Execution of Monolithic Java Programs on Large Non-Dedicated Collections of Commodity Workstations

Konstantin Shagin
Execution of Monolithic Java Programs on Large Non-Dedicated Collections of Commodity Workstations

Research Thesis

Submitted in Partial Fulfillment of the Requirements For the Degree of Doctor of Philosophy

Konstantin Shagin

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

HESHVAN 5769

HAIFA

NOVEMBER 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 thank Prof. Assaf Schuster and Dr. Michael Factor for their guidance and support. Their insights and academic experience proved to be invaluable.

The generous financial help of the Technion is gratefully acknowledged
I dedicate this work to my beloved mother Yekaterina Shagin and my grandparents

Hannah and Yefim
Contents

Abstract xv

List of Symbols xvii

1 Introduction 1
   1.1 Motivation and Goals ................................................. 1
   1.2 The JavaSplit Framework .............................................. 2
   1.3 Contributions of This Thesis ......................................... 5

2 Related Work 7
   2.1 Distributed Computing Frameworks for Java .......................... 7
      2.1.1 Cluster-aware VMs ............................................ 8
      2.1.2 Compiler-based Java DSMs .................................... 9
      2.1.3 Systems Using Standard JVMs ................................. 10
      2.1.4 System comparison summary .................................. 11
   2.2 Instrumentation-based Functional Partitioning ...................... 12
   2.3 Home-based Lazy Release Consistent Distributed Systems .......... 13
   2.4 Fault-Tolerance Schemes for Lazy Release Consistency .......... 14
   2.5 Fault-Tolerant Java Virtual Machines ............................. 15
   2.6 Optimistic Concurrency Control ................................... 16
      2.6.1 Speculative Locking for Shared-Memory Multiprocessors .... 16
      2.6.2 Distributed Transactional Memory ............................. 17
## CONTENTS

3 Instrumentation of Standard Libraries

3.1 Twin Class Hierarchy ........................................... 23
  3.1.1 Root Class ................................................. 27
  3.1.2 Irregular Classes ........................................... 27
  3.1.3 Classes with Special Semantics ............................ 29
  3.1.4 Invalid Method Overriding .................................. 30
  3.1.5 Object Constants ............................................ 30
  3.1.6 Runtime Exceptions ........................................ 31
  3.1.7 Reflection ................................................ 31
  3.1.8 Native Classes ............................................. 32
  3.1.9 Applicability Discussion ................................... 35

3.2 Facilitating Instrumentation of System Classes .......... 36
  3.2.1 Static Instrumentation .................................... 36
  3.2.2 Dynamic Instrumentation .................................. 37

3.3 TCH Overhead Analysis ........................................ 39

3.4 Application of TCH in profiling ............................. 41

3.5 Load-time Instrumentation Frameworks ..................... 44
  3.5.1 Custom Class Loaders ...................................... 44
  3.5.2 JVM Dependent Interception ............................... 45
  3.5.3 Library-based Interception ................................ 45
  3.5.4 Class Loader Independent Interception ................. 45

4 Transparent Distributed Runtime for Java .................. 47

4.1 Distributed Memory Management Protocol .................. 48
  4.1.1 Lazy Release Consistency ................................. 49
  4.1.2 JavaSplit Implementation of LRC ......................... 50

4.2 Distributed Locking Protocol ................................ 58
  4.2.1 Distributed Locking for Unreliable Channels .......... 60
  4.2.2 Correctness Proof ........................................ 61
  4.2.3 Distributed Conditional Waiting ......................... 62
CONTENTS

4.3 Instrumentation for Distributed Execution ................................................. 63
  4.3.1 System Classes ................................................................. 66
  4.3.2 Static Fields ................................................................. 67
  4.3.3 Array Transformation ....................................................... 67
  4.3.4 Rewriting Lock Operations ................................................. 68
4.4 Performance Evaluation ................................................................. 69
  4.4.1 Instrumentation Overhead ................................................... 69
  4.4.2 Communication Latency and Throughput .................................. 71
  4.4.3 Benchmark Applications .................................................... 72
  4.4.4 Windows/Ethernet measurements ........................................ 73
  4.4.5 Linux/Infiniband measurements .......................................... 80

5 Fault-Tolerant Distributed Runtime ....................................................... 83
  5.1 System Model ................................................................. 85
  5.2 Replication Groups ............................................................ 85
  5.3 Persistence of Shared Objects ............................................... 86
  5.4 Thread Checkpointing .......................................................... 88
  5.5 Fault-tolerant Distributed Locks ............................................. 90
  5.6 Correctness Proof ............................................................. 93
    5.6.1 Correctness of the Distributed Locking Protocol .................. 93
    5.6.2 Correctness of the Memory Management Protocol ................. 95
  5.7 Portable Checkpointing for Java Threads .................................... 97
  5.8 Incremental Checkpointing ................................................... 100
  5.9 Performance Evaluation ....................................................... 101

6 Speculative Locking ..................................................................... 109
  6.1 Speculative Lock Acquisition .................................................. 110
    6.1.1 Data Conflict Detection ................................................ 111
    6.1.2 Managing Recoverable States ........................................ 112
    6.1.3 Implementing Efficient Local Rollback .............................. 113
    6.1.4 Rollback Dependencies ................................................. 114
CONTENTS

6.1.5 Speculative Deadlocks ........................................... 115
6.1.6 Speculation Preclusion ........................................... 116
6.2 Speculative Locking in JavaSplit ................................. 117
   6.2.1 Implementing Data Conflict Detection .................... 117
   6.2.2 Implementing Consistent Rollback ......................... 119
   6.2.3 Conditional Waiting for Speculative Locking ............ 120
   6.2.4 Correctness Proof ........................................... 121
6.3 Performance Evaluation ........................................... 124
   6.3.1 Evaluation Goals ........................................... 124
   6.3.2 Theoretical Analysis ...................................... 125
   6.3.3 Application Performance Results ......................... 131
   6.3.4 Micro-Benchmark Results .................................. 138
   6.3.5 Performance Evaluation Summary ......................... 145

7 Transactional Memory vs. Speculative Locking ................. 147
   7.1 Distributed Transactional Memory Protocols ................ 149
   7.2 Transactional Memory vs. Speculative Locking ............. 151
   7.3 Implementing DTM on top of Speculative Locking ........... 154
   7.4 Conclusion .................................................. 156

8 Future Work ..................................................... 157
   8.1 Instrumentation Optimizations ................................. 157
   8.2 Array Partitioning ........................................... 158
   8.3 Single-Writer Consistency Protocols .......................... 158
   8.4 Passing Data Modifications Instead of Write Notices ....... 159
   8.5 Home Migration and Load Balancing ......................... 159
   8.6 Distributed Garbage Collection ................................ 160
   8.7 Passing Lock Ownership in Speculative Mode ............... 160
   8.8 Predictive Data Push ........................................ 162

9 Conclusion ...................................................... 165
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>165</td>
</tr>
<tr>
<td>Hebrew Abstract</td>
<td>i</td>
</tr>
</tbody>
</table>
List of Figures

3.1 A fragment of the class hierarchy in Java, before and after the TCH transformation. The instrumented versions of system classes become user classes. ........................................ 24
3.2 A Java class before and after the TCH transformation. (Although we present source code, the actual transformation can be performed on bytecode as well.) ....................... 25
3.3 Instrumentation process. The code inserted during the second phase uses original class names. ................................................................. 26
3.4 High-level view of class hierarchy transformation ........................................ 26
3.5 A possible wrapper for a Java integer array. Accessor methods are final to enable inlining. ................................................................. 28
3.6 Implementation of an instrumented native class using the delegation pattern (in Java-like coding style). Note the string conversion in the constructor. ......................... 32
3.7 Implementation of several originally native methods ........................................ 34
3.8 Dynamic class loading procedure .................................................................. 38
3.9 Summary of existing interception techniques for JVM ................................. 44
4.1 Example of JavaSplit distributed memory management protocol operation .... 51
4.2 Distributed locking protocol example ............................................................ 59
4.3 Conditional waiting for distributed token-based locking algorithms ................ 64
4.4 An instrumented read access of the field myIntField in class A. The instructions in bold are added by the instrumentation. If the object is valid for read, i.e., the value of the state field is not 0, only the first 3 added instructions are executed. ............ 65
4.5 Utility class members added by the distribution transformation ....................... 65
4.6 Execution times (sec) and speedups ............................................................... 74
4.7 Execution times (sec) and speedups ........................................... 75
4.8 Execution times (sec) and speedups ........................................... 76
4.9 Execution times (sec) and speedups ........................................... 77
4.10 SparseMatmult results in the Windows/Ethernet setting ....................... 78
4.11 Comparison of Infiniband and Ethernet runs on Linux. Infiniband exhibits linear speedup ................................................................. 81
4.12 Breakdown of the Linux runs. The black and the gray indicate the time spent in data misses and in local computation respectively. .................. 81

