant Distributed Systems

Tal Zamir
Speculative Lock Acquisition for
Fault-Tolerant Distributed Systems

Research Thesis

Submitted in Partial Fulfillment of the Requirements
For the Degree of Master of Science
in Computer Science

Tal Zamir

Submitted to the Senate of the Technion — Israel Institute of Technology

AV 5768 HAIFA AUGUST 2008
The Research Thesis Was Done Under The Supervision of
Prof. Assaf Schuster and Dr. Michael Factor in the Department of Computer Science

Acknowledgment

I would like to thank Prof. Assaf Schuster and Dr. Michael Factor for their constant feedback and guidance throughout the course of the research. Their insights and academic experience proved to be invaluable.

I would also like to thank Mr. Konstantin Shagin, who dedicated countless hours and single-handedly created an impressive state-of-the-art distributed framework, on which this research had been conducted.
I would like to thank my beloved parents, Zehava and Israel Zamir, for their endless support and encouragement throughout the years.
Contents

Abstract ........................................ 1

List of Symbols .................................. 3

1 Introduction .................................... 5
   1.1 Motivation .................................... 5
   1.2 Speculative Lock Acquisition ................. 7

2 Speculative Lock Acquisition .................. 11
   2.1 Data Conflict Detection ........................ 11
   2.2 Fault-Tolerance Support ......................... 12
   2.3 Speculative Deadlocks .......................... 13
      2.3.1 Timeout-Based Speculative Deadlock Detection 15
      2.3.2 Message-Based Speculative Deadlock Detection 15
   2.4 Speculation Preclusion ......................... 17

3 Speculative Lock Acquisition in JavaSplit .... 19
   3.1 Implementing Data Conflict Detection .......... 20
      3.1.1 Monitoring Local Read Accesses .......... 21
      3.1.2 Monitoring Remote Write Accesses .......... 22
      3.1.3 Invalidation-Based Conflict Detection .... 22
   3.2 Employment of Fault-Tolerance Capabilities .... 24
   3.3 Conditional Waiting for Speculative Locking .. 25
   3.4 Correctness Overview .......................... 27
4 Performance Evaluation

4.1 Evaluation Goals .................................................. 31
4.2 Theoretical Analysis ............................................... 32
  4.2.1 Throughput on a Bare JVM .................................... 33
  4.2.2 Throughput on JavaSplit without Speculative Locking .... 34
  4.2.3 Throughput on JavaSplit with Speculative Locking ....... 34
  4.2.4 Speculation-Optimal Properties ............................... 37
4.3 Application Performance Results ................................. 38
  4.3.1 SPECjbb* .................................................... 38
  4.3.2 TSP ......................................................... 42
  4.3.3 Hashtable .................................................... 43
4.4 Micro-Benchmark Results ........................................... 44
  4.4.1 Network Latency .............................................. 44
  4.4.2 Data Conflicts ............................................... 46
  4.4.3 Read Access Monitoring ...................................... 48
  4.4.4 Speculative Deadlocks ...................................... 50
4.5 Performance Evaluation Summary ................................ 51

5 Related Work ......................................................... 53

6 Future Work .......................................................... 57
  6.1 Passing Lock Ownership in Speculative Mode ................ 57
  6.2 Predictive Data Push ............................................. 58
  6.3 Instrumentation Optimizations .................................. 58

7 Conclusion ............................................................. 63

References ............................................................... 63

Hebrew Abstract ......................................................... i
List of Figures

2.1 An example of a speculative deadlock involving two processes .......................... 14
2.2 An example of a speculative deadlock involving multiple processes .................. 16

3.1 An example of the HLRC implementation of JavaSplit ................................. 21
3.2 Pseudo-code inserted before a read access .................................................. 22
3.3 A single speculative lock acquisition possibly resulting in data conflict .......... 23
3.4 Conditional wait loop ................................................................................. 27

4.1 Model application - worker thread code ...................................................... 32
4.2 Flow of a blocking lock acquisition ............................................................ 35
4.3 Throughput and worker breakdown for SPECjbb* (low acq. rate) .................. 40
4.4 Throughput and worker breakdown for SPECjbb* (high acq. rate) ............... 41
4.5 TSP throughput and worker block time breakdown ....................................... 43
4.6 Hashtable throughput ................................................................................. 45
4.7 Average lock acquisition latency with varying network latencies ................ 45
4.8 SPECjbb* throughput with varying data conflict probabilities ..................... 46
4.9 Varying preemptive checkpointing frequency .............................................. 47
4.10 Speculative deadlock probability as a function of speculation quota .......... 50
4.11 SPECjbb* throughput as a function of speculation quota ............................ 51

6.1 Passing lock ownership in speculative mode ............................................... 59
6.2 Predictive data push .................................................................................. 60
Abstract

The overhead of lock acquisitions in distributed runtime environments often becomes a significant performance bottleneck. Lock acquisitions are expensive because they may result in communication with remote processes, which is performed on the critical path of local application execution.

In many cases, system consistency is not violated even if a process does not block-waiting until the lock is acquired, but rather continues its execution as if the lock acquisition operation had already been completed.

While this observation had been exploited to increase performance of shared memory multi-processor systems at the micro-architecture level, it had never been applied to the domain of fault-tolerant distributed systems.

In this work, we apply the concept of speculative lock acquisition to the domain of fault-tolerant distributed systems, eliminating traditional lock acquisition overheads. Moreover, applying speculative locking allows applications running on these systems to achieve the highest possible degree of parallelism, by enabling concurrent execution of critical sections protected by the same lock.

The proposed scheme relies on the ability of the underlying system to checkpoint and reestablish a process state. This capability is common in existing distributed software shared memory systems with fault-tolerance support. This makes speculative locking highly attractive for these systems.

We evaluate the performance of the speculative lock acquisition technique by implementing it in JavaSplit, an existing fault-tolerant distributed shared memory runtime for Java.
List of Symbols

Abbreviations

STM  Software Transactional Memory
JVM  Java Virtual Machine
OS   Operating System
RC   Release Consistency
LRC  Lazy Release Consistency
HLRC Home-Based Lazy Release Consistency
HB   Happens-Before
$op_1 \rightarrow op_2$  $op_1$ Happened-Before $op_2$
JS   JavaSplit
SL   Speculative Locking
SMP  Symmetric Multiprocessing
RAM  Random Access Memory
JGF  Java Grande Forum
GHz  Gigahertz
GB   Gigabyte
ms   Milliseconds

Java Instructions

GETFIELD  Fetch field from object
AALOAD    Load reference from array
IALOAD    Load int from array
DALOAD    Load double from array
Performance Analysis Annotations (for section 4.2)

$L$  
Number of shared objects and their associated locks

$W$  
Number of worker threads

$P_{\text{write}}$  
Write probability (per loop iteration)

$P_{\text{fail}}$  
Speculation failure probability

$P_{\text{conflict}}$  
Data conflict probability

$P_{\text{deadlock}}$  
Speculative deadlock probability

$T_{\text{chk}}$  
Average checkpoint time

$T_{\text{roll}}$  
Average rollback time

$T_{\text{net}}$  
Network round-trip delay

$M$  
Maximal number of speculations per speculative session (quota)

$S$  
Actual number of speculations in a speculative session

$F$  
Iteration frequency (throughput)

$T_1$  
Computation time with a bare JVM

$T_{1,\text{JS}}$  
Computation time with JS bytecode instrumentation

$T_{1,\text{SL}}$  
Computation time with JS and SL bytecode instrumentation

$T_{\text{compute}}$  
Computation time in a loop iteration

$Q$  
Average lock requester queue size
Chapter 1

Introduction

1.1 Motivation

Modern networks, composed of cheap multiprocessor commodity workstations, become an attractive computation alternative, competing with traditional custom-made monolithic multiprocessor systems, both in price and in scalability. Multithreaded applications can be distributed on such networks using an abstraction of a global shared memory, which is transparent to the application developer. The developer assumes the traditional shared memory model while a runtime middleware seamlessly manages the distribution of the application over several machines. To provide this transparent service, these middlewares intercept the application’s memory read/write operations and, if necessary, send corresponding requests to remote machines. The application doesn’t have to worry about synchronizing data between machines participating in the application’s execution. Thus, software distributed shared memory systems enable leveraging a cluster of commodity computers for high-performance computing.

Many variations of such systems had already been implemented [5, 11, 13, 16, 20, 25, 29, 32, 34]. To reduce the overhead required to keep the memory coherent, most modern systems implement some kind of relaxed memory consistency model, such as Release Consistency [7]. Release Consistency provides the application developer with a reasonable looser shared memory model in which the memory view of a thread depends on synchronization operations. This model allows the underlying runtime to reduce the number of messages which are passed between the machines, thereby improving
CHAPTER 1. INTRODUCTION

Fault-tolerant software distributed shared memory systems had also been suggested [5, 14, 15, 23, 30]. Fault-tolerance is usually achieved by using some kind of checkpointing mechanism to occasionally checkpoint the state of application threads. Upon failure, the runtime restores the failed application process to its latest checkpoint while keeping the distributed system state consistent.

Parallel applications running on these systems employ locks to gain exclusive access to shared data and resources. Since application processes must cooperate in order to ensure that the lock ownership is exclusive, the overhead of lock acquisition is relatively high. Consequently, a process trying to acquire a lock may block for a non-negligible period of time which can be especially high when several processes compete to acquire the same lock. The overhead of lock operations frequently becomes a performance bottleneck preventing parallel applications from achieving the desired degree of scalability.

Application programmers tend to protect access to multiple data elements by a single lock. Hence, processes entering critical sections protected by the same lock do not necessarily access the same data. Consequently, in many cases, processes do not have to perform a blocking lock acquisition to ensure the integrity of the execution. Integrity would not be violated even if processes were allowed to execute critical sections protected by the same lock simultaneously. Moreover, even when the processes access the same data element, the correctness of the application is affected only if one of the processes modifies the data.

There have been several proposals to exploit the above observation at the micro-architecture level [22, 24, 26]. These proposals describe hardware mechanisms that can be incorporated in multiprocessor design in order to detect and take advantage of parallelism in multithreaded applications. The threads optimistically execute a critical section without taking exclusive ownership of the associated lock. In case a data collision is detected, the conflicting threads that are not the rightful owners of the lock roll back to the pre-acquisition state. The advances in hardware support for thread-level speculation [4, 8, 18] greatly facilitate restoration of thread state and thus increase the applicability of speculative locking to multiprocessors.

Other proposals [1, 2] show how to remove unnecessary synchronization using static analysis. However, these static analysis techniques can only remove synchronization operations where there is no possible execution path where the synchronization is needed. Therefore, these techniques can
not improve the performance of common scenarios in which lock contention is rare, yet possible.

Work on Software Transactional Memory [28] suggests to implement synchronization operations as transactions and to execute these transactions using an optimistic algorithm which is built into the consistency model. It avoids blocking on critical sections by optimistically speculating that no real data conflict will occur during the transaction and performs an abort operation in case the speculation fails, using a transaction log for recovery. However, STM requires the application developer to modify its application to support STM semantics, which could make it less attractive for existing applications.

Lock Prefetching [12] is a technique which sends out lock acquisition messages ahead of time, so that the lock will arrive when it is needed, thereby hiding the lock acquisition overhead. However, this method requires that the programmer (or compiler) will statically insert annotations, telling the runtime when to prefetch the lock. This approach can also be very sensitive to variations in network latency and to changes in the executed application.

A better approach, called Lock Acquirer Predictor [27] provides a runtime predictor which tries to predict the next acquirer for a lock and then pre-fetches the lock, along with its memory consistency data. However, these techniques cannot fully exploit implicit lock parallelism, as they cannot allow multiple threads to concurrently acquire the same lock, even if there is no data conflict.

1.2 Speculative Lock Acquisition

In this work we propose to apply the idea of optimistic lock acquisition, or speculative locking, in the domain of software shared memory. In general, the benefit from employment of speculative locking in software shared memory is much more significant than in multiprocessors.