5.1 Update request processing by the replicated shared object repository ....... 87
5.2 Checkpointing ........................................................................... 89
5.3 Token loss detection by monitoring of the lock transfer counters .......... 92
5.4 Java thread state capturing and reestablishment with Brakes .......... 98
5.5 Normalized execution times of single-threaded runs. The value 1 on the vertical axis corresponds to the execution time of the original application on a single JVM ........................................ 102
5.6 Benchmark execution times (in seconds). The dashed line indicates the execution time of the original benchmark using two threads in a single JVM running on a dual-processor machine. The horizontal axis of each graph represents the number of processors ........................................ 104
5.7 Benchmark execution times (in seconds) ....................................... 105
5.8 Benchmark execution times (in seconds) ....................................... 106
5.9 Time breakdown of JavaSplit and JavaSplit-FT on 28 processors ........ 107

6.1 Blocking lock acquisition vs. speculative .................................... 111
6.2 Speculative deadlock. Neither process can transfer lock ownership ...... 116
6.3 Pseudo-code inserted before a read access .................................... 118
6.4 Conditional wait loop ................................................................ 121
6.5 Model application - worker thread code ...................................... 125
6.6 Flow of a queued lock acquisition .............................................. 128
6.7 Throughput and worker breakdown for SPECJBB* (low acq. frequency) .. 133
6.8 Throughput and worker breakdown for SPECJBB* (high acq. frequency) .. 134
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.9</td>
<td>TSP throughput and worker block time breakdown</td>
<td>136</td>
</tr>
<tr>
<td>6.10</td>
<td>Hashable throughput</td>
<td>137</td>
</tr>
<tr>
<td>6.11</td>
<td>Average lock acquisition latency with varying network latencies</td>
<td>139</td>
</tr>
<tr>
<td>6.12</td>
<td>SPECJBB* throughput with varying data conflict probabilities</td>
<td>140</td>
</tr>
<tr>
<td>6.13</td>
<td>Varying preemptive checkpointing frequency</td>
<td>140</td>
</tr>
<tr>
<td>6.14</td>
<td>Speculative deadlock probability as a function of speculation quota</td>
<td>144</td>
</tr>
<tr>
<td>6.15</td>
<td>SPECjbb* throughput as a function of speculation quota</td>
<td>144</td>
</tr>
<tr>
<td>7.1</td>
<td>Write access</td>
<td>155</td>
</tr>
<tr>
<td>8.1</td>
<td>Passing lock ownership in speculative mode</td>
<td>161</td>
</tr>
<tr>
<td>8.2</td>
<td>Predictive Data Push</td>
<td>163</td>
</tr>
</tbody>
</table>
Abstract

Interconnected commodity workstations may provide the high processing capacity required to solve many current computational problems. This thesis presents a parallel processing framework that allows utilization of available workstations with minimal programming and management efforts. The proposed system, which we call JavaSplit, is a distributed runtime environment for standard multithreaded Java programs. It is resilient to multiple node failures and therefore preserves the integrity of the computation when workstations abruptly terminate their participation. JavaSplit hides the distributed and unreliable nature of the underlying environment from the programmer and can execute preexisting Java programs.

Portability is one of JavaSplit's most notable features, which distinguishes it from the existing distributed runtime environments that provide single system image. Each JavaSplit node carries out its part of the computation using nothing but its local standard Java Virtual Machine (JVM). Hence, the portability of JavaSplit is not only equivalent to that of the Java language, but the runtime can also execute an application on a heterogeneous set of workstations. The portability is realized by instrumenting the program bytecode to enable distributed execution and thread checkpointing. In order to support portable instrumentation of Java system classes (the library classes that are incorporated into the JVM), JavaSplit employs a novel instrumentation method, which we call the Twin Class Hierarchy approach.

JavaSplit distributed memory management protocol improves over those in the existing state-of-the-art systems. First, it reduces the number of memory synchronization operations to the minimum required to implement the designated memory model. Second, it introduces a novel method of version bookkeeping in which the version size is not proportional to the number of threads. Consequently, this method significantly reduces the space occupied by the versions (attached to each object) and
the bandwidth dedicated to their transfer over the wire. Finally, JavaSplit memory management protocol incorporates a scheme for prompt disposal of outdated memory consistency metadata, which discards most stale items without global thread cooperation.

The innovation of the JavaSplit fault-tolerance scheme is mostly in the independent treatment of thread states and shared data, which are considered separate entities as far as resilience to failures is concerned. This modular approach allows to reason about inter-consistency of these entities and facilitates optimizations, such as data prefetching. Another notable part of the proposed scheme is the novel fault-tolerant token-based distributed locking protocol, which allows on-the-fly token recovery without global cooperation of nodes.

JavaSplit checkpointing capabilities enable employment of speculative locking, an optimistic concurrency control mechanism, which removes the lock acquisition from the execution critical path and enables concurrent execution of critical sections protected by the same lock. This thesis presents JavaSplit speculative locking scheme and compares its properties to those of Transactional Memory. To the best of our knowledge this is the first work to suggest employment of speculative locking in a general-purpose distributed runtime.
List of Symbols

Abbreviations

JS  
OS  
JVM  
JIT  
TCH  
OOC  
IOC  
RC  
LRC  
HLRC  
SOR  
HB  
$op_1 \rightarrow op_2$  
SL  
DSL  
SMP  
RAM  
JGF  
GHz  
GB

JavaSplit  
Operating System  
Java Virtual Machine  
Just-In-Time Compiler  
Twin Class Hierarchy  
Original Object Class  
Instrumented Object Class  
Release Consistency  
Lazy Release Consistency  
Home-Based Lazy Release Consistency  
Shared Object Repository  
Happens-Before  
$op_1$ Happened-Before $op_2$  
Speculative Locking  
Distributed Speculative Locking  
Symmetric Multiprocessing  
Random Access Memory  
Java Grande Forum  
Gigahertz  
Gigabyte
List of Symbols

ms  Milliseconds
STM  Software Transactional Memory
DTM  Distributed Transactional Memory
DVM  Distributed Virtual Machine
TSP  Traveling Salesman Problem

Java Instructions
GETFIELD  Fetch field from object
AALOAD   Load reference from array
IALOAD   Load int from array
DALOAD   Load double from array
Chapter 1

Introduction

1.1 Motivation and Goals

Collections of interconnected workstations are widely considered a cost-efficient alternative to supercomputers. Nevertheless, there is no general parallel processing framework that is accepted and used by all. It is hard to adapt existing solutions because they often target a specific platform, require special networking hardware, or enforce complicated programming conventions. As a result, many organizations choose to implement custom frameworks which, for the same reasons, cannot be used by others. The effort invested in devising such frameworks compromises the cost benefit of using off-the-shelf computing resources.

Another difficulty in using a network of workstations for parallel processing is that a participating workstation may suddenly become unable to continue its part of the computation. This can be caused by a failure, e.g., a power outage, or by user intervention, e.g., shutdown or restart. These events result in the immediate loss of the data and processes located on the node. Unless the application or the runtime that executes it are fault-tolerant, the above actions will surely violate the integrity of the computation. This problem is especially acute in non-dedicated environments, where the runtime system has little control over the participating machines.

The goal of this thesis is to propose a general framework for parallel computing that addresses the aforementioned issues. Hence, the proposed framework should have the following qualities. First, it should achieve a performance gain which will be sufficient for justifying the cost and maintenance
of multiple processing nodes. Second, it should be available in the most common combinations of hardware, operating systems and networks. Third, it should provide a convenient and easy-to-adopt programming model. Finally, it should be resilient to multiple node failures.

1.2 The JavaSplit Framework

The proposed framework, which we call JavaSplit [29, 31, 28], is essentially a fault-tolerant distributed Java Virtual Machine (JVM) that runs on a plurality of standard JVMs. It executes standard multithreaded Java programs on a collection of available interconnected workstations. JavaSplit distributes the threads of a Java program among the processing nodes and maintains the consistency of shared data. It is resilient to multiple, possibly simultaneous node failures.

The programming model of JavaSplit is standard Java, a popular computer language. The programmer writes a multithreaded application assuming that there is a main memory accessible by all threads. There is no need to use any networking constructs, such as sockets, or employ nonstandard data passing libraries in order to achieve cooperation among threads that are running on different machines. JavaSplit makes the distributed and unreliable nature of the underlying hardware completely transparent to the programmer. Thus, the framework can execute preexisting applications.

Since each node carries out its part of a distributed execution on a standard JVM, the portability of JavaSplit is equivalent to the portability of a Java program. A node can use any JVM implementation, as long as it complies with the JVM specification [64]. This means not only that JavaSplit can execute on most existing platforms, but also that it can employ a heterogeneous set of workstations and JVMs in a single execution. Moreover, a node can locally optimize the performance of its JVM, e.g., via a just-in-time compiler. Unlike systems that utilize specialized networking hardware [7, 103], JavaSplit employs IP-based communication. It accesses the network through the Java socket interface. The use of standard JVMs in conjunction with IP-based communication enables JavaSplit to perform computation on any given heterogeneous set of machines interconnected by an IP network, such as the Internet or the intranet of a large organization.

Transparency, portability and the ability to capture and restore a thread state are all achieved by means of bytecode instrumentation. JavaSplit instruments the program to intercept events that are
1.2. THE JAVASPLIT FRAMEWORK

interesting in the context of distributed execution, e.g., synchronization, creation of new threads, and accesses to objects referenced by multiple threads, henceforth shared objects. The bytecode rewriter also augments the program and a subset of runtime classes with thread checkpointing capabilities. Thus, combined with the runtime modules, which are also written in Java, the resulting bytecode can be executed on any standard JVM.

The consistency of the shared objects is maintained by a protocol inspired by *Home-Based Lazy Release Consistency* (HLRC) [112]. The node on which the object is created manages the object’s master copy. The threads that need to use the object create local (cached) copies, based on that copy. At certain synchronization points the modifications are flushed from the cached copies to the master copy. The memory model implemented by the above protocol is *Lazy Release Consistency* (LRC) [55] which is compatible with the new Java Memory Model [53]. The proposed distributed object management scheme incorporates support for distributed synchronization, including the Java-specific operations, e.g., wait and notify.

JavaSplit distributed memory management protocol improves over those in the existing LRC implementations. First, it reduces the number of memory synchronization operations to the minimum required to implement LRC. Second, it introduces a novel method of version bookkeeping, which is considerably more scalable than its traditional counterparts. Finally, JavaSplit incorporates a scheme for local disposal of a large portion of the outdated memory consistency metadata.

Resilience to multiple node failures is accomplished by checkpointing and replication. At certain execution points each application thread independently checkpoints itself. A snapshot of a thread does not include the application’s shared objects. Shared objects and thread states are made persistent by replication in the volatile memory of other nodes. The key idea of our fault tolerance scheme is to capture the states of the application threads in such a way as to guarantee that rollback of any thread to its latest saved checkpoint does not violate system consistency. If this requirement is fulfilled, a failed thread will recover by simply restarting from its most recent checkpoint. Hence, the presented scheme keeps only a single checkpoint per thread. The support for fault-tolerance can be entirely disabled, when unneeded. In this case JavaSplit executes the application without any overhead related to resilience to failures.

Scalability considerations played an important role in the design of the fault-tolerance scheme. As a result, neither failure-free execution nor recovery from a failure require global cooperation of
nodes. Moreover, recovery never results in rollback of non-failing nodes. During recovery, a non-failing node continues its normal execution, unless it wishes to acquire a lock held by a failed thread. In this case, it must wait until the thread is restored.

The fault-tolerance scheme is tightly integrated into the shared object space management scheme. We exploit the fact that in the employed protocol a thread may affect the data observed by remote threads only during lock ownership transfer. By checkpointing a thread when it is about to transfer ownership, we ensure that the most recent thread snapshots are inter-consistent.

The cost of creating and saving a thread snapshot is reduced by an incremental checkpointing scheme. Instead of serializing all application objects (accessed by the thread) into a snapshot, JavaSplit puts there only the new and the modified. This significantly reduces the checkpointing time as well as the the snapshot size.

JavaSplit checkpointing capabilities allow us to optimize its performance by employment of speculative locking. During lock acquisition, instead of waiting for a lock to be properly acquired, a process optimistically continues executing. This removes the lock acquisition from the execution critical path and enables concurrent execution of critical sections protected by the same lock. In case a data conflict is detected, the process rolls back to a state preceding the lock operation.

JavaSplit requires minimal management efforts. A Java-enabled workstation that wishes to participate in the computation does not need to install any software. It merely needs to run a thin Java client that notifies the system core that the current workstation is available. The application and runtime classes are loaded from the network using the customizable Java class loading mechanism. Moreover, the client program can be incorporated in a Web page applet. Consequently, a workstation completely unaware of JavaSplit can join the runtime simply by pointing its Java-enabled browser to that page.

The proposed system is suitable for utilization of idle interconnected computing resources. The portable client program mentioned above can be incorporated into a screen saver. Since the system is resilient to node failures, it will remain consistent even if a machine’s user reboots it or disconnects it from the network. Moreover, fault-tolerance allows us to minimize a workstation’s response time to its native user by treating node reclamations as node failures. When a user’s presence is detected (e.g., by monitoring input devices), we can immediately kill all local runtime processes and release all the memory they occupy.
1.3 Contributions of This Thesis

The main theoretical contributions of this thesis are in design of the following protocols and methods.

1. A method for safe instrumentation of standard libraries in object-oriented languages, which we call the Twin Class Hierarchy approach [30].

2. An object-based distributed shared memory protocol that improves state-of-the-art home-based protocols, e.g., [55, 89, 112]

3. A generic fault-tolerance scheme for distributed shared memory protocols that implement Lazy Release Consistency.

4. A lightweight fault-tolerant distributed locking protocol (with support for conditional waiting).

5. A generic speculative locking protocol for fault-tolerant distributed runtime systems.

The primary practical contribution of this thesis is JavaSplit, a fault-tolerant distributed runtime system for Java applications. This is the first distributed runtime for Java to combine portability with transparency. Moreover, it is the first system to combine the above two properties with fault-tolerance.
Chapter 2

Related Work

The JavaSplit framework is closely related to research in a number of domains. Being able to execute a standard Java application, it is akin to systems implementing a distributed Java Virtual Machine. The portable design makes JavaSplit comparable to other instrumentation-based distributed frameworks as well as to systems that execute cluster-aware parallel applications on a collection of standard JVMs. By incorporating a novel home-based lazy-release-consistent distributed memory management scheme JavaSplit extends a line of works on distributed shared memory protocols. Due to its resilience to node failures, JavaSplit is also related to fault-tolerant distributed systems and to fault-tolerant Java Virtual Machines. The support for speculative locking in a distributed setting places it alongside the distributed transactional memory systems.

In this chapter we overview the most prominent works in the above areas, classify them, and compare the properties of the systems they describe to those of JavaSplit.

2.1 Distributed Computing Frameworks for Java

The distributed shared memory frameworks for Java can be classified by their implementation into three categories: (i) cluster-aware VMs; (ii) compiler-based DSM systems, and (iii) systems using standard JVMs.
2.1.1 Cluster-aware VMs

Cluster-aware virtual machines are the systems whose nodes execute their part of distributed execution on a non-standard dedicated JVM. In most cases, such a non-standard JVM is a result of modifying an existing standard JVM, rather than creating a new one from scratch. The most prominent examples of cluster-aware JVMs are Java/DSM [111], Cluster VM for Java (former cJVM) [8, 9], JESSICA2 [107, 113] and CoJVM [67, 68].

In Java/DSM the local VM is very similar to a standard JVM, except that all objects are allocated on an existing C-based software DSM, called TreadMarks [54]. Similar to our work, TreadMarks implements LRC. The single system image provided by Java/DSM is incomplete: a thread's location is not transparent to the programmer, and the threads cannot migrate between machines. In contrast, Cluster VM for Java, JESSICA2 and CoJVM provide a complete single system image of a traditional JVM.