First, the main use of software shared memory is in distributed environments in which the overhead of interprocess communication is much higher than inter-thread interaction in multiprocessor systems. Therefore, the benefit from removing communication-inducing lock acquisitions from the critical path of process execution in software shared memory is higher than in multiprocessors.

Second, the number of available processors in software shared memory systems is potentially larger than that in commonly used multiprocessors. Hence, achieving a higher level of parallelism and scalability in software shared memory systems may yield a higher computational power.
Finally, the flexibility of software shared memory systems allows implementation of more elaborate data conflict detection and checkpointing mechanisms.

The proposed method relies on fault-tolerance capabilities of a software shared memory system, namely, the ability to checkpoint and reestablish the state of an individual process without violating application consistency. There has been a considerable amount of research concerning transparent fault-tolerance in software shared memory [6, 14, 15, 23, 30]. Our speculative locking scheme can easily take advantage of an existing failure resilience mechanism. The efficiency of speculative locking in the presence of frequent data conflicts depends on the overhead of process rollback as well as on the timing of process checkpoints.

In order to detect data conflicts during concurrent execution of critical sections associated with the same lock, the proposed method requires monitoring local read and write operations. Since software shared memory systems rely on this capability to implement their memory coherency protocols we can assume it is generally available. The common methods to monitor data access are OS-level access protection and automatic program instrumentation. In most cases, the monitoring already implemented in a software shared memory system is sufficient for the purpose of data conflict detection. The granularity of data conflict detection is the same as the granularity of data access monitoring in a given system.

Implementation of a general speculative locking scheme in software shared memory uncovers design issues that do not arise in the multiprocessor domain. A vivid example of such an issue concerns preventing a process from affecting the global state between speculative acquisition of a lock and the actual receipt of lock ownership. In multiprocessor implementation, a speculating thread delays flushing its cache to the main memory. However, in a software shared memory system there are no analogous hardware mechanisms. In order to ensure a process does not affect the global state during speculation, it is prohibited from performing certain operations, including passing ownership of any lock to another process. This, however, can result in a situation which we call a speculative deadlock.

In its simplest form, a speculative deadlock manifests itself when there are two processes, each of which speculatively acquires a lock that is owned by the other process. Due to the aforementioned restriction, neither process can release the lock it owns for the benefit of the other process. Consequently, neither process can properly acquire the requested lock, which prevents normal progress of
1.2. SPECULATIVE LOCK ACQUISITION

their execution. In this work we present methods to detect and resolve speculative deadlocks.

We implement the proposed speculative locking scheme in JavaSplit, a fault-tolerant distributed runtime for multithreaded Java applications [5, 6]. JavaSplit works by rewriting the bytecode of a given application, transforming it into a distributed application that incorporates all the runtime logic. Each runtime node carries out its part of the resulting distributed computation using nothing but its local standard (unmodified) Java Virtual Machine (JVM). To support the Java memory model, JavaSplit incorporates an object-based software shared memory.

Both access monitoring and checkpointing are implemented in JavaSplit through bytecode instrumentation and neither require hardware support nor modification of the local JVM. Data conflict detection for the purpose of speculative locking is incorporated into the existing shared data access monitoring logic. The fault-tolerance scheme allows each process (that represents a thread in the original application) to checkpoint and reestablish its state independently of the other processes. The latter property makes JavaSplit an attractive testbed for implementing speculative locking.

The main contributions of this work are the following. First, we present the idea of using speculative locking to improve performance and scalability of a fault-tolerant software shared memory system. Second, we describe a generic technique for implementing speculative locking in the above environment. Third, we discuss design and implementation issues that are specific to this domain. Finally, we evaluate the proposed approach in an existing fault-tolerant software shared memory system and show that speculative locking can provide a significant performance improvement.

The structure of the rest of this work is as follows. Chapter 2 presents the proposed speculative lock acquisition scheme. Chapter 3 describes its implementation in JavaSplit. Chapter 4 presents the performance evaluation of speculative locking in JavaSplit. Related work is discussed in Chapter 5. Chapter 6 presents several directions for additional research. In Chapter 7 we conclude.
Chapter 2

Speculative Lock Acquisition

In this chapter, we describe the general technique of speculative lock acquisition, applicable to many fault-tolerant distributed systems.

The basic idea of speculative locking is replacement of the original blocking lock acquisition procedure with a non-blocking lock acquisition. After requesting the lock, the process optimistically continues its execution as if the lock had already been acquired. If, at some point, a data conflict is detected, the process rolls back to some state preceding the speculative acquisition.

Until it can be determined whether there has been a data conflict, a process that has speculatively acquired a lock is considered to be in *speculative mode*. We will refer to a process in speculative mode as a *speculating process*.

2.1 Data Conflict Detection

A data conflict is a situation in which a speculating process reads outdated contents of a data element as a result of not waiting for completion of the lock acquisition procedure. In order to detect a data conflict, a speculating process monitors its data accesses. When the lock ownership is properly acquired, a speculating process compares the list of speculatively read data elements against the modifications it should have observed. If a match is found, the process rolls back.

Distributed shared memory systems achieve inter-process data consistency by monitoring shared
data accesses. This is usually accomplished by OS-level access protection or by automatic instrumentation of application code or binaries. Consequently, the access monitoring mechanisms needed by the proposed speculative locking scheme are present in most systems and require only slight adjustment to support speculative locking. This greatly facilitates the implementation of data conflict detection in distributed shared memory systems.

2.2 Fault-Tolerance Support

The proposed speculative locking scheme relies on rollback capabilities of a distributed software shared memory, capabilities which are inherent in fault-tolerant systems. The underlying system should allow the current state of any process to be lost or discarded without compromising the integrity of the execution. A process is recovered to a state consistent with the rest of the system.

In the presence of data conflicts, the benefit from the proposed speculative locking protocol depends on the efficiency of system recovery from speculation failure. In some fault-tolerant systems, a failure of a process or a node may require rollbacks of additional processes, in order to preserve system consistency. In general and for purposes of speculative locking in particular, it is highly desirable that a rollback of a failed process has little effect on other processes, preventing a "domino effect".

There are two restrictions imposed on speculating processes. First, a speculating process must be able to rollback to a state preceding the speculative lock acquisition. Otherwise, the consistency of the execution may be violated if a data conflict is detected. In systems with a fault-tolerance scheme that maintains only the most recent checkpoint per process, this restriction implies that a speculating process cannot be checkpointed. Otherwise, a process will not be able to rollback past a lock acquisition. Second, while in speculative mode, a process must not affect the state of the system in a way that would make the system inconsistent in the case it rolls back due to a data conflict. Since the results of a speculative execution may be incorrect when followed by discovery of a data conflict, they should not be exposed to other processes. It is possible to consider a speculative locking scheme in which a speculating process can affect the state of other processes. However, in such a scheme, a data conflict discovery would require rolling back all the processes affected by the speculating process. We feel that the overhead of a data conflict and of monitoring process
dependencies in such schemes exceeds the benefit from speculative locking. Moreover, since the number of concurrent node failures allowed by a particular fault-tolerance scheme may be limited, it is desirable to impose the above restriction.

The rollback operation required to support speculative locking is potentially much more lightweight than process recovery resulting from a node failure. In the case of a node failure, the entire process state is lost. However, when a process rolls back due to a data conflict, the latest state of both the process and the system logic is still available. This allows to employ an optimized rollback operation in the latter case. First, significant portions of the system logic may not need to be rolled back. Second, most data necessary to reestablish an older process state may be locally available. Finally, since the process is restored on the same node, its physical address does not change, and therefore remote requests arrive to the restored process without the delay associated with updating process whereabouts in a distributed directory service.

In checkpoint-based fault-tolerance schemes, a process can roll back only to a number of past checkpoints. When implementing speculative locking in such schemes, it may be desirable to increase the checkpointing frequency in order to reduce the amount of lost computation in the case of a rollback. There is an inherent tradeoff between the overall checkpointing overhead and the rollback penalty. Hence, the optimal checkpointing frequency is derived from checkpointing cost and rollback probability.

2.3 Speculative Deadlocks

In most distributed shared memory systems, the restrictions on speculating processes outlined in section 2.2 prevent a process from transferring lock ownership while in speculative mode. This may lead to a situation in which a process cannot exit the speculative mode because the lock it speculatively acquired is owned by another process that also indefinitely remains in speculative mode.

The simplest case of a speculative deadlock occurs when there are two processes each of which has speculatively acquired a lock owned by the other process. As a result, neither process can transfer lock ownership to the other process and therefore both processes remain in speculative mode forever. Figure 2.1 shows an example of such a scenario.
In general, any number of processes can be involved in a dependency loop analogous to the described above.

Note, that a process that *owns* a lock does not necessarily *hold* the lock, i.e., it is not necessarily inside a critical section protected by the lock. Therefore, speculative deadlock is not a traditional application-level deadlock and it would not have occurred in a system without speculation. Since exiting speculative mode is a precondition for process termination, an application cannot finish executing in the presence of an untreated speculative deadlock.

We suggest mechanisms for speculative deadlock prevention in section 2.4 and in section 6.1. These mechanisms can reduce the number of speculative deadlocks and, for some executions, to completely eliminate the speculative deadlock phenomena. However, in general, speculative deadlocks can still occur and therefore a mechanism for speculative deadlock detection is required.

Once a speculative deadlock is detected, it can be resolved by rolling back one of the processes in the dependency loop. After rolling back, the process is no longer in speculative mode and therefore it can handle the pending lock requests. After transferring lock ownership to the requesting processes,
2.3. SPECULATIVE DEADLOCKS

the process continues execution.

We will now examine several methods to detect speculative deadlocks.

2.3.1 Timeout-Based Speculative Deadlock Detection

The simplest way to detect speculative deadlocks is based on timeouts (a watchdog approach). When a process enters speculative mode, a timer is started. When the timer expires it is assumed that the process is involved in a speculative deadlock and therefore the process is rolled back. Naturally, the number of false alarms and detection latency greatly depend on the timeout value.

The main advantage of this approach is that it is very lightweight. There is almost no processing overhead for detecting speculative deadlocks using this approach. When the timer expires, a rollback is initiated. There is no need for any additional monitoring. Additionally, this approach does not require any inter-process communication. No messages are sent to detect a deadlock, reducing network load and process coordination.

However, if the selected timeout value is too short, it may lead to false deadlock detection, causing processes to perform expensive rollbacks for no reason. If the selected timeout value is too long, detection latency will be longer, increasing the wasted application time upon deadlock detection. Additionally, if all processes use the same timeout value, multiple processes in a dependency loop may detect a deadlock simultaneously, causing all of them to perform a rollback, even if the deadlock could have been resolved by a single rollback of one of the processes in the loop. Yet, progress is ensured, as after rolling back, the process transfers ownership of the requested locks.

To reduce the probability for multiple rollbacks per deadlock, each process may use a slightly different timeout value, based on its unique identifier. For example, the processes randomly choose a timeout value out of a predefined time range and use their identifiers as randomization seeds. The process with the minimal timeout will be the first to time-out and detect a deadlock. It will rollback and resolve the deadlock before other processes rollback.

2.3.2 Message-Based Speculative Deadlock Detection

A more complex but also more prompt method is sending probe messages whose goal is to detect loops in the speculative dependency graph. In most cases, the probe messages can be piggybacked
on existing protocol messages. Once a loop is detected, the process with the minimal identifier in the loop is rolled back.

Whenever a new dependency is created, \textit{i.e.}, a process requests a lock which cannot be transferred, deadlock detection messages will be propagated throughout the dependency graph. Each process adds its unique identifier before forwarding a detection message. When a process receives a detection message with its own identifier and given that it has the minimal identifier in the message's identifier list, it rolls back, resolving the deadlock.