CoJVM incorporates a DSM that employs an object-based variant of the Home-based Lazy Release Consistency protocol (HLRC) [112]. Hence, the memory management scheme in CoJVM is closely related to that of JavaSplit, which draws from HLRC and is also object-based. In contrast to JavaSplit which improves over HLRC, CoJVM employs the original HLRC implementation [112]. Unlike JavaSplit that uses standard IP based communication, CoJVM nodes connect through the Virtual Interface Architecture (VIA) [106].

Instead of using a DSM, Cluster VM for Java uses a proxy design pattern with caching and object migration optimizations. JESSICA2 uses a home-based global object space (GOS) to implement a distributed Java heap. JESSICA2 possesses many desirable capabilities, e.g., support for load balancing via thread migration, adaptive migrating-home protocol for the GOS and a dedicated JIT compiler. The latter feature distinguishes JESSICA2 from most of the similar works. For instance, Cluster VM for Java, CoJVM and Java/DSM are unable to use a standard JIT compiler, and do not implement a dedicated one. By contrast, the systems that use a standard JVM, including JavaSplit, are able to utilize the JIT compiler supplied with the local JVM participating in the execution.

The great efficiency advantage of the cluster-aware virtual machines in comparison to systems using standard JVMs, is the ability to access machine resources, e.g., memory and network interface, directly rather than through the JVM. On the downside, due to modification of a node's local JVM,
none of the above works possesses true cross-platform portability. Without the need to preserve portability, the cluster-aware VMs allow usage of efficient non-standard networking hardware, which further improves their performance.

2.1.2 Compiler-based Java DSMs

Compiler-based DSM systems compile the sources or the bytecodes of a Java program into native machine code while adding calls to DSM handlers. There are only two compiler-based systems known to us: Hyperion [7, 42] and Jackal [103]. Both systems support standard Java and do not require any changes in the programming paradigm.

Hyperion translates Java bytecodes to C source code and then compiles the C source using a native C compiler. The DSM handlers are inserted during the translation to C. The Java-bytecode-to-C translator performs various optimizations in order to improve the performance of the DSM. For example, if a shared object is referenced in each iteration of a loop (that does not contain synchronization), the code for obtaining a locally cached copy of the object is lifted out of the loop. Hyperion employs existing DSM libraries to implement its DSM protocol, and is able to use various low-level communication layers.

Jackal combines an extended Java compiler and runtime support to implement a fine-grain DSM. The compiler translates Java sources into Intel x86 code rather than the Java bytecode. The Jackal compiler stores Java objects in shared regions and augments the program it compiles with access checks that drive the memory consistency protocol. Similarly to Hyperion, it performs various optimizations, striving to achieve a more efficient distributed execution [104]. Jackal incorporates a distributed garbage collector and provides thread and object location transparency.

Under certain circumstances, the usage of a custom source-level Java compiler (like the one used by Jackal) is disadvantageous. First, such compilers need to be updated in case the Java language changes or is augmented with additional language constructs. Good examples of such changes are the inner classes, introduced in JDK 1.1 and the generics, introduced in JDK 1.5. Since the changes in the Java language do not necessarily result in changes in the JVM bytecode, the systems whose input is bytecode are less susceptible to the evolution of the Java language. Second, in contrast to bytecode, the required application source code can be unavailable, for example due to confidentiality considerations.
CHAPTER 2. RELATED WORK

The portability of compiler-based systems is no better than the portability of cluster-aware VMs. However, this category has several performance advantages. First, the usage of a dedicated compiler allows performing various compiler optimizations, which have potential to significantly improve performance. Second, since the application is compiled to machine code, the speed of a local execution is increased without requiring a JIT compiler. Finally, similarly to the cluster-aware VMs, these systems can directly access the machine’s memory and network interface.

2.1.3 Systems Using Standard JVMs

There are quite a few Java-based distributed shared memory/objects frameworks that utilize standard JVMs, e.g., [83][92][45][16][11]. However, most such systems significantly deviate from the standard multithreaded Java programming paradigm. There are a few systems such as JavaParty [83, 44, 43], JDSM [92] and Terracotta [14] that are close enough to providing multithreaded Java programming paradigm in order to be considered here. However, none of the existing systems, except our JavaSplit, is able to execute standard, possibly pre-existing, Java programs without additional input from the programmer.

JavaParty supports distributed parallel programming by extending Java with a preprocessor and a runtime. The main modification to the Java programming paradigm is the introduction of a new reserved word, remote, to indicate classes that should be distributed across machines. The source code is transformed into regular Java code plus RMI hooks which are passed to the RMI compiler. The single system image is further reduced by the fact that the programmer must also distinguish between remote and local method invocations, due to the differing argument passing conventions.

In JDSM, access checks, in the form of method invocation to memory consistency operations, are inserted manually by the user (or possibly a higher-level program translator) for field read/write accesses. JDSM requires that an input program is an SPMD-style multithreaded program. Moreover, the programmer must use special classes provided by JDSM and mark the shared objects.

Terracotta employs a server-based architecture to execute a Java program on a cluster. All shared data is stored on the server which also performs various management tasks such as distributed locking and memory consistency control. Due to the centralized design, Terracotta cannot allow distributed management of the entire Java heap. Hence, it requires that the user marks the remote data using XML-like configuration files, which are fed to the system along with the application classes.
2.1. DISTRIBUTED COMPUTING FRAMEWORKS FOR JAVA

Terracotta further deviates from the Java programming model by requiring that the application does not create Java threads; parallel execution is achieved by starting the same program on multiple JVMs. The input application is seamlessly modified using an extended aspect-oriented programming framework. Unlike JavaSplit, Terracotta is unable to instrument the Java system classes, which imposes a limitation on object sharing. On the bright side, Terracotta provides data persistence by saving the state of the server.

There are several advantages in using standard JVMs. First, the system can use heterogenous sets of nodes. Second, each node can locally optimize its performance, e.g., via a JIT. Third, a local garbage collection can be utilized to collect unneeded local objects that are not referenced from other nodes. The main drawback of systems in this category is a relatively slow access to the node’s resources. Since JavaSplit uses only standard JVMs, it possesses all the mentioned above features. In addition, it allows programmer to write standard multithreaded Java applications, without being aware of the distributed nature of the underlying system and without imposing any programming restrictions. To the best of our knowledge, this combination of portability with transparency is unique to JavaSplit.

2.1.4 System comparison summary

Having presented an overview of the distributed runtime systems for Java, we now summarize their features in Table 2.1 using the following criteria.

**Transparency.** The system is considered transparent if both the programmer and the person that starts the execution can be completely unaware of the runtime being distributed.

**Portability.** A system is considered to be portable if it is at least as portable as Java.

**Just-in-time compiler.** Most presented systems that do not use a standard JVM, do not have a JIT. Only JESSICA2 implements a dedicated one.

**Garbage Collection.** The presented systems either do not implement GC at all, implement a fully-fledged distributed GC, or perform local GC, i.e., only collect objects that are not shared. The latter may be performed either by the standard native GC (designated as ‘native’), or by a dedicated one (designated as ‘local’).
Table 2.1: Feature Summary.

<table>
<thead>
<tr>
<th>DSM</th>
<th>Transparency</th>
<th>Portability</th>
<th>JIT</th>
<th>Garbage coll.</th>
<th>Shared Mem.</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java DSM</td>
<td>incomplete</td>
<td>X</td>
<td>X</td>
<td>distributed</td>
<td>page</td>
<td>bytecode</td>
</tr>
<tr>
<td>CVM for Java</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>local</td>
<td>proxy</td>
<td>bytecode</td>
</tr>
<tr>
<td>JESSICA2</td>
<td>V</td>
<td>Y</td>
<td>dedicated</td>
<td>X</td>
<td>object</td>
<td>bytecode</td>
</tr>
<tr>
<td>CoJVM</td>
<td>V</td>
<td>Y</td>
<td>N/A</td>
<td>local</td>
<td>page</td>
<td>bytecode</td>
</tr>
<tr>
<td>Hyperion</td>
<td>V</td>
<td>X</td>
<td>N/A</td>
<td>distributed</td>
<td>fine</td>
<td>source</td>
</tr>
<tr>
<td>J2Dal</td>
<td>X</td>
<td>Y</td>
<td>native</td>
<td>native</td>
<td>proxy</td>
<td>source</td>
</tr>
<tr>
<td>Terracotta</td>
<td>X</td>
<td>Y</td>
<td>native</td>
<td>native</td>
<td>object</td>
<td>bytecode</td>
</tr>
<tr>
<td>JavaSplit</td>
<td>V</td>
<td>Y</td>
<td>native</td>
<td>native</td>
<td>object</td>
<td>bytecode</td>
</tr>
</tbody>
</table>

**Shared memory implementation.** There are two main options to implement shared memory here: to utilize a distributed shared memory or to employ the proxy pattern. The utilized DSMs can be also classified to fine-grain/course-grain or object-based/page-based.