In the example shown in figure 2.2, when process 4 speculatively requests a lock from process 5, which is also in speculative mode, a new edge in the dependency graph is created (annotated \(e\)). Speculative deadlock detection messages will be sent throughout the connected directed graph, leading to the rollback of process 1, which has the minimal unique identifier in the loop consisting of processes 1 to 5.

This mechanism is expensive in terms of the required communication and processing overhead. Each new edge in the dependency graph triggers the transmission of detection messages throughout the connected graph. Additionally, each process receiving a detection message has to perform some processing to handle the request and to forward it on. However, the mechanism ensures quicker detection of speculative deadlocks and a single rollback per speculative deadlock.

The detection timeout value in the time-based detection approach can be dynamically learned using the message-based approach. Thus, we can use message-based deadlock detection for a limited time to identify an average deadlock detection timeout.

Figure 2.2: An example of a speculative deadlock involving multiple processes
2.4 Speculation Preclusion

Under certain circumstances, speculative acquisition of a lock can have a negative impact on performance. We employ a number of runtime heuristics to detect such situations. When detected, the system acquires the lock in a non-speculative (blocking) fashion.

The most common case in which there is no benefit from speculation is speculative acquisition followed by a rollback. This occurs due to discovery of a data conflict or due to resolution of a speculative deadlock. To avoid repetitive rollbacks, we monitor the distribution of rollbacks associated with a particular lock. Based on these statistics, in each given case, the system makes a decision on whether it should perform a speculative acquisition.

A possible policy is to prevent speculative acquisition of a particular lock in the case a large portion of past speculations (or a number of most recent speculations) resulted in a rollback. Speculation on that lock may be disabled for a certain time period or for the next few acquisition operations.

Application processes with a high lock acquisition frequency may spend most of their execution time in speculative mode, rarely exiting it. This has two negative (and synergetic) effects. First, such processes are likely to become a part of speculative deadlock loops. Second, as mentioned in section 2.2, a large number of fault-tolerance schemes cannot checkpoint a speculating process. Consequently, a process that rarely exits speculative mode can rarely save its execution state and therefore may lose a significant portion of its computation due to a rollback.

In order to limit the time an application process spends in speculative mode, we restrict the number of speculative lock acquisitions in each speculative mode session. When the quota is exceeded, the locks are acquired non-speculatively. A similar policy is to preclude speculative lock acquisitions if a speculating process has been in speculative mode for a predefined amount of time.

If the selected speculative acquisition quota is too small, it may not take full advantage of the benefits of speculative lock acquisition, as the process will frequently perform non-speculative acquisitions. If the quota is too high, it may cause too many speculative deadlocks, reducing the overall system performance. For each execution, there exists an optimal quota which provides the optimal performance. This optimal quota can be dynamically detected by the system. The general concept of this dynamic quota detection suggests increasing the quota while there are no speculative deadlocks and decreasing it upon speculative deadlock detection.
Chapter 3

Speculative Lock Acquisition in JavaSplit

We have implemented the proposed speculative locking scheme in JavaSplit [5], a fault-tolerant distributed runtime for Java. JavaSplit is essentially a distributed Java Virtual Machine (JVM) that runs on a plurality of standard Java Virtual Machines. It executes multithreaded Java programs written for a single JVM on a collection of available interconnected workstations. JavaSplit transparently distributes the threads of a Java program among the processing nodes and maintains the consistency of objects accessible by multiple threads. The implemented memory model is Home-Based Lazy Release Consistency (HLRC) [33], which is compatible with the new Java Memory Model. The distributed object management scheme incorporates support for distributed synchronization, including Java-specific operations, e.g., wait and notify.

JavaSplit employs bytecode instrumentation to enable execution of a given application in a distributed environment. Calls to JavaSplit runtime handlers are weaved into the application bytecode to intercept events that are interesting in the context of distributed execution, e.g., synchronization, creation of new threads, and accesses to shared objects. In addition, the JavaSplit bytecode rewriter augments the program and a subset of JavaSplit runtime classes with thread checkpointing capabilities. Since JavaSplit runtime modules are also written in Java, the threads of the instrumented application can be executed on any standard JVM. Consequently, the portability of JavaSplit is
equivalent to the portability of a Java application.

JavaSplit is resilient to multiple and possibly concurrent node failures. As will be described in section 3.2, the employed fault-tolerance scheme is well-suited for integration of speculative locking.

3.1 Implementing Data Conflict Detection

To understand how data conflict detection is integrated into JavaSplit, we first describe its memory consistency model.

The Lazy Release Consistency (LRC) model uses lock acquire and release operations to define a partial order, called happened-before-1, denoted by → and defined as follows:

- If $op_1$ and $op_2$ are memory operations by the same process and $op_1$ occurs before $op_2$ in program order, then $op_1 \rightarrow op_2$.

- If $op_1$ is a release operation by process $p_1$ and $op_2$ is a consequent acquire of the same lock by process $p_2$, then $op_1 \rightarrow op_2$.

- If $op_1 \rightarrow op_2$ and $op_2 \rightarrow op_3$, then $op_1 \rightarrow op_3$.

LRC requires that memory operations preceding an operation $op$ in the happens-before-1 order are visible to the application thread when it completes $op$. As opposed to strict consistency models, this model lets the runtime propagate a thread’s shared memory modifications upon lock transfer, reducing the overhead required to keep the memory view consistent.

Upon lock ownership transfer, the releaser of the lock passes a list of write notices to the acquirer of the lock. Each write notice contains an identifier of a shared object which had been modified by a write operation which happened-before this lock acquisition operation. The sent list does not include shared memory units whose modifications are already familiar to the acquirer. The acquirer needs to invalidate the cached objects according to the received write notices. Upon first access to an invalidated object, the latest updated copy will be fetched from its home node.

The model is also home-based. Thus, the node on which the object is created manages the object’s master copy, the most up-to-date version of the object. The threads that need to use the object create local (cached) copies, based on the master copy. When transferring lock ownership
3.1. IMPLEMENTING DATA CONFLICT DETECTION

![Diagram of HLRC implementation of JavaSplit]

Figure 3.1: An example of the HLRC implementation of JavaSplit

between application threads, the modifications are flushed from the cached copies to the master copy.

A simple example of the memory consistency model is provided in figure 3.1.

The implementation of data conflict detection necessary to support speculative locking is tightly integrated into the distributed memory management logic. It consists of monitoring local read accesses, monitoring remote write accesses and identifying conflicts between the two access types.

3.1.1 Monitoring Local Read Accesses

In order to monitor data read accesses while in speculative mode, we instrument application read operations, namely accesses to object fields and accesses to array elements. In the original application bytecode, the former are identified by the GETFIELD instruction, while the latter are identified by array load instructions such as AALOAD, IALOAD and DALOAD. The main purpose of the inserted code is to mark objects and arrays whose contents had been read during speculative mode.
CHAPTER 3. SPECULATIVE LOCK ACQUISITION IN JAVASPLIT

Figure 3.2: Pseudo-code inserted before a read access

Figure 3.2 describes the inserted logic in pseudo-code. If the current thread is not in speculative mode, then the execution proceeds normally. Otherwise, if the object is not marked, it is marked and added to a list of speculatively read objects. The list is maintained to enable clearing all marks when exiting speculative mode. In the actual bytecode, the contents of the outer IF statement are encapsulated in a handler method, in order to minimize the size of the instrumented code.

3.1.2 Monitoring Remote Write Accesses

In implementations of LRC, including the protocol employed by JavaSplit, a list of data modifications, a.k.a. write notices, is passed along with lock ownership from one process to another. This list includes all remote modifications that must be observed by the recipient of lock ownership at the point of lock acquisition. Based on these write notices, a process invalidates local copies of the data elements (typically memory pages or objects) and then fetches their updated versions on demand.

When a process speculatively acquires a lock, it continues execution without waiting for the lock ownership accompanied by the write notices and, as a result, it does not perform the necessary invalidations. Consequently, a speculating process may read outdated data elements, which would have been invalidated should the lock acquisition have been performed without speculation. It follows that the list of write notices contains information about all the remote write accesses that may conflict with the read operations performed in speculative mode. Thus, no additional monitoring of remote write accesses is required to support speculative locking.

3.1.3 Invalidation-Based Conflict Detection

Data conflict detection is performed upon receipt of lock ownership. It is integrated into the logic that invalidates local copies of shared objects based on the received write notices. In the memory
3.1. IMPLEMENTING DATA CONFLICT DETECTION

Figure 3.3: A single speculative lock acquisition possibly resulting in data conflict.

consistency protocol of JavaSplit, a write notice consists of a globally unique ID of the modified object and a modification version. Upon receipt of a write notice, the local copy of an object is invalidated only if the local version is older than the received modification version. If the object is not invalidated based on a write notice, it means that a possible speculative read of the associated object does not violate memory consistency. Consequently, a data conflict may occur only for invalidated objects.

Upon object invalidation based on a received write notice, the conflict detection logic checks whether the local copy of the object is marked as speculatively read. If indeed this is the case, the current thread rolls back. Note that despite a decision to roll back there is a possibility that there was no actual data conflict. This is due to the possibility that a remote process that performed the write access and the local process accessed different fields of the object. In general, the frequency of false positives depends on the granularity of the memory coherency unit.

Figure 3.3 describes the flow of a single speculative lock acquisition, as implemented in JavaSplit.
3.2 Employment of Fault-Tolerance Capabilities

In JavaSplit, fault-tolerance is accomplished by checkpointing and replication. At certain execution points each application thread (executing in a separate process) independently checkpoints itself. A checkpoint of a thread does not include the application’s shared objects, i.e., objects that can be accessed by more than one thread. Shared objects and thread checkpoints are made persistent by replication in the volatile memory of several other nodes.

The key idea of JavaSplit’s fault tolerance scheme is to capture the states of the application threads in such a way as to guarantee that rollback of any single thread to its latest saved checkpoint does not violate system consistency. If this requirement is fulfilled, a failed thread recovers by simply restarting from its most recent checkpoint. To achieve this, JavaSplit exploits the fact that in the employed memory consistency protocol a thread may affect the data observed by remote threads only at lock ownership transfer. By checkpointing a thread when it is about to transfer lock ownership to another thread, JavaSplit ensures that the most recent thread checkpoints are inter-consistent. JavaSplit assumes that threads communicate only through shared memory and do not communicate by using other methods, such as files or sockets.

The fault-tolerance scheme outlined above allows JavaSplit to keep only a single checkpoint per thread. In addition, recovery never results in rollback of non-failing nodes. Moreover, neither failure-free execution nor recovery from a failure require global cooperation of nodes.

The ability to rollback a thread independently of other threads makes JavaSplit well-suited for accommodating the proposed speculative locking mechanism, which may induce rollback frequency higher than the expected frequency of node failures. Furthermore, the employed fault-tolerance scheme does not prohibit checkpoints which are not induced by lock transfer operations. This allows taking additional checkpoints to minimize the amount of computation lost due to data conflicts. Although the number of concurrent node failures tolerated by JavaSplit depends on the number of maintained thread checkpoint replicas, the number of concurrent speculation-induced rollbacks is unlimited.

In general, the rollback operation required by speculation locking is more lightweight than the recovery procedure performed in the case of a node failure. Each thread locally stores its latest checkpoint and therefore a rollback does not require fetching the checkpoint from the volatile memory.
3.3. CONDITIONAL WAITING FOR SPECULATIVE LOCKING

of a remote machine. Moreover, unlike the case of node failure, the thread that performs a rollback preserves its network address. Therefore, it continues receiving remote requests without the delay of updating the global directory service. Finally, the local state of the runtime modules is not lost as in the case of a node failure. Consequently, only the portion of JavaSplit runtime data that is closely related to the thread execution state needs to be reconstructed. For example, the lock management module is not affected by the rollback. Lock acquisition procedures continues normally despite the rollback and the set of locks considered to be owned by the current node does not change.

The two restrictions on the implementation of speculative locking described in section 2.2 preclude a process from passing lock ownership while in speculation mode. As described above, transfer of lock ownership requires a releaser process checkpoints itself. This contradicts the first restriction, because JavaSplit maintains only the latest checkpoint. Therefore, if a process checkpoints itself while in speculation mode, it will not be able to rollback past the speculative acquisition. This issue could be resolved by making a slight modification in JavaSplit fault-tolerance scheme to allow keeping several checkpoints per process. However, passing lock ownership while in speculation mode also contradicts the second restriction, as follows. In JavaSplit, before a process transfers lock ownership to another process, it flushes its modifications to the master copies of the associated shared objects, which makes the modifications visible to all other processes. By passing lock ownership while in speculative mode, a process would expose data that may be incorrect due to speculation.

The inability of a speculating process to transfer lock ownership results in the speculative deadlock phenomenon which is presented in section 2.3. The same section also describes how speculative deadlocks are resolved.

3.3 Conditional Waiting for Speculative Locking

Java provides the application developer with multiple thread synchronization mechanisms. The basic building blocks for these synchronization mechanisms are the acquire, release, wait and notify operations. A synchronized block in Java is implemented by a corresponding pair of acquire-release operations on the lock object which is used in the synchronized block. The wait/notify operations (conditional waiting) can only be called by the owner of the lock. The wait operation releases the lock and then puts the thread in a wait queue which is associated with the lock. A
CHAPTER 3. SPECULATIVE LOCK ACQUISITION IN JAVASPLIT

notify operation on that lock will wake up the first waiting thread in the queue, which will then try to re-acquire the lock.

In JavaSplit, each lock is assigned a home node, which is responsible for keeping track of the current lock owner. The lock’s home node is also the initial lock owner. The lock owner keeps the lock token which contains all the relevant data about the lock, including the lock requester queue and the wait queue, which are initially empty. This information, associated with each lock, is passed along with each lock ownership transfer.

Upon lock acquisition, the acquiring application thread is blocked and a lock ownership request is sent to the lock’s home node. The request is then forwarded to the current lock owner. The owner adds the identifier of the requesting thread to the end of the lock request queue. When the lock owner releases the lock, it transfers the lock to the next thread in the lock request queue, passing write notices along with the lock’s request and wait queues.

In order to perform wait or notify operations a thread must be the lock owner. A thread that performs the wait operation is inserted into the wait queue of a lock, releases the lock and blocks. Thus, the lock becomes ready for transfer to the next requester.

A notify request removes a thread from the wait queue and sends a notification to the notified thread. The notified thread continues its execution but must re-acquire the lock in order to proceed.

In non-speculative JavaSplit, the wait queue is passed along with the lock ownership. Consequently, the owner of the lock can always access the wait queue locally. However, if locks are acquired speculatively, the wait queue may not be available when conditional waiting operations are performed. This issue is resolved by logging the conditional waiting operations performed in the absence of the required wait queue and then applying the logged operations to the wait queue when it is received. Fortunately, the semantics of conditional waiting operations permit delaying their effect for an unbounded but finite time.

Multithreaded Java programs tend to encapsulate the wait operation in conditional loops. An example of this programming pattern is shown in Figure 3.4. In the presence of speculative locking the execution of the waiting thread is likely to result in a data conflict, as follows. The waiting thread speculatively acquires a lock and begins waiting. When a notification is received, it speculatively re-acquires the lock and, since the condition appears to be still false, it continues to wait. After the second lock acquisition is complete, a data conflict of the condition variable is detected and
the waiting thread needs to rollback. In order to prevent this scenario, lock acquisitions that are a part of a wait operation (such as the second lock acquisition in the example above) are performed without speculating. This simple policy significantly reduces the number of rollbacks in applications which employ conditional wait loops.

### 3.4 Correctness Overview

In this section, we provide a high-level correctness proof for speculative locking in JavaSplit. The proof is coupled to the memory consistency model implemented in JavaSplit, which is a variation of the lazy release consistency model, passing write notices along with lock ownership transfers. Similar correctness analysis can be performed for the integration of speculative locking in other suitable systems.

We focus on showing that speculative locking cannot cause a violation of the happens-before order and that speculative locking does not prevent any process from completing its execution.

**Claim 3.4.1** If a process enters speculative mode in a state which does not violate the happens-before order, it will also leave speculative mode in such a state.

**Proof:** Let us examine a process which enters speculative mode. Speculative mode begins when the process performs a speculative lock acquisition. While in speculative mode, a process is allowed to continue its computation, including access to local and shared objects and acquisition of additional locks.

Modifications made to shared objects are cached locally and are not visible to other processes. Additionally, the process is not allowed to publish a checkpoint of its speculative state. Speculative
mode ends when the process receives ownership of all its requested locks.

A speculating process may violate the happens-before relation only by reading the contents of shared objects. Let \( R \) be the set of shared object read operations performed by the process during this speculative mode session. Let \( W \) be the set of write notices received along with lock ownership transfers during this speculative mode session.

If none of the write notices in \( W \) invalidates an object read by an operation in \( R \), none of the operations in \( R \) violates the happens-before order, because none of the lock acquisitions in this session invalidated these objects. Thus, the process leaves speculative mode in a state which does not violate the happens-before order.

If a write notice \( w \) in \( W \) invalidates an object read by an operation \( r \) in \( R \), the process is currently performing an invalid execution, violating the happens-before order. However, this violation is local, because the process could not affect the state of other processes while in speculative mode or checkpoint its invalid state. This violation is detected when \( w \) is processed by the runtime and a rollback takes place, restoring the process to a previous state which does not violate the happens-before order. Thus, the process leaves speculative mode in a state that does not violate the happens-before order.

**Claim 3.4.2** For every \( i \), after leaving speculative mode for the \( i \)-th time, the process state does not violate the happens-before order.

**Proof:** Each process starts in non-speculative mode and its state does not violate the happens-before order, proving the claim for \( i = 0 \). Let us assume that after leaving speculative mode for the \( i \)-th time, the process state does not violate the happens-before order. Until the process enters speculative mode for the \((i + 1)\)-th time, it executes in non-speculative mode, not violating the happens-before order. According to claim 3.4.1, after the process leaves speculative mode for the \((i + 1)\)-th time, it is in a state which does not violate the happens-before order. Thus, by induction, the claim is correct.

**Claim 3.4.3** When the process completes execution, it is in a state that does not violate the happens-before order.

**Proof:** A process cannot terminate in speculative mode. This is ensured by the runtime's speculative locking mechanism. Thus, if the process reaches the last application instruction in speculative mode,
3.4. CORRECTNESS OVERVIEW

the runtime will wait until speculative mode is over, before announcing that the process terminated. Let us examine the last speculative mode session of a process. According to claim 3.4.2, after leaving speculative mode for the last time, the process does not violate the happens-before order. If there are any additional instructions after leaving speculative mode for the last time, they are performed in non-speculative mode, which cannot violate the happens-before order.

**Claim 3.4.4** A process does not stay in speculative mode forever.

**Proof:** Our time-based speculative deadlock detection mechanism ensures that after a maximal period of time in speculative mode, the process rolls back, assuming a speculative deadlock was created. This ensures that the process eventually leaves speculative mode.

**Claim 3.4.5** If an application completes its execution in a system which does not perform any speculative lock acquisitions, it will also complete its execution in a system which performs speculative lock acquisitions.

**Proof:** According to claim 3.4.4, every process eventually leaves speculative mode. We need to show that application forward progress is ensured. There are three ways to leave speculative mode.

First, speculative mode may end successfully, which means that the application continues its execution normally and can checkpoint its state, achieving forward progress.

Second, speculative mode may end because of a speculation failure, which means that the application had processed a write notice that invalidated some speculatively-read object. To avoid a state of repeatedly entering speculative mode and failing, the runtime avoids additional speculative lock acquisition for locks which constantly fail to be acquired speculatively, ensuring forward progress.

Last, speculative mode may end because a speculative deadlock was detected. The process is rolled back to a non-speculative state. Before continuing execution, the runtime ensures that the process handles all its pending incoming requests, i.e., transfers locks to the requesting processes. Handling pending requests ensures that no dependencies on this process exist, breaking the dependency loop and ensuring forward progress.
Chapter 4

Performance Evaluation

4.1 Evaluation Goals

This chapter attempts to answer the following questions:

- Can speculative locking improve the performance of fault-tolerant shared memory systems?
  - Can speculative locking provide higher levels of concurrency?
  - Can speculative locking eliminate the lock acquisition overhead?
- Is speculative locking scalable?
- Which applications can benefit from speculative locking? Which applications cannot?
- How does application, environmental and system properties effect the performance of speculative locking?
- What is the overhead of the speculative locking protocol?
- How can we further improve the performance of speculative locking?

To address the above questions in a system independent fashion, we present an analytical model that allows to estimate the performance gain from employment of speculative locking. Although this model is limited to applications with periodic lock access patterns, it is generic enough to roughly predict performance of a large variety of existing programs.
... 
public void run() 
{
    while(...) 
    {
        DataObject dataObject = selectRandomObject();
        synchronized(dataObject) 
        { 
            dataObject.read();
            if(shouldWrite()) 
            {
                dataObject.write();
            }
            compute(T1);
        }
        compute(T2);
    }
... 

Figure 4.1: Model application - worker thread code

Our evaluation environment is a cluster of 15 commodity Windows XP SP2 desktop machines with Intel Pentium dual-core 1.6 GHz processor and 2GB of RAM each, interconnected by 1 Gbit Ethernet, running Sun Java 1.6 JVM. The network round-trip time for this cluster is ranging between 0.2ms and 0.6ms.

4.2 Theoretical Analysis

For the purpose of theoretical performance analysis, we define a model of a lock-intensive multi-threaded application. Partial Java code for a worker thread in this application is shown in figure 4.1. Many multi-threaded applications can be reduced to have a similar application skeleton. Specifically, the applications in the following sections all follow this model.

We define the throughput of this application as the total number of loop iterations performed per time unit by all available workers. This throughput will be annotated by $F$ (for frequency).

Following is a list of application, environmental and system parameters which effect the application throughput:
4.2. THEORETICAL ANALYSIS

\[ L \quad \text{Number of shared objects and their associated locks} \]
\[ W \quad \text{Number of worker threads} \]
\[ P_{\text{write}} \quad \text{Write probability per loop iteration} \]
\[ T_{\text{chk}} \quad \text{Average checkpoint time} \]
\[ T_{\text{roll}} \quad \text{Average rollback time} \]
\[ T_{\text{net}} \quad \text{Network round-trip delay} \]
\[ M \quad \text{Maximal number of speculations per speculative session (quota)} \]
\[ T_1 \quad \text{compute(T1) execution time with a bare JVM} \]
\[ T_{1,JS} \quad \text{compute(T1) execution time with JS bytecode instrumentation} \]
\[ T_{1,SL} \quad \text{compute(T1) execution time with JS and SL bytecode instrumentation} \]
\[ Q \quad \text{Average lock request queue size} \]

4.2.1 Throughput on a Bare JVM

When the application is executed on a physical shared memory system, the most significant operations in each iteration are the lock acquisition operation and the executions of the compute method. Each worker holds a lock for \( T_1 \) time units. The average lock requester queue size is \( Q \). Therefore, the average lock acquisition time here is \( Q \times T_1 \). Overall, the time required per loop iteration is \( Q \times T_1 + T_1 + T_2 \).