**Input.** Some presented systems need the source code of the application, while the others require only the bytecodes.

### 2.2 Instrumentation-based Functional Partitioning

There are functional partitioning frameworks [100, 98] that, similarly to JavaSplit, rewrite application bytecode for distributed execution. Their goal however is not high performance computing; they aim at splitting an application into distinct entities running on the most functionally suitable sites.

**J-Orchestra** [100, 66, 101] may execute a computation intensive application with a GUI on two machines: one with a fast processor and another with a graphical screen. Unlike JavaSplit, which employs an object-based distributed shared memory and monitors accesses to the shared data, J-Orchestra uses proxies to access remote objects. It substitutes method calls and direct object references with remote method calls and proxy references respectively.

The key difference in comparison to JavaSplit is in the treatment of Java system classes with native dependencies (i.e., classes that have native methods or can be accessed from such classes). In J-Orchestra they are perceived as unmodifiable code and therefore cannot be rewritten to access remote objects through a proxy. This results in certain constraints on the data placement. All
instances of a system class with native dependencies are placed on the same node. Moreover, any class that can be referenced from it must also be placed on that node. Due to the strong class dependencies within Java packages, this usually results in partitions that coincide with package boundaries. In contrast, JavaSplit supports arbitrary partitioning due to its ability to handle system classes with native dependencies.

Addistant [98] instruments Java bytecode at load-time using the Javassist [18] framework. Like J-Orchestra, it employs the remote proxy model to bridge between objects on different nodes. Due to the special status of the system classes within the JVM, Addistant is unable to transform them at load-time. Therefore, it introduces several bytecode rewriting workarounds, the applicability of which depends on the type of interaction between the classes. The novel instrumentation approach employed by JavaSplit would resolve this issue.

2.3 Home-based Lazy Release Consistent Distributed Systems

Home-based Lazy Release Consistency (HLRC) [112, 51] is a distributed shared memory protocol. The memory model implemented by this protocol is Lazy Release Consistency (LRC) [55]. (Note that the former is a protocol while the latter is a memory model.) HLRC has been shown to have a number of advantages in comparison to the homeless implementations of LRC [110].

Many works suggest improvements of the design and the implementation of the original HLRC protocol presented in [112]. In [85] and [52] HLRC is implemented over VIA, while in [78] and [77] it is implemented over Infiniband. A large number of works, such as [76, 108, 32, 56, 81, 22], propose home migration schemes, in which the home of a coherence unit changes according to the application access pattern. Petit et al. [82, 87] propose adaptive schemes for eager propagation of modifications. Some works [63, 6] describe data prefetching optimizations. Blas et al. [12, 89] extend HLRC for efficient execution on SMP clusters.

The distributed memory management protocol in JavaSplit is a home-based implementation of the LRC memory model and therefore bears some similarity to the original HLRC protocol. Like [63] and [6], it incorporates a data prefetching protocol. The latter is a lightweight tunable scheme based on the object reference graph, which takes into account not only the application access pattern but also the network bandwidth.
CHAPTER 2. RELATED WORK

JavaSplit improves over the original HLRC and its derivatives. First, it performs the costly memory synchronization operations only when transferring lock from one application thread to another. This reduces the frequency of these operations at least by the factor of 2, since the prior art requires that they are performed on each acquire and when lock requests are serviced. Second, the JavaSplit protocol replaces the use of vector timestamps, whose size is proportional to the number of threads, with timestamps whose size typically does not exceed the size of two integers. Since a timestamp is attached to each coherency unit, this type of version bookkeeping significantly reduces both the memory occupied by the timestamps and the bandwidth used by their transfer in the network. Finally, JavaSplit manages update notifications received throughout the thread lifetime in a way that allows a large portion of notifications to be removed without requiring a system-global garbage collection procedure.

Unlike [85, 52, 78, 77] which employ high-performance networking interfaces, JavaSplit uses standard socket-based IP communication, in order to preserve its portability.

2.4 Fault-Tolerance Schemes for Lazy Release Consistency

There have been several works that augment a LRC-based DSM with a transparent fault tolerance mechanism. Most of these works, e.g., [25, 96, 109], use a homeless implementation of LRC as their base memory consistency protocol.

Sultan et al. [94, 95] follow a log-based approach that allows to tolerate a single node failure in a home-based, lazy release consistent DSM. They use volatile logging of protocol data combined with independent checkpointing to stable storage. Since their scheme is log-based, this work focuses on how to dynamically optimize log trimming and checkpoint garbage collection to efficiently control the size of the logs and the number of checkpoints kept. Kongmanwattana et al. [59] also use a log-based mechanism to tackle the same problem. The main disadvantage of both schemes is that, during recovery, the number of nodes that need to cooperate with the failing one is unbounded. Moreover, in the latter scheme, recovery may require some of the non-faulty processes to roll back to an earlier checkpoint. In contrast, recovery in our scheme does not require cooperation of other nodes. A thread is simply restarted from its latest checkpoint. Another important difference is that in our system the shared data is a part of a checkpoint, which significantly reduces the
2.5. FAULT-TOLERANT JAVA VIRTUAL MACHINES

checking time and snapshot size.

Park et al. [79, 80] present two additional log-based fault-tolerance schemes for HLRC. A distinguishing feature of this work in comparison to the prior art is that it can tolerate a number of concurrent node failures. On the downside, the nodes that have failed concurrently need to cooperate in order to restore the state of the shared data.

Christodoulou et al. [20, 21] extend a home-based LRC protocol designed for clusters of SMP nodes. As in our system, data persistence is achieved through replication in volatile memory of nodes and checkpointing is integrated into synchronization operations. Despite the similarity of the basic ideas, their system can tolerate only a single node failure.

2.5 Fault-Tolerant Java Virtual Machines

Friedman et al. [33] and Alvisi et al. [75] describe the design and implementation of Java Virtual Machines resilient to node fail-stop failures. These works are comparable to ours by the virtue of their ability to reliably execute a standard Java application.

In [33], JikesRVM [69] is extended with support for active replication. The primary VM sends execution frames to the backup. Based on this information, the backup can instantly resume execution in the case the primary fails. During recovery, the replica restores file and socket descriptors. This approach allows to sustain more than one failure by maintaining a number of replicas using a group communication protocol.

In [75], Sun JDK 1.2 is augmented with fault-tolerance capabilities. In contrast to [33], the backup is cold, i.e., it logs recovery data rather than maintains an active replica. In the case of a failure, the application is replayed from the most recent checkpoint using the logged information about the non-deterministic events. This results in a longer recovery procedure than in [33]. Moreover, unlike [33], this approach cannot employ a JIT compiler, which significantly slows down the failure-free execution.

Compared to JavaSplit fault-tolerance capabilities, the above works provide a more efficient and somewhat more accurate scheme in the context of a single JVM. First, JavaSplit executes each application thread in a separate JVM process, which increases the overhead of running multithreaded programs on a single machine. Second, the JavaSplit fault-tolerance scheme is by far less efficient in
the presence of frequent synchronization operations than those employed by the above fault-tolerant JVMs. It requires that a thread checkpoint is taken whenever one thread transfers lock ownership to another. Finally, JavaSplit does not address fault-tolerance in the context of I/O operations. Clearly, JavaSplit targets an entirely different mode of employment.

2.6 Optimistic Concurrency Control

Speculative lock acquisition and transactional memory have its roots in optimistic concurrency control (OCC) [62, 99], a popular methodology to increase the throughput of database systems. Although optimistic concurrency control have been employed in the context of distributed databases, to the best of our knowledge, it has not been employed in general-purpose distributed runtime systems. Thus, the JavaSplit distributed speculative locking protocol and the recent work on distributed transactional memory [13, 50, 60, 70] adapt OCC to a new domain. Below we describe the works most closely related to ours in this context: (i) speculative locking for shared-memory microprocessors and (ii) distributed transactional memory.