Therefore, the overall throughput for \( W \) worker threads on a physical shared memory system with \( W \) cores/processors is:

\[
F_{\text{bare}} = \frac{W}{Q \times T_1 + T_1 + T_2} \quad (4.2.1)
\]

There are \( W \) worker threads contending on \( L \) locks. Thus, \( W/L \) workers are contending on each lock. Requesters leave the lock requester queue at a rate of \( 1/T_1 \). Additionally, on every loop iteration, each worker spends \( T_2 \) time units outside the synchronized block. Therefore, the average lock request queue size can be defined as follows:

\[
Q = \max(0, \frac{W}{L} - \frac{T_2}{T_1}) \quad (4.2.2)
\]

\[ \text{Technion - Computer Science Department - M.Sc. Thesis MSC-2008-22 - 2008} \]
4.2.2 Throughput on JavaSplit without Speculative Locking

When the application is distributed using JavaSplit, each loop iteration consists of sending a message to the lock owner ($T_{net}$ time units), waiting until the lock passes through all $Q$ requesters in the lock queue, each thread in the queue has to finish working with the lock ($T_{1,JS}$ time units), to checkpoint its state and to pass the lock on ($T_{chk} + T_{net}$ time units). When lock ownership is finally received, the thread can perform its computation ($T_{1,JS} + T_{2,JS}$ time units).

Even if the lock requester queue is empty, the current owner still has to checkpoint its state to pass the lock to the requesting thread. Figure 4.2 shows the flow of a non-speculative lock acquisition, when there is one lock requester in the lock request queue.

Additionally, each worker has to serve the lock ownership requests of other requesters. These lock ownership transfers require the thread which transfers the ownership to halt its execution and checkpoint its state ($T_{chk} + T_{net}$ time units), as described in section 3.2. These requests arrive at an average rate of one request per worker per iteration.

Thus, the overall throughput of non-speculative distributed JavaSplit is:

$$ F_{JS} = \frac{W}{2 * (T_{net} + T_{chk}) + Q * (T_{net} + T_{chk} + T_{1,JS}) + T_{1,JS} + T_{2,JS}} $$

(4.2.3)

4.2.3 Throughput on JavaSplit with Speculative Locking

Ideally, with speculative locking, the basic iteration time is equal to $T_{1,SL} + T_{2,SL}$. However, a loop iteration consists of two additional factors.

First, similarly to JavaSplit without speculative locking support, each worker has to serve the lock ownership requests of other threads. Additionally, in this analysis, we will take a strict assumption that before each speculative mode session, a checkpoint is taken, servicing the incoming lock ownership requests and reducing the lost application time in case of a speculation failure. Thus, if in each speculative session we perform $S$ acquisitions ($S < M$), the worker will perform $S$ iterations in speculative mode and then it will leave speculative mode and perform a checkpoint. Therefore, we get that on average, every loop iteration has an overhead of ($T_{chk} + T_{net}$)/$S$ time units for checkpointing.

The second factor is the overhead of speculation failure, either due to a data conflict or due to
4.2. THEORETICAL ANALYSIS

![Diagram of lock acquisition]

Figure 4.2: Flow of a blocking lock acquisition

a speculative deadlock. In both cases, the failure overhead is equal to the rollback time ($T_{\text{roll}}$ time units) and the lost application time (in the worst case, $S \times (T_{1,SL} + T_{2,SL})$ time units). We annotate the probability for any speculation failure by $P_{\text{fail}}$, which consists of the probability of a conflict ($P_{\text{conflict}}$) and the probability for a deadlock ($P_{\text{deadlock}}$).

Therefore, if we define $T_{\text{compute}} = S \times (T_{1,SL} + T_{2,SL})$, the time it takes to perform $S$ iterations is approximately:

$$\frac{S}{F_{SL}} = T_{\text{compute}} + (T_{\text{chk}} + T_{\text{net}}) + P_{\text{fail}} \times (T_{\text{roll}} + T_{\text{compute}})$$

(4.2.4)

And therefore, the throughput of speculative distributed JavaSplit is:

$$F_{SL} = \frac{W}{T_{1,SL} + T_{2,SL} + (T_{\text{chk}} + T_{\text{net}})/S + P_{\text{fail}} \times (T_{\text{roll}}/S + T_{1,SL} + T_{2,SL})}$$

(4.2.5)

Data Conflict Probability

The probability for a data conflict ($P_{\text{conflict}}$) depends on multiple factors.

One factor is the number of write operations performed by the application ($P_{\text{write}}$) and the timing
of these operations. Applications which perform fewer write operations have a lower probability for a data conflict. However, even applications which perform many write operations, may still avoid data conflicts, if the application also performs synchronization operations which push write notices ahead of the corresponding speculative read operations.

Also, applications which use a single lock to protect several separate data elements have a lower probability for a data conflict, as each worker may access a different data element. Thus, the write notice will be received for one data element and the speculative read operation will be for a second data element, avoiding a conflict.

Another factor is the time spent between sending out the lock acquisition request and performing a speculative read operation. If the lock ownership reply is received before a speculative read operation is performed, data conflicts may be avoided. However, if the last lock ownership reply, which ends speculative mode, is received after multiple speculative read operations had been performed, it may increase the probability for a data conflict: the more speculative read operations performed, the higher the chance that an incoming write notice will conflict with one of the speculative read operations.

**Speculative Deadlock Probability**

The speculative deadlock probability ($P_{\text{deadlock}}$) is a complex function of multiple parameters. The number of locks, $L$, has a major effect on this probability. If $L = 1$, no speculative deadlocks will occur (by definition). If $L >> W$ and $L >> M$, the probability for a speculative deadlock will be low, as this reduces the probability for multiple workers to request the same locks in a conflicting order.

The amount of time each worker spends in speculative mode is also a factor effecting the speculative deadlock probability. If each worker spent all of its time in speculative mode and $L > 1$, speculative deadlocks are bound to happen: two workers will eventually request locks from each other. Among other factors, this amount of time depends on the value of $M$. The more speculative lock acquisitions the worker can perform, the longer it will stay in speculative mode. If $M \rightarrow \infty$ a worker may spend all of its time in speculative mode, setting $P_{\text{deadlock}} = 1$.

Thus, increasing $M$ increases the number of lock acquisitions performed in a batch, which is
beneficial, but also increases the probability for a speculative deadlock, which reduces overall performance. For each application execution, an optimal value for $M$ exists, annotated by $M_{\text{max}}$.

### 4.2.4 Speculation-Optimal Properties

#### Application Properties

The performed theoretical analysis can help estimate how applications will react to speculative locking.

Speculative locking should be beneficial for applications which have a low probability for data conflicts ($\text{low } P_{\text{conflict}}$). We have observed that a low probability for data conflicts exists when application threads access different fields of the same shared object, when the application pushes write notices ahead of speculative read operations, when few write operations take place and when speculative read operations occur after lock ownership is received.

Speculative locking will also better benefit applications with a low probability for speculative deadlocks ($\text{low } P_{\text{deadlock}}$). We have seen that single-lock applications and applications with a significant amount of locks usually have fewer speculative deadlocks. Additionally, applications which do not require more than one concurrent speculative lock acquisition will have lower probability for speculative deadlocks, because the time spent in speculative mode will be shorter in these applications. Other applications which spend less time in speculative mode are applications with a lower load on the lock request queues, smaller checkpoint sizes and applications which acquire locks for shorter periods of time. All of these applications have lower probability for speculative deadlocks.

As an example, single-lock applications with low data conflict probability are ideal for speculative locking, as they have no speculative deadlocks and allow scalability where other systems must serialize access. The speedup is especially significant in applications with $T_1 >> T_2$. These applications will have $Q \to W$ and therefore will not be scalable with non-speculative systems. We will demonstrate this in the next sections.

#### Environmental and System Properties

We have also seen that the throughput of a system with speculative locking support depends on system properties.
For example, it is always desirable that the underlying fault-tolerant system would perform checkpoints and rollbacks as quickly as possible (reduces $T_{chk}$ and $T_{roll}$). Also, it is desirable that read access monitoring will require the minimal possible overhead (reducing $T_{1,SL}, T_{2,SL}$).

Reducing the time to detect a speculative deadlock will also improve performance (reducing $T_{roll}$). This depends on the speculative deadlock detection mechanism, which we will not analyze in this chapter.

To reduce the effect of speculative deadlocks, applications with a large number of locks and short computation time between lock acquisitions should have a limit on the number of speculations. That limit should be set by the speculative runtime and should be equal to $M_{max}$.

The round-trip delay effects the speedup gained by speculative locking significantly. First, it reduces the throughput of non-speculative systems, because as $T_{net}$ increases, the lock fetch time increases. Second, it increases the amount of time spent in speculative mode, increasing the probability for a speculative deadlock and the probability for data conflicts. Hence, using speculative locking in computation environments with high round-trip delays may be very beneficial, but only when the application has properties which reduce the probability for speculation failures.

### 4.3 Application Performance Results

In this section we compare the performance of several applications on JavaSplit with and without speculative locking support. The results are correlated to the theoretical analysis presented in the previous section.

#### 4.3.1 SPECjbb*

We have implemented a minimal version of the SPECjbb2000 benchmark, which allows better flexibility in the evaluation and analysis of speculative locking. This benchmark is an abstraction of a multi-threaded server middleware in which the server employs multiple worker threads to process incoming client requests. The application uses numerous objects and their corresponding locks. Each client request requires exclusive access to one of the shared objects and may modify it. After performing this exclusive data access, the client request is processed. Performance is measured by the total number of client requests served by the distributed server per second. This benchmark
4.3. APPLICATION PERFORMANCE RESULTS

serves as a good test case for evaluating how speculative locking handles applications which use fine-grained locking with thousands of objects and locks.

**Low Acquisition Rate**

We have executed this benchmark on our evaluation cluster with a varying number of workers. The application had 10,000 shared objects and their associated locks, each worker handled 100 client requests, processing each client request for approximately 20ms. Due to the nature of this execution (i.e., low lock acquisition frequency), no data conflicts or speculative deadlocks had occurred.

Figure 4.3 shows the overall throughput on JavaSplit with and without speculative locking support. The figure also displays a breakdown of the time each worker thread spends in blocked state. SL and JS denote the results obtained on JavaSplit with and without speculative locking, respectively.

Overall, both systems achieve scalable throughput in this case. The speculative locking system achieves better throughput due to elimination of the lock acquisition overhead. Without speculative locking support, workers are blocked when lock fetches are performed. Both systems block workers to perform checkpoints (serving external lock requests).

As the number of workers increase, both SL and JS spend more time on performing checkpoints, as the workers need to serve more incoming lock requests.

Note that upon lock acquisition, before entering speculative mode, the implementation of speculative locking in JavaSplit waits until all pending external requests are served, to reduce the probability for speculative deadlocks. This behavior explains the slight increase in speculative lock acquisition time as the number of workers increase. Thus, some checkpoining occurs in the context of "Spec. Lock Fetch (SL)" and is not shown in the "Checkpointing (SL)" bar.

Correlating the results to the theoretical analysis, this execution has \( Q \rightarrow 0 \) and therefore speculative locking only reduces the lock acquisition overhead \( (T_{net} + T_{op}) \) which is associated with every lock fetch. Additionally, this execution has \( T_2 >> T_1 \), reducing the probability for speculative deadlocks.

**High Acquisition Rate**

To evaluate how speculative locking handles a higher lock acquisition rate, we eliminated the processing time required per client request, resulting in a high rate of lock acquisition requests. To
enable correct measurement, we increased the overall number of client requests handled by each worker to be 5,000.

When executed with speculative locking, this requires setting a speculation quota or speculative deadlocks are bound to happen. By experimentation, setting the quota to be 30 speculations per session prevents speculative deadlocks and provides the optimal throughput. Section 4.4.4 performs additional analysis regarding the optimal quota value.

Thus, each worker sends out speculative lock acquisitions until reaching the quota. Then, it performs non-speculative lock acquisitions until all speculations performed in this speculative session are completed.

Figure 4.4 shows the achieved throughput and the block time breakdown of a single worker. The performance of both systems scale. However, the worker block time increases as a function of worker number, due to the increasing overload on the home nodes. The same number of home nodes need to serve more requests per second as the number of workers increases, increasing the lock request service time. JavaSplit with speculative locking support shows better throughput, yet each worker performs a significant amount of non-speculative lock acquisitions, due to the low speculation quota.
4.3. APPLICATION PERFORMANCE RESULTS

![Graph showing throughput and worker breakdown for SPECjbb](image)

Figure 4.4: Throughput and worker breakdown for SPECjbb* (high acq. rate)
4.3.2 TSP

We have executed a multi-threaded benchmark solving the Traveling Salesman Problem using multiple worker threads. Each worker scans a range of permutations. Whenever a new best path is found, a global shared object containing the "best path" is updated. Each worker constantly monitors the "best path" shared object, to avoid scanning paths which have higher cost than the cost of the best known path. This variable is shared and might have simultaneous read/write access and therefore has a common lock protecting data access to it. Throughput of this benchmark is defined as $1/T_{solution}$, where $T_{solution}$ is the period of time required by the entire cluster to solve a given TSP problem.

Figure 4.5 shows the TSP throughput of JavaSplit with and without speculative locking support, with a varying number of workers, 11 cities, each with a maximal coordinate of 1000. The breakdown of a worker’s block time is also displayed, omitting operations which required negligible time (such as, checkpointing time, in this case). The "Spec. Failures (SL)" bar shows the amount of time the worker lost due to data conflicts, including rollback time and lost application time.

With one worker, solving TSP is faster in JavaSplit without speculative locking support, because in this case only one lock acquisition is performed and the rest of the execution is pure computation. As speculative locking has some overhead for read access monitoring, we get a slowdown in this case.

With multiple workers, the throughput of JavaSplit without speculative locking decreases as the number of workers increases, because of increasing lock fetch time. The throughput of JavaSplit with speculative locking is proportional to the number of workers. Note that there is a slight overhead caused by recovering from data conflicts, which are inherent in this application (whenever a best tour is found, a data conflict is bound to happen). The time spent on speculation failures generally increases as the number of workers increase. However, the number of rollbacks depends on multiple parameters, including how work is distributed among the worker threads, which can explain the slight drop in speculation failure time for 26 workers.

In this execution, which uses a single lock ($L = 1$), as the number of workers increase, we get $Q \rightarrow W$. Thus, the lock request queue is usually full, causing lock fetches in non-speculative systems to have a significant impact on performance. Lock fetch time is a function of the worker number, causing scalability issues. No speculative deadlocks occur, because only a single lock exists. The
4.3. APPLICATION PERFORMANCE RESULTS

Figure 4.5: TSP throughput and worker block time breakdown

The number of data conflicts is limited by the number of best tours found for a specific problem, reducing the probability for a speculation failure ($P_{fail}$) and resulting in higher speculative locking speedups.

4.3.3 HasTable

Following is a custom benchmark written to demonstrate the benefits speculative locking can provide to applications using coarse-grained locking.

Many multi-threaded applications use a single lock to protect a shared resource (coarse-grain locking). This approach simplifies application development by reducing application complexity and thereby reducing the probability for bugs resulting from invalid synchronization.

However, when the application keeps the lock acquired for a long period of time, this approach may have a significant impact on application scalability. Application threads are frequently blocked when trying to acquire the single lock, even if no real data contention exists.

Workers in the HasTable benchmark repeatedly acquire the same lock and hold it in acquired state for a non-negligible period of time. While holding the lock they may read and write to the
hashtable associated with the lock and perform some other processing or I/O operations. The maximal possible throughput for this application is $W/T_1$ (using the annotations defined in section 4.2).

Figure 4.6 shows the throughput of JavaSplit without speculative locking support, JavaSplit with speculative locking support and $P_{\text{write}} = 0$ and JavaSplit with speculative locking support and $P_{\text{write}} = 0.05$. In the case of JavaSplit without speculative locking support, the worker threads acquire the single lock in a serialized manner, and therefore the entire cluster can’t achieve a throughput of more than $1/T_1$. The speculative locking version achieves near-linear scalability, as there are no speculative deadlocks and very few data conflicts.

In general, applications with coarse-grain locking can significantly benefit from speculative locking and this is an example of such an application. Writes are performed on different hashtable buckets, resulting in few data conflicts ($P_{\text{conflict}} \to 0$). No speculative deadlocks occur ($L = 1$). This setting allows speculative locking to enable the highest degree of parallel execution. However, for non-speculative systems, throughput is not scalable and is limited by $1/T_1$.

4.4 Micro-Benchmark Results

4.4.1 Network Latency

To measure the latency of a lock acquisition operation, we execute a micro-benchmark in which a single worker thread acquires multiple locks from a single home node. We measure the average time to acquire a lock with varying network latencies: 0.1ms, which is the approximated latency of our Ethernet cluster; 10ms and 100ms which were software-simulated and therefore only serve as an approximation of network latency. In our measurements of the lock acquisition latency in JavaSplit with speculative locking support, we measure the critical path overhead and not the overall time the thread spends in speculative mode.

Figure 4.7 shows the average lock acquisition delay for JavaSplit without speculative locking support, JavaSplit with speculative locking support and a bare sequential JVM (Sun Java 1.6 JVM). As expected, the bare JVM and speculative locking are unaffected by the change in network latency. JavaSplit suffers a significant increase in lock acquisition latency as the network latency increases. The increase in the lock acquisition latency is due to the time required to transfer the lock request.
4.4. MICRO-BENCHMARK RESULTS

Figure 4.6: Hashtable throughput

Figure 4.7: Average lock acquisition latency with varying network latencies
message and the time required to distribute the checkpoint in the process which transfers lock ownership.

4.4.2 Data Conflicts

To evaluate the effect of data conflicts on the performance of speculative locking, we executed SPECjbb* with varying probabilities for data conflict. This experiment was executed with 16 workers, negligible client request processing time and 100,000 locks.

Figure 4.8 shows the throughput of SPECjbb* with varying data conflict probabilities. The figure also displays the amount of time required to recover from data conflicts. As the percentage of conflicts increases, more time is spent on recovering from speculation failures, reducing the overall throughput proportionally.

The time spent by each worker to handle speculation failures is a multiplication of the number of data conflicts by the time lost per data conflict. The throughput generally decreases as the data conflict probability increases.
4.4. MICRO-BENCHMARK RESULTS

Figure 4.9: Varying preemptive checkpointing frequency

To avoid losing a significant amount of computation time upon speculation failure, the runtime should perform preemptive checkpoints. Performing preemptive checkpoints at a high frequency may result in a significant checkpointing overhead which may be unnecessary. However, reducing this frequency may increase the time spent to recover from a speculation failure, which includes the rollback time and the lost application time.

Figure 4.9 shows the time spent on checkpointing and on recovering from failures for a micro-benchmark with some non-negligible data conflict probability \( P_{conflict} = 0.5 \). The overall overhead of checkpointing and recovery is also displayed. It can be noticed that, for this execution, an optimal checkpointing frequency exists (the frequency which provides the minimal total overhead). This optimal checkpointing frequency can be dynamically detected: the speculative locking mechanism can track the checkpointing overhead and the rollback overhead at runtime and modify the checkpointing frequency accordingly, trying to reduce the overall overhead.
4.4.3 Read Access Monitoring

The read access monitoring required to support speculative locking was implemented in JavaSplit via Java bytecode instrumentation, in addition to the existing bytecode instrumentation. We estimate the overhead of this additional instrumentation using sequential applications from the Java Grande Forum (JGF) Benchmark Suite (version 2.0) [3]. We compare the throughput of the programs instrumented for execution on JavaSplit with and without speculative locking support.

The measurements were performed on a commodity Windows XP SP2 desktop machine with Intel Pentium dual-core 1.6 GHz processor and 2 GB of RAM, using Sun Java 1.6 JVM.

The sequential benchmarks in the JGF benchmark suite are divided into three categories. The first one measures the performance of low level operations such as arithmetic, casts, assignments, allocation of data, exceptions, loops, and method invocations. The applications in the second category are short codes that carry out specific operations frequently used in Grande applications. The third category consists of large scale applications that demonstrate Java’s potential in tackling real problems.

Table 4.1 summarizes the results. We omitted multiple stack-based benchmarks in which the comparison shows that the performance of the SL-rewritten bytecodes is close to the non-SL version. The “Difference” column shows the difference between non-SL-instrumented benchmarks and SL-instrumented benchmarks. Let \( A \) and \( B \) be the throughputs of the non-SL-instrumented benchmark and its SL-instrumented version respectively. The corresponding value in the “Difference” column is \( 100 \times (A - B) / A \). Positive values indicate higher throughput of the non-SL-instrumented application.

We observe throughput differences of up to a 43% slowdown in the sequential execution of SL-instrumented benchmarks compared with non-SL-instrumented benchmarks. With the current implementation of access monitoring in JavaSplit for the purpose of speculative locking, each access to an object field or an array element is instrumented in order to monitor local accesses in speculative mode. The execution of this additional bytecode per access instruction reduces the throughput of these benchmarks.

Additionally, the current implementation of speculative locking object access instrumentation in JavaSplit has a complex mechanism to access static objects, resulting in a significant slowdown when such accesses are performed. Usage of static objects is inherent in many of the JGF benchmarks for