2.6.1 Speculative Locking for Shared-Memory Multiprocessors

There have been a number of works that suggest employment of speculating locking to improve performance of parallel applications on shared-memory multiprocessors [74, 84, 86]. These works describe micro-architectural hardware extensions whose purpose is to enable rollbacks, detect collisions, and ensure speculative writes are not propagated to other threads.

There are several important differences between the above works and our protocol. First, they require that the thread state is checkpointed before each speculative acquire. Second, in contrast to our scheme, 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. If these limitations are exceeded the thread either blocks or rolls back. Finally, although the multiprocessor schemes allow speculating threads to execute past the release operation, they prohibit speculating on several locks simultaneously. Hence, a speculating thread that needs to acquire another lock performs a non-speculative acquisition or blocks. The latter property renders these protocols less optimistic than ours, because they must validate one speculation before making the next one.
2.6. OPTIMISTIC CONCURRENCY CONTROL

In Speculative Lock Elision (SLE) [84], 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 misspeculation, the memory updates are buffered, while the register state is either checkpointed or preserved using a reorder buffer, if available.

The Speculative Synchronization technique proposed in [74] allows threads not only to speculatively acquire locks but also to 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. 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 [86] 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, conflicting accesses resulting in a misspeculation 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 misspeculations, 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.

2.6.2 Distributed Transactional Memory

Transactional Memory [49] simplifies parallel programming for shared-memory multiprocessors by allowing a group of load and store instructions to execute in an atomic way without employment of locks. The general idea of the transactional memory model had been extended to software [90, 47, 72]. Software Transactional Memory (STM), first proposed in [90], provides a similar programming
paradigm yet does not require any hardware support. Its hardware independence allows a more flexible implementation.

Recently, the community has started showing interest in transactional memory on a cluster of interconnected processors [50, 70, 60, 13]. In the absence of physical shared memory and built-in cache coherence protocols, the distributed transactional memory systems synchronize access to shared data using message passing and remote procedure calls.

In the above schemes, a transaction coordinates its accesses to the shared data with remote peers either during execution of the open operation [13, 50] or during commit [13, 70, 60]. In both cases, a transaction sends at least one remote request and waits for a reply. For ease of reference, we classify these schemes as blocking-open and blocking-commit, respectively. While in the latter the commit operation always induces a blocking remote request, the open operation in the former may be local, provided that the current node has a valid cached copy of the object. However, due to the transactional "strict" consistency model cached copies are often invalidated.

It seems, that in the above distributed transactional memory systems, henceforth DTM$s$, a thread spends a non-negligible portion of its time waiting for replies from remote nodes, which decreases the overall throughput of the system. The commit operation is performed once per transaction, while the open operations is performed each time a transaction accesses an object for the first time. The resulting remote requests are placed on the execution critical path due to a design that requires validating a transaction before it commits. In contrast, our distributed speculative locking scheme is capable of detecting conflicts in parallel to the computation. In addition, unlike some of the above schemes, it does not employ centralized and broadcast-based techniques.

It should be noted that without a proper performance comparison, which is beyond the scope of this work, it is hard to determine the relationship between performance of distributed transactional memory and distributed speculative locking. The advantages of the latter may be shadowed by the overhead of mispredictions. Moreover, the above performance shortcomings of distributed transactional memory are not necessarily inherent and hence they may be resolved by a more optimistic design. It should also be noted that in addition to performance there are other considerations such as the programming paradigm, memory model and compositability.
Chapter 3

Instrumentation of Standard Libraries

Code instrumentation is the act of modifying the code of an existing application by inserting new code statements and modifying or deleting existing ones. Instrumentation may be applied to source code as well as to compiled code. In the latter case, it may be applied statically, before the execution begins, or dynamically at run time.

Code instrumentation is a powerful mechanism for understanding and modifying program behavior. It is employed in various fields, including debugging, logging, visualization, access control, performance evaluation, distributed computing, and aspect-oriented programming. The latter can be used in any of the other fields. In JavaSplit, bytecode instrumentation is employed to enable distributed execution as well as to support thread state capture and reestablishment.

Many modern object-oriented languages, e.g., Java, C#, Eiffel, Smalltalk, O’Caml and Objective-C, supply a rich set of reusable core classes, known as standard class libraries or system classes. These classes improve the usability of a language, allowing the programmer to concentrate on the higher-level tasks. A subset of standard classes provides convenient interfaces to operating system facilities, e.g., I/O and networking. Henceforth, we will use the terms system classes and user classes to designate the standard library classes and user-defined classes, respectively.

In many cases, instrumentation requires modifying all the code used by the original program, including the system classes. For example, in JavaSplit, every class used by the application must be augmented with data access checks, in order to preserve the integrity of distributed execution. This applies to all Java system classes, such as java.util.HashMap, java.io.File,
and `java.lang.StringBuilder`. Support for thread checkpointing in JavaSplit has similar requirements. Any method that can be located on the execution stack during thread state capture must be instrumented.

Instrumentation of system classes is problematic in a number of aspects. First, under certain circumstances, it may be difficult and sometimes impossible to replace the original system classes with the instrumented without modifying the runtime system. Second, the loading order of system classes may change due to instrumentation, which often leads to runtime errors. Finally, if the code inserted by instrumentation utilizes system classes, the instrumentation may produce incorrect results or cause infinite recursion. This is because the functionality of the instrumented system classes differs from the original. To put it simply, using instrumented system classes in the inserted code is equivalent to ‘instrumenting the instrumentation’, as described below.

Consider a profiler for Java that instruments Java *bytecode* to record the sizes of objects created by the application. After each object creation statement, it inserts code that stores the size of the new object in an instance of a system class `java.util.LinkedList` by calling its `add` method. Assume that the original application also uses this list class. Therefore, during profiling, the application must use the instrumented list class to detect the objects created in its own instances of the list class. However, if the code added by the profiler uses the instrumented version as well, infinite recursion occurs. When an object is created, the profiler invokes the `add` method of the list; this method creates an object representing a new list entry. If the list class used in the instrumentation code is itself instrumented, the creation of a new entry in the `add` method will be followed by another invocation of `add`, leading to infinite recursion. Even without the recursion problem, the profiler would yield incorrect results, recording the sizes of the list entry objects created by the instrumentation code. A similar problem exists in JavaSplit runtime logic, which employs Java system classes in its implementation.

To produce correct results, the instrumentation code must use the original system classes, while all other code uses their instrumented counterparts. Therefore, the runtime must use the original and instrumented versions of the same class simultaneously. This, however, is problematic because both versions have the same name, i.e., there is a *name clash problem*.

To solve the problem we employ a novel instrumentation approach that allows both the original and instrumented versions of a system class to coexist within the same execution environment. It
prevents the name clash by renaming the instrumented classes and modifying all code, other than the instrumentation code, to use these renamed classes. The inheritance hierarchy of the instrumented classes is isomorphic to the original one. Therefore, we call our solution the Twin Class Hierarchy approach (TCH).

Although, in the context of the JavaSplit framework, TCH is merely a means to instrument system classes, it is essentially a general approach that enables sound instrumentation of system class in object-oriented languages. Hence, in the remainder of this chapter we discuss TCH independently of JavaSplit and the Java language. A stand-alone description of this method can be found in [30].

One of the most important features of TCH is its portability. Being entirely based on instrumentation, it is orthogonal to the implementation of the targeted language framework. Neither the compiler nor the runtime environment need to be modified. Since in most cases, and in JavaSplit in particular, the initial goal of code instrumentation is portability, a nonportable solution would be unacceptable.

In most languages, including Java, the TCH class name transformation does not entirely preserve the integrity of the code, because it comes into conflict with features such as inheritance, exceptions, and reflection. Therefore, the TCH approach incorporates techniques that overcome these difficulties. Despite the language-specific nature of several problematic features, most of our solutions are generic, or at least can be employed by the popular contemporary languages, e.g., C#, Smalltalk, and Java.

The implementation of TCH is not entirely automatic, but may require a certain amount of hand tuning to adapt it to a particular language. However, once the tuning is completed, TCH can be used automatically by any general instrumentation in that language.