48 CHAPTER 4. PERFORMANCE EVALUATION
4.4. MICRO-BENCHMARK RESULTS

Table 4.1: JGF Benchmark Suite – microbenchmark results. Due to space limitations we present only the subset of benchmarks which have a throughput difference.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JS</th>
<th>SL</th>
<th>Difference (%)</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Assign:Same:Scalar:Instance</td>
<td>130254360.00</td>
<td>8889680.00</td>
<td>35.60</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Same:Scalar:Class</td>
<td>133047680.00</td>
<td>7816248.00</td>
<td>40.69</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Same:Array:Local</td>
<td>88210512.00</td>
<td>6078680.00</td>
<td>20.88</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Same:Array:Instance</td>
<td>260327.50</td>
<td>1979246.10</td>
<td>24.74</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Same:Array:Class</td>
<td>4927519.00</td>
<td>3813912.00</td>
<td>22.56</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Other:Scalar:Instance</td>
<td>101551066.00</td>
<td>73672504.00</td>
<td>27.47</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign:Other:Scalar:Class</td>
<td>133054512.00</td>
<td>78916248.00</td>
<td>40.69</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>2: LUFact:Kernel:SizeB</td>
<td>52.50</td>
<td>40.355</td>
<td>23.11</td>
<td>(Mflops/s)</td>
</tr>
<tr>
<td>2: HeapSort:Kernel:SizeA</td>
<td>735294.1</td>
<td>418410.03</td>
<td>43.10</td>
<td>(items/s)</td>
</tr>
<tr>
<td>2: Crypt:Kernel:SizeA</td>
<td>3556.6084</td>
<td>3121.7483</td>
<td>12.23</td>
<td>(Kbyte/s)</td>
</tr>
<tr>
<td>2: FFT:Kernel:SizeA</td>
<td>155160.7</td>
<td>134089</td>
<td>13.58</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>2: SparseMatmul:Kernel:SizeA</td>
<td>36.055527</td>
<td>32.242462</td>
<td>10.58</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: LUFact:Kernel:SizeB</td>
<td>53.2972</td>
<td>40.64349</td>
<td>23.74</td>
<td>(Mflops/s)</td>
</tr>
<tr>
<td>2: HeapSort:Kernel:SizeB</td>
<td>557537.94</td>
<td>340785.16</td>
<td>38.88</td>
<td>(Items/s)</td>
</tr>
<tr>
<td>2: Crypt:Kernel:SizeB</td>
<td>3667.7057</td>
<td>3281.9167</td>
<td>10.52</td>
<td>(Kbyte/s)</td>
</tr>
<tr>
<td>2: FFT:Kernel:SizeB</td>
<td>131654.16</td>
<td>111430.62</td>
<td>15.96</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>2: SOR:Kernel:SizeB</td>
<td>4.356919</td>
<td>3.109936</td>
<td>28.62</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: SparseMatmul:Kernel:SizeB</td>
<td>11.884935</td>
<td>10.466272</td>
<td>11.94</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: Crypt:Kernel:SizeC</td>
<td>3065.5447</td>
<td>3324.6892</td>
<td>9.30</td>
<td>(Kbyte/s)</td>
</tr>
<tr>
<td>2: LUFact:Kernel:SizeC</td>
<td>57.970824</td>
<td>43.968105</td>
<td>24.17</td>
<td>(Mflops/s)</td>
</tr>
<tr>
<td>2: HeapSort:Kernel:SizeC</td>
<td>426562.94</td>
<td>275154.62</td>
<td>35.49</td>
<td>(Items/s)</td>
</tr>
<tr>
<td>2: FFT:Kernel:SizeC</td>
<td>121740.76</td>
<td>108504.016</td>
<td>10.87</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>2: SOR:Kernel:SizeC</td>
<td>2.4316702</td>
<td>1.7372572</td>
<td>28.56</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: SparseMatmul:Kernel:SizeC</td>
<td>1.495171</td>
<td>1.3734904</td>
<td>8.16</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>3: Euler:Init:SizeA</td>
<td>25464.94</td>
<td>17257.23</td>
<td>32.23</td>
<td>(Gridpoints/s)</td>
</tr>
<tr>
<td>3: Euler:Run:SizeA</td>
<td>0.36689305</td>
<td>0.22182902</td>
<td>39.27</td>
<td>(Timesteps/s)</td>
</tr>
<tr>
<td>3: Euler:Total:SizeA</td>
<td>0.003650928</td>
<td>0.002219091</td>
<td>39.22</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MolDyn:Run:SizeA</td>
<td>19950.025</td>
<td>15146.326</td>
<td>24.08</td>
<td>(Interactions/s)</td>
</tr>
<tr>
<td>3: MolDyn:Total:SizeA</td>
<td>0.015810527</td>
<td>0.011980639</td>
<td>24.22</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MonteCarlo:Run:SizeA</td>
<td>800.1549</td>
<td>624.3756</td>
<td>29.86</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>3: MonteCarlo:Total:SizeA</td>
<td>0.0783921</td>
<td>0.054842602</td>
<td>29.99</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: RayTracer:Run:SizeA</td>
<td>706.9247</td>
<td>450.9922</td>
<td>36.20</td>
<td>(Pixels/s)</td>
</tr>
<tr>
<td>3: RayTracer:Total:SizeA</td>
<td>0.028572245</td>
<td>0.019465098</td>
<td>31.87</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: AlphaBetaSearch:Run:SizeA</td>
<td>103456.13</td>
<td>90858.1</td>
<td>12.18</td>
<td>(Positions/s)</td>
</tr>
<tr>
<td>3: Euler:Init:SizeB</td>
<td>38539.73</td>
<td>2651.166</td>
<td>31.08</td>
<td>(Gridpoints/s)</td>
</tr>
<tr>
<td>3: Euler:Run:SizeB</td>
<td>0.16102631</td>
<td>0.09180715</td>
<td>42.61</td>
<td>(Timesteps/s)</td>
</tr>
<tr>
<td>3: Euler:Total:SizeB</td>
<td>0.001595515</td>
<td>0.1172407</td>
<td>42.52</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MolDyn:Run:SizeB</td>
<td>23101.254</td>
<td>18562.988</td>
<td>19.65</td>
<td>(Interactions/s)</td>
</tr>
<tr>
<td>3: MolDyn:Total:SizeB</td>
<td>8.56E-04</td>
<td>6.88E-04</td>
<td>19.59</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MonteCarlo:Run:SizeB</td>
<td>831.0019</td>
<td>560.6743</td>
<td>32.53</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>3: MonteCarlo:Total:SizeB</td>
<td>0.01329469</td>
<td>0.008936151</td>
<td>32.78</td>
<td>(Solutions/s)</td>
</tr>
</tbody>
</table>
which we observe slowdowns.

Adding bytecode instrumentation to support speculative locking also increases the number of bytecode instructions per method. Large methods can make it hard for the just-in-time compiler to perform optimizations, resulting in slowdowns.

In section 6.3 we suggest several methods to reduce this overhead.

4.4.4 Speculative Deadlocks

When there is no limitation on the number of speculations performed per speculative session, a thread may stay in speculative mode forever. For example, a new speculative lock acquisition may start before previous speculative lock acquisitions end, never allowing the thread to leave speculative mode. Therefore, as described earlier, we must set a quota on the number of speculative lock acquisitions per speculative session.

According to the theoretical analysis, the probability for speculative deadlocks in applications with a high lock acquisition frequency should increase as the speculation quota increases. Figure 4.10 shows the probability for speculative deadlocks as a function of the speculation quota, when running SPECjbb\(^8\) with 100,000 locks, 1,000 iterations per worker and negligible processing time per loop iteration. These results confirm the theoretical analysis. As the quota increases, threads may spend more time in speculative mode, increasing the probability for speculative deadlocks.
4.5. PERFORMANCE EVALUATION SUMMARY

Figure 4.11: SPECjbb* throughput as a function of speculation quota

Section 4.2.3 describes an optimal speculation quota value, $M_{max}$, which provides the best throughput for a specific execution, balancing between batching of speculative lock acquisition requests and the amount of speculative deadlocks. Figure 4.11 shows the throughput of the same SPECjbb* execution as a function of speculation quota. $M_{max}$ is the speculation quota which provides the maximum throughput in this graph.

4.5 Performance Evaluation Summary

Performance evaluation confirms that speculative locking may indeed boost the performance of applications executed with a fault-tolerant distributed shared memory system.

In some cases, where little lock contention exists, the speedup gained by speculative locking may be limited to the removal of the lock acquisition overhead from the critical path of the application. This overhead elimination may be very significant, for example, in environments with high network latencies or large checkpoint sizes.

In other cases, for applications which suffer from lock contention, speculative locking can enable higher levels of concurrency, enhancing application scalability up to the maximal possible level.

We have seen that the performance of speculative locking can be sensitive to data contention. Applications with multiple data conflicts may not be suitable for execution with speculative locking.
CHAPTER 4. PERFORMANCE EVALUATION

The proposed scheme employs several heuristics to detect these situations and avoid speculating on conflicting locks. In section 6.2 we suggest an additional method to reduce the data conflict probability.

The speculative deadlocks phenomena may limit the speedup gained by speculative locking. At the worst case, to avoid speculative deadlocks, the mechanism can automatically reduce the percentage of speculative lock acquisitions performed. Methods to detect and resolve speculative deadlocks were presented. We also characterized application and environmental properties which effect the magnitude of speculative deadlocks. In section 6.1 we discuss a method to significantly reduce the probability of speculative deadlocks.

Read access monitoring may also reduce the performance of speculative locking. In JavaSplit, read access monitoring is implemented by additional bytecode instrumentation, which slows down object-based computations. This overhead is highly-dependent on the implementation of access monitoring in the underlying system. Section 6.3 suggests methods of reducing this overhead.
Chapter 5

Related Work

Speculative lock acquisition descends from *Optimistic Concurrency Control* [19, 31] which has been applied to increase the throughput of database systems. This approach allows optimistic execution of a transaction, followed by a validation phase which determines transaction success based upon a system-specific collision detection mechanism. Upon collision, the transaction is aborted and retried. Following attempts to execute a transaction may be more efficient as some of the required data becomes locally available due to the first attempt.

There have been a number of works that suggest employment of speculating locking to improve performance of parallel applications on shared-memory multiprocessors [22, 24, 26]. These works describe micro-architectural hardware extensions whose purpose is to detect collisions, to ensure speculative writes are not propagated to other threads and to enable rollbacks. Unlike the scheme described in this work, these methods require that the thread state is checkpointed before each speculative acquire. These methods indeed allow speculating threads to execute past the release operation. However, they prohibit speculating on several locks simultaneously. Hence, a speculating thread that needs to acquire another lock performs a non-speculative acquisition. Another downside of these methods is that the amount of data accessed while in speculative mode is limited by the cache size or by the size of the hardware extensions that enable speculation. In the case these limitations are exceeded the thread either blocks or rolls back.

In *Speculative Lock Elision* [24], the hardware dynamically identifies synchronization operations, predicts them as being unnecessary, and elides them. Conflicting memory accesses of other processes
are detected through cache invalidation requests. Local data accesses are monitored by snooping the load/store queue, if possible, or by augmenting each cache block with an access bit. To enable rollback in the case of a mis-speculation, the memory updates are buffered in a special buffer, while the register state is either checkpointed or preserved using a reorder buffer, if available.

The *Speculative Synchronization* technique proposed in [22] allows threads not only speculatively acquire locks but also execute past active barriers and unset flags. The blocking synchronization operations are deferred to a *speculative synchronization unit*, while the threads optimistically continue execution. Thus, similar to our scheme, at any given moment one thread is considered to be the rightful owner of the lock. This thread never rolls back due to a data conflict, which guarantees *forward progress*. Access monitoring is performed similarly to SLE, as follows. External modifications are detected through invalidations issued by the cache coherence protocol, while local accesses are monitored by access bits attached to each cache line. Thread checkpointing and rollback are implemented using pre-existing *thread-level speculation* hardware.

*Speculative Lock Reordering* [26] potentially improves concurrency of the two speculation methods described above. It exploits the fact that individual operations by different threads can appear in any order as long as all threads observe sequential ordering of critical sections. Consequently, a read-write collision resulting in a mis-speculation in the previous schemes can be reordered so that the critical section performing the read access appears as being executed before the critical section performing the write access. To avoid unnecessary mis-speculations, the modifications of all threads are buffered and collision detection is delayed until all threads leave the critical section. During validation, a thread dependency graph is constructed and commit order is determined. Threads with unresolvable dependencies are re-executed. On the downside, delayed validation precludes true collisions from being detected eagerly, i.e. as soon as they occur.

*Transactional Memory* [10] simplifies parallel programming for shared-memory multiprocessor by allowing a group of load and store instructions to execute in an atomic way without employment of locks. The modifications performed by a thread during the execution of an atomic block (or transaction) are delayed via a specialized cache hierarchy. Atomic blocks execute speculatively while the cache coherence protocol detects conflicts. If a conflict is detected the thread rolls back to the beginning of the block. Otherwise, its writes are committed and thus made visible to the other threads.
The general idea of transactional memory model has been extended to software [28, 9, 21]. Software transactional memory, first proposed in [28], provides a similar programming paradigm yet does not require any hardware support. Its hardware independence allows a more flexible implementation. Nevertheless, the performance of software transactional memory remains inferior in comparison to its hardware counterpart.

There is a certain similarity between speculative lock acquisition and transactional memory. Both approaches avoid blocking on critical sections by optimistically speculating that no real data conflict occurs during execution of an atomic block. Both approaches require collision detection and perform a rollback in the case speculation fails. While the abort mechanism in software transactional memory is based on a transaction log of some sort, the proposed scheme employs existing rollback capabilities of a fault-tolerant system.

A recent work [17] proposes a distributed implementation of software transactional memory. This work exposes a significant performance impact of communication delays during validation and commit phases. In contrast, the speculative locking scheme proposed in this work masks communication delays by allowing processes to continue execution in parallel to interaction with remote nodes.

Previous work [1, 2] shows how to remove unnecessary synchronization using static analysis. However, these static analysis techniques can only remove synchronization operations where there is no possible execution path where the synchronization is needed. Therefore, these techniques can not improve the performance of common scenarios in which a lock contention is rare, yet possible.

Other work [12] suggests Lock Prefetching - a technique which transparently sends out lock acquisition messages ahead of time, so that the lock will arrive when it is needed, thereby hiding the lock acquisition overhead. However, this method requires that the programmer (or compiler) will statically insert annotations, telling the runtime when to prefetch the lock. This approach can also be very sensitive to variations in network latency and to changes in the executed application.