Of course, one could argue that the problem could be solved by avoiding the use of system classes in the inserted code. However, this restriction significantly limits the convenience and power of the instrumentation. Instead of reusing system classes, the user would have to reimplement them. In addition, the functionality of those classes that cannot be reimplemented would become unavailable. For example, in the Microsoft Intermediate Language (MSIL) used in .NET, it is impossible to synchronize thread activities without using classes from the System.Threading package.

Their special status within the runtime makes system classes difficult to instrument. In fact, most frameworks, ([1, 2, 4, 98, 100] for example), avoid instrumenting them, modifying only user classes. Some systems [1, 2, 4] limit their functionality to user classes, while others [98, 100] invest
CHAPTER 3. INSTRUMENTATION OF STANDARD LIBRARIES

considerable effort into implementing context-specific workarounds.

TCH facilitates the instrumentation of system classes. With TCH class name transformation, the instrumented system classes become user classes, which are easily instrumented by most frameworks. In particular, as we show in Section 3.2, TCH allows Java system classes to be instrumented, statically or dynamically, without any modification of the Java Virtual Machine (JVM) or the operating system components. By contrast, existing frameworks that are capable of instrumenting Java system classes compromise portability. For example, Keller et al. [57] make changes in the JVM. Duncan et al. [27] modify the dynamically linked libraries (DLLs) that are used to access the file system.

Although the main motivation of TCH is to allow the original system classes to be used, our solution can be applied to user classes as well. This is useful when a reusable non-standard library, e.g., a library for management of log files, employed by the application to be instrumented, is also utilized in the code inserted by the instrumentation. We focus on system classes because it is the more difficult problem.

In order to validate the TCH approach independently of JavaSplit we employ it in an instrumentation-based profiler for Java. The profiler collects statistics about memory allocations performed by an application. We use it to explore the performance of SPECjbb benchmark [5] and of the JavaSplit runtime modules. Without the ability to instrument all system classes, the profiler would not be able to detect many of the allocations that occur within the system classes.

The main contributions of this chapter are as follows. First, we present a novel instrumentation strategy that extends the capabilities of code instrumentation in object-oriented languages. Second, we discuss issues that arise when implementing our strategy in contemporary languages. Finally, we show how TCH enables dynamic instrumentation of Java system classes.

The structure of the rest of this chapter is as follows. Section 3.1 presents the Twin Class Hierarchy approach. In Section 3.2 we discuss the difficulties of instrumenting system classes and show how TCH alleviates them. Section 3.3 explores the overhead of TCH. Section 3.4 demonstrates the contribution of TCH in profiling. Section 3.5 presents a classification of load-time instrumentation frameworks and describes their relation to TCH.
3.1 Twin Class Hierarchy

At the heart of the TCH approach lies the idea of renaming the instrumented classes to allow the instrumented and original classes to be used simultaneously by the runtime. For each original user or system class we produce an instrumented version with a different name. The inheritance relations of the instrumented classes mimic the original inheritance hierarchy. Thus, the new hierarchy is isomorphic to the original one. Figure 3.1 illustrates class renaming in Java. The new name is produced by adding a prefix ‘TCH.’ to the original one, so that SomeClass becomes TCH.SomeClass. Java system classes are designated by dark gray. The irregularity of `java.lang.Throwable` is discussed in Section 3.1.3.

In the instrumented version of a class, all code other than that inserted by the instrumentation is modified to use the TCH class names (see Figure 3.2). For example, in Java bytecode, the renaming affects the instructions `instanceof, invokevirtual, new, getfield, etc.` (See [65] for the exact semantics of the above instructions.) However, as we show in Section 3.1.7, there is no need to modify the strings that designate class names. In fact, they cannot always be distinguished from the other strings.

The TCH-related transformations are independent of the purpose-specific instrumentation. Therefore, it is possible to rewrite the code in two phases, applying the TCH transformations prior to the purpose-specific transformations (see Figure 3.3). To solve the circular dependency resulting in infinite recursion, the latter phase inserts code that uses the original class names. Both phases can be performed statically, or both can be performed dynamically. It is also possible to produce the TCH versions of classes statically, and then apply the purpose-specific transformations at run time. The TCH phase is implemented only once per language. When implemented, it can be reused with any general instrumentation process.

The class name transformation, if performed naively, compromises the integrity of the instrumented code in several contexts, including inheritance, reflection, and exception handling. In this section we study the problems that emerge and present techniques to resolve them. Together with the modification of class names, the presented solutions constitute the previously mentioned TCH transformations. In this section, the term instrumented class means class resulting from applying the TCH transformations on the original class.
Figure 3.1: A fragment of the class hierarchy in Java, before and after the TCH transformation. The instrumented versions of system classes become user classes.
### 3.1. TWIN CLASS HIERARCHY

![Java class code](image)

- **Public class A extends somepackage.C**
  - // fields
  - private int myIntField;
  - protected B myRefField;
  - public java.util.Vector myVectorField;
  - // methods
  - protected void doSomething(B b, int n) {
      if(b instanceof java.util.List) {
      //...}
  java.lang.Class vecClass =
      java.lang.Class.forName("java.util.Vector");
  ...
  }
  public B doSomethingElse(java.lang.String str) {
      java.lang.System.out.println(str);
      java.io.File f = new java.io.File(str);
  ...
  }

(a) Original class

- **Public class TCH.A extends TCH.somepackage.C**
  - // fields
  - private int myIntField;
  - protected TCH.B myRefField;
  - public TCH.java.util.Vector myVectorField;
  - // methods
  - protected void doSomething(TCH.B b, int n) {
      if(b instanceof TCH.java.util.List) {
      //...}
  TCH.java.lang.Class vecClass =
      TCH.java.lang.Class.forName("java.util.Vector");
  ...
  }
  public TCH.B doSomethingElse(TCH.java.lang.String str) {
      TCH.java.lang.System.out.println(str);
      TCH.java.io.File f = new TCH.java.io.File(str);
  ...
  }

(b) Instrumented class

Figure 3.2: A Java class before and after the TCH transformation. (Although we present source code, the actual transformation can be performed on bytecode as well.)
Figure 3.3: Instrumentation process. The code inserted during the second phase uses original class names.

Figure 3.4: High-level view of class hierarchy transformation
3.1. TWIN CLASS HIERARCHY

3.1.1 Root Class

Most modern object-oriented languages, e.g., Java, Smalltalk and C#, have an object class at the top of their class hierarchy (java.lang.Object in Java, Object in Smalltalk, System.Object in C#). Due to the name change, the instrumented object class (IOC) is not the top hierarchy class, but a subclass of the original object class (OOC).

Figure 3.4 presents a high-level view of the class hierarchy transformation. In the new class hierarchy, the IOC is a direct subclass of the OOC. Since the IOC is not the root class, the language may have certain special classes that are not its subclasses. For example, in C#, the TCH transformation cannot replace the superclass of an array class, because arrays are implicitly defined, e.g., using the [ ] operator at the source code level. As a result, despite the TCH transformation, arrays remain subclasses of System.Object and are not subclasses of TCH.System.Object. Classes that do not subclass the IOC are denoted irregular classes. These classes present a few problems that are discussed and resolved in Section 3.1.2.

3.1.2 Irregular Classes

The presence of irregular classes is problematic mainly in languages with static typing, e.g., Java, C# and O’Caml. Languages with dynamic typing (such as Smalltalk) may be affected by it, depending on whether the purpose-specific instrumentation modifies OOC behavior.

In statically typed languages, TCH class renaming would transform all uses of OOC in the original code, e.g., local variables, class fields, and method parameters, into uses of IOC. Thus, a variable of type OOC that originally could have referenced an irregular class, e.g., an array class, is transformed to a variable that is unable to do so, because irregular classes do not subclass IOC. For example, in instrumented C# code, a local variable of type TCH.System.Object cannot contain a reference to an array of integers whose direct superclass remains System.Object, despite the class name transformation. To solve this problem, OOC references are not replaced by IOC references in the instrumented code. Rather, they are left as they are, in order to allow them to reference the irregular classes. The only exception to this rule is made in the code that originally creates an OOC instance: after instrumentation it creates an IOC instance.