A better approach, called Lock Acquirer Predictor [27] provides a runtime predictor which tries to predict the next acquirer for a lock and then pre-fetches the lock, along with its memory consistency data. However, LAP cannot fully exploit implicit lock parallelism, as it cannot allow multiple threads to concurrently acquire the same lock, even if there is no data conflict among the threads.
Chapter 6

Future Work

In this chapter we present several future extensions to the speculative locking scheme. These additional optimizations may increase the performance of the speculative locking mechanism by reducing the overhead associated with the protocol.

6.1 Passing Lock Ownership in Speculative Mode

One of the limitations of the current speculative locking mechanism is its inability to pass locks in speculative mode. The speculative deadlocks phenomena is the result of this limitation.

However, in some cases, it seems to be possible to pass lock ownership while the process is in speculative mode. We can demonstrate this point on the implementation of speculative locking in JavaSplit.

Assume process $B$ requests a lock $L_2$, which is owned by process $A$, but in released state. $A$ can transfer ownership of $L_2$ to process $B$ during speculative mode only if $A$ performed a checkpoint after its last release of $L_2$. In this case, the set of write notices passed along with the lock ownership transfer is the set of write notices which were accumulated up to the point of the latest checkpoint, taken before entering this speculative mode session and after releasing $L_2$. Thus, all the modifications performed during speculative mode are not visible to process $B$. This method does not violate the happens-before order, because this order only requires that process $B$ will observe modifications performed up to the release point of lock $L_2$. 
The suggested mechanism reduces the probability for speculative deadlocks, because in many cases locks can be transferred during speculative mode, reducing the number of inter-process speculative dependencies. In the example shown in figure 6.1, a speculative deadlock is avoided because process A is able to transfer ownership of lock $L_2$ to process $B$ during speculative mode.

### 6.2 Predictive Data Push

Section 4.2 suggests that applications which inherently push write notices ahead of time may have fewer data conflicts.

This raises the possibility of pushing data ahead of time by the runtime without depending on the application, significantly reducing the probability for data conflicts. The home node for each shared object should track which processes frequently read this object. Then, whenever a home receives a diff for a shared object, it pushes the data to its frequent readers, ahead of time. Then, when the readers acquire a lock speculatively, they will already have a more up-to-date version of the speculatively-read objects, reducing the probability that a write notice associated with the incoming lock will invalidate the objects’ version.

Figure 6.2 provides an example of how this mechanism could work. After process $A$ writes to object $X$, it sends the diff to the home node of $X$, which pushes the updated data to process $C$, which is a frequent reader of object $X$. Process $B$ does not write to object $X$ and when it eventually passes the lock to process $C$ along with a write notice for $X$, the speculative read of $X$ performed by process $C$ does not cause a conflict, because it read updated data (due to the data pushed by the home of $X$).

### 6.3 Instrumentation Optimizations

A major portion of the overhead associated with speculative locking in JavaSplit is due to instrumentation and can be significantly reduced by static analysis. At the moment, each access to an object field or an array element is instrumented in order to monitor local accesses in speculative mode.
Figure 6.1: Passing lock ownership in speculative mode
Figure 6.2: Predictive data push
6.3. INSTRUMENTATION OPTIMIZATIONS

Many of these access checks can be eliminated by a number of well known static analysis techniques. For example, it is possible to eliminate access checks by pulling access checks outside the body of loops or to eliminate consecutive access checks of the same object.

Another optimization example is creating (at instrumentation-time) two versions of the application’s basic blocks. One version will have SL instrumentation and the other will not have any SL instrumentation. Whenever the thread enters speculative mode, it executes basic blocks with SL instrumentation. Shortly after the thread leaves speculative mode, it executes basic blocks which are not SL-instrumented. If such a mechanism is implemented, it can completely eliminate the SL instrumentation overhead when the application is in non-speculative mode. Note that many applications spend most of their time in non-speculative mode.

We have not implemented these optimizations as this has not been the primary goal of our research. Once the unnecessary access checks are eliminated we expect that read access monitoring overhead in JavaSplit will become negligible.
Chapter 7

Conclusion

In this work, we presented the idea of improving the performance of a distributed runtime environment by implementing speculative locking on top of its existing fault-tolerance capabilities.

Although the general notion of optimistic lock acquisition has been applied in several domains, such as shared-memory multiprocessors and database systems, we are the first to suggest its employment in a general-purpose fault-tolerant distributed runtime environment.

The work focused on presenting this technique on fault-tolerant distributed shared memory systems, which proved to be excellent candidates for integration of speculative locking techniques, taking advantage of existing mechanisms, such as access monitoring and rollback capabilities.

To demonstrate speculative locking, we have implemented it into JavaSplit, a full-fledged fault-tolerant transparent distributed runtime for execution of multithreaded Java applications. We explored multiple issues which arise from this integration, such as the phenomena of speculative deadlocks and correct employment of fault-tolerance capabilities.

The performance evaluation of speculative locking shows that it is beneficial for applications with little data conflict probability and little speculative deadlock probability. Performance is also significantly affected by the efficiency of read monitoring, checkpointing and rollback capabilities of the underlying distributed system. Yet we have shown that our approach is capable of removing lock acquisitions from the critical path of applications and, in some cases, significantly increase execution concurrency, while not affecting the scalability of the underlying system.
References


REFERENCES


REFERENCES


تكون

שמשמיש僬

בסקפקלציה

杳

살יב נעלת סקפקלציהו במערכת ז friday מחוזות מפואר המنصرו המחבר שכרת היחה Maverick שך

משישב מגמיס זיוס חוליווימ יאessoa. מיך בכרוש שכרת איר商业地产ים. זיויזת לכל היכן היא

תופעה בים "זנק סקפקלציה". מבריקה הפושט מזור. זנק הוא עשיר לחרוזה שטוח שמיים➡️נ-

ב מבריקס מפלס-מזרחי. אציור לחן היא מניפה שמיים במזרחי אל תעל את לחן ב- לחן ב

מbucks ממלף את ממלא במעאות אל תעל את לחן. אלי החיתים יביסים מעל סקפקלציה-המחנה לוקה

מק מוקלים-מזרחי את מחטרה המ_sizes התרומחת. עם זה, מח_goals mostrar מחתקים במובב

סקפקלציה. המוכרים אɟ מקסימיםโล אולם שימושי על חותם מזרחי הליך או אולימפת לחזיר מפרץ

כולם, נרצ החנק הוא לא מקסימים התרומחת ייצאה מחקת סקפקלציה אולימפת חיתים. יש להgiatan שחקן

וז Então איש ממקסימים הממורה גבר האימפרציהولا מחבק מח虮ים-המחנה חיתים-המחנה

אולימפת מחתקים את המ числе למצל את לחון הערה. האם זה או מכינים השמת מחתקים בין

ברעדת נספח שורותridor במקומיות המחק.

ליסיס. בחרו אתpanion מחטף לארץ של לחון המחתקים של מצל סקפקלציה במעל שמערכת

מחזהה עדויות בכ๋ ניצול. או מחטף מח kýサービス לברוע עלייל סקפקלציה במעל שמערכת ברעדת

סיכ כידי מחזון הוא מקסימי בלוניק מבן- ดังนั้นימס מחתח מחתקים זימיים. עומר לחון

מקסימליות. מבצל החנק או מחתח מחתקים במעל שמעני הוא אɟ מקסימי לApiClientות של זיוס-

ביבי המרומית המחתקים המחזרנאות האימפרציה. מחוזות שיאו מקסימי לחון תלעות את זיוס מחצל

במקבל. או מחטף מח ký人民服务 של מחתקים של לחון מחתקים במעלם עומר מחתקים י_patch מחתקים

מחדש ממסס-讎變 מחתקים מחתקים מחתקים. המחזון והיתרואת מחתקים המחזון

בובסêt שלמחזון מחתקים המחתקים מחתקים מзвучות המחזון. המחזון אɟ מחתקים מחתקים מחתקים

מחתקים פוא מחתק, או מחטף מח ký서비스 של מחתקים מחתקים מחתקים מחתקים המחזון בזוזיון-סטלים. אולימפת משימה עניינה

שמשמיש快餐 עמי מחזון מחתקים מחתק מחתקים מחתקים מחתקים.
התרומא המרכזית של הפרויקט היא שיטת הספקולציה במושגי תורת הריצוף, שהיא תורת תקשורת בבית הספר למתמטיקה ול계랏ון באוניברסיטת טכניון. השיטה המרכזית הוא עקרון התיקון, שמתאר את התופעה של התיקון של משליות זיכרון על ידי התיקון של משליות זיכרון. }

**זיהוי**

1. **שם**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?

2. **התרומא**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת على זיהוי של משליות זיכרון?

3. **האוניברסיטת**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?

4. **הנ限り**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?

5. **העונת**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?

6. **ה荄מש**
   - א. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ב. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
   - ג. מהו אפליקציה אוניברסיטאית המבוססת על זיהוי של משליות זיכרון?
תקציר

רשימה מจบ המודעה. המורכבות מספקת על מخيص מקף סטמריטיס. מוחות אלטרנטיביים או תוכניות
לפרישת יישור נובדו. אלטרנטיביות מתקיימות בברק קלינית וטריטורית. המודעות על רמותי
של שונים שושנים מי. גורמים שונים מתאימים לחץ-חברה קים על ניו רמות של אפקיטיזה על
ዘ.circular מאוחדים מתאימים-ת;heightה. המספונות אפקיטיזה של רמות האפקיטיזה על ניו משוב של
מעündig בשילジュ משושן פי. הэффект האפקיטיזה על ניו משובים בה שושנה אפקיטיזה של
דרישות המזומנות שינו פדיגמא או קומפליקציה פורמלית של אפקיטיזה.

אפקיטיזה מרדת-תחום עלי במרץ במעמדה על מוסל הלטני את עקביות המידה המושחת בכל
גנישותaturas המבודדותๆ בגר改變ו. ג'ילנט את, המושבות על מסלול הקורטיאט של רמות האפקיטיזה, וצ
מקבלי יקדים. ממקו שית משותפים בין התחיוטים המשותפים בחרוניות המומר. ג'ה משותפים בין
ע.ReadIntים בחומת מספר שותישים מרחקLiver sidelined 미.-pic ידועו אפקיטיזה של רמות האפקיטיזה של
אות אחר משביםGY המבוק שמלים מוזר ג'ש שושנה ז'ות הקורטיאט את המושפת המטרית.

אות אחר שמקבליournals שלק.

على أطرف معبرو أحبار均匀 Wakacje على معبرو محترم-مدعوات معلم بكسر مشوش פי. برعم
מיكوك-אָראָיקטוגרפה:纹理们 منتظم 크게 קיים במקבלי. המלך לקח בליזאט בליזאט על המאמר בדריה
ולא המראות מעשה כים למדים, הקצין רעיון. המדידה וחרותה ממעת התמונת בפשיטה
למירים, איה מושחת את מעבר纹理ים לומג הקדום עליל. מכונשות ממגין ספקולציה הקיימיות ממקדיש
לטועות מוסמך.
כרזונים קל組織 משולשים לקוראות. ישראל וחברה זמורה, על הימורים כאונס לא建築 תכופת.

מתוך.
הכרת taraf
ברצוני להודות לפטר, אשר שליטت על הפתחה מקמלה והקיפה
המחברים. חיבורים והפסטיבליים ניתן להשלים עם כל לאורכת הדר.
ברצוני גם להודות למני. קוקסוונטיים ישים, שטרם ש籌ית רבוד מקספור על מכת ליצר
השדית מרשימה, עליה בבעה העברת מחקר זה.
נגילה ספקרטיסית עבים מערכים מבדיקים תמים

בפני נפילת

יחבר על מתקר

לכל מולי התכנית שול הדרישות לקבלת תואר
מִמסֶּרֶת למדעים
מדעי המחשב

טל דמי

ornecedor הtrusted אוים - מון טכנולוגיה לישראל
ענף

אב וכש.ח

אוגוסט 2008
גילה סלקונטייבית Zubir מעריכת מבתרות
עמיתות בנד בנילוח
טל זָנִיר