Another problem with irregular classes is that they are not affected by the changes made by
a purpose-specific instrumentation in the implementation of IOC. For example, if a field myID_ and a method id() returning this field are added to IOC, only the subclasses of IOC will have an ID. This problem, which is relevant also in languages with dynamic typing, may be resolved by applying the transformations applied to the IOC directly to the implementation of irregular classes. In the above example, all irregular classes should be augmented with the ID field and ID method. Unfortunately, this solution is inapplicable to classes that do not have a class definition, but rather are defined implicitly, e.g., array classes. In these special cases, the problem is solved by defining a wrapper class around the implicitly defined classes. This wrapper should subclass the IOC, thus inheriting all its functionality. All references to the original class should be replaced with references to the wrapper. Figure 3.5 illustrates a wrapper for a one-dimensional Java integer array. In all the instrumented bytecode the invocations of the bytecode instructions newarray, iaload and iastore should be replaced by invocations of the wrapper constructor, and methods load and store respectively.

In statically typed languages, a variable of type OOC that points to a subclass of IOC needs to be downcast to IOC whenever accessing a method or a (public) field added to IOC by the purpose-specific instrumentation. If the object referenced by a variable of type OOC is an instance

Figure 3.5: A possible wrapper for a Java integer array. Accessor methods are final to enable inlining.
of an irregular class, then it needs to be downcast to its own class, before accessing the additional member. (If the irregular class was not augmented with these members, the code accessing them must be skipped.)

3.1.3 Classes with Special Semantics

Some languages attribute special semantics to certain classes (or other class-like constructs, e.g., Java interfaces). For instance, in Java, only a subclass of java.lang.Throwable can be thrown as an exception. Only a class implementing the java.io.Serializable interface can be marshaled into a bitstream. The renaming causes the instrumented classes to lose their special semantics, often violating the integrity of the instrumented code. For example, in the instrumented code the argument of a throw statement in Java is no longer a subclass of java.lang.Throwable, but rather a subclass of TCH.java.lang.Throwables, which is illegal.

Since the special semantics are class-specific, so in theory should be the solution. In practice, a general technique presented below solves the problem for those cases of special class semantics of which we are aware. (Note that very few system classes have unusual semantics.)

In order to regain the special semantics of an instrumented class, we make it a direct subclass of its original version. This solves the problem, because it creates an ‘is a’ relationship between the instrumented class and its original counterpart. For instance, in the case of Java’s throwable class, the instrumented class TCH.java.lang.Throwables becomes a subclass of java.lang.Throwable (Figure 3.1(b)) and therefore can be used in a throw statement. This solution also works for all of Java’s special interfaces. When this technique is applied, additional irregular classes are created.

The above solution would not be possible unless the original special classes could be subclassed. For example, it would not work if Java’s throwable class were a final class. (In Java, final classes cannot be subclassed.) It would also fail if the original class was not a direct subclass of the OOC. Fortunately, those classes with special semantics can be subclassed and directly subclass the OOC.
CHAPTER 3. INSTRUMENTATION OF STANDARD LIBRARIES

3.1.4 Invalid Method Overriding

In most languages with static typing, an inherited method cannot be overridden by a method with a different return type and the same argument types. In previous sections we saw cases of instrumented classes subclassing their original counterparts. In section 3.1.1, we describe how an IOC is made a direct subclass of an OOC. In section 3.1.3 we show how instrumented classes regain their special semantics by subclassing their original versions. One consequence of the TCH class name transformation is that an instrumented class may contain a method that differs from the original method only by a return type. (Method arguments remain the same if a method does not have object reference arguments.) Therefore, subclassing a non-TCH class by its instrumented version can lead to a violation of method overriding rules. For example, in Java, the return type of the method toString() in IOC is TCH.java.lang.String, which is different from its return type in OOC.

We solve this problem by renaming the problematic methods, and modifying the code to use the new names, which are produced by adding a prefix, e.g. 'TCH__', to the original names. For example, when creating the IOC, its toString() method is renamed TCH__toString().

3.1.5 Object Constants

Most languages contain object constants. For instance, in Java, string literals are objects of type java.lang.String. In pure object-oriented languages like Smalltalk, all constants and strings, e.g., 1977, 0.333 and 'hi', are objects.

Since object constants are instances of the original system classes, they are not compatible with the rest of the code, which is instrumented to use the TCH classes. We solve this problem by replacing each object constant of a class A with an instance of TCH.A. For example, in Java, each string literal is replaced with a corresponding instance of TCH.java.lang.String. (A regular string may be converted into a TCH string by using the underlying character array of the former to create an instance of the latter.) The above problem exists both in statically and dynamically typed languages. In the latter, if object constants are not converted to the TCH form, they will not be affected by the instrumentation.
3.1. TWIN CLASS HIERARCHY

3.1.6 Runtime Exceptions

Runtime environments, e.g., JVM or .NET Common Language Runtime (CLR), may throw runtime exceptions. These exceptions are not TCH objects.

In statically typed languages the exceptions are caught on the basis of their class. The TCH class name transformation causes the rewritten catch statements to catch the TCH exceptions but miss the runtime exceptions. To solve this problem we modify each catch statement to catch both the instrumented and the original versions of the exception classes associated with it. In both dynamically and statically typed languages, a caught runtime exception is converted, before the execution of an exception handler, to an instance of the corresponding TCH class. This ensures that the exception handler will process a TCH version of the exception.

3.1.7 Reflection

The TCH approach modifies class names and method names (Section 3.1.4). Consequently, it must be adapted to preserve the behavior of the reflection mechanisms of a language. Since reflection may be implemented differently in each language, there is no generic solution for this problem. However, the following simple technique allows the support of the basic features of reflection in most languages.

In the instrumented code, when invoking reflection related methods, we perform translation between the original and the instrumented class (or method) names. Each input parameter of a reflection method designating the name of a class (or a method) is converted to the TCH form (by addition of the TCH prefix). Each output (usually the return type) designating the name of a class (or a method) is converted to the non-TCH form. (Note that if a name is already in the desired form, we leave it as is.) The translation is performed inside the reflection-related methods.

For example, consider the methods forName and getName of the Java class java.lang.Class. The former method returns an instance of java.lang.Class that corresponds to its single string parameter. The latter method, which does not have any parameters, returns the name of a class represented by the given instance of java.lang.Class. In the instrumented code, the class name parameter of forName, 'SomeClass' is converted to 'TCH.SomeClass'. Therefore, the returned instrumented class object represents the instrumented class rather than the original class. Thus, if we use the returned class to create an instance, we will create an instance of the
instrumented class and not the original. Similarly, the instrumented string returned by `getName`, `TCH.SomeClass` is translated to `SomeClass`. Consequently, if the returned string is compared to a hardcoded class name string, the result of the comparison will be correct because the hardcoded names remain in non-TCH form, despite the TCH transformations.

### 3.1.8 Native Classes

Contemporary languages have system classes whose implementation is integrated into the runtime. Specifically, the implementation of a subset of their methods is hardcoded. Such methods are called `native` in Java and `internalcalls` in C#. Henceforth, we conform to Java terminology and call them `native methods`. System classes that contain native methods will be referred to as `native classes`.

Implementation of a native method is always bound to a particular method in a particular class. It can be accessed only by calling that method, but cannot be reused in the implementation of another method (or in another class). For example, consider the native method `currentTimeMillis()` defined in the Java class `java.lang.System`. Its implementation, which queries the underlying OS for the current time, cannot be incorporated into any other class. Therefore, after TCH class
renaming, the native functionality is unavailable in the instrumented system classes. To correct this, TCH must provide the instrumented system classes with an alternative implementation that simulates the original API. We accomplish this by using the original version of a class in the implementation of its instrumented counterpart. This is possible only because the TCH approach allows the original and the instrumented versions of a class to coexist.

In most cases, an instrumented version of a native class is implemented as a wrapper around the original class. The wrapper methods delegate invocations of the native API methods to an encapsulated instance of the original class. Figure 3.6 illustrates the implementation of the instrumented version of a hypothetical class File that originally had native API methods to access the file it represents. Figure 3.7 shows additional examples of the delegation approach. If necessary, the wrapper methods convert the parameters and the return type from TCH form to the original form and vice-versa. While non-static methods are implemented using an instance of an original class, static methods do not require such an instance.

The delegation pattern is highly suitable when the native methods are used to access operating system resources, such as the clock, the file system, or the network. In general, delegation is applicable to those native classes that have a distinct purpose and whose operation does not affect instances of other classes. Note that the above delegation approach works well in presence of protected and private native methods. The wrapper class defines only the API methods, which are public. Therefore, it needs to invoke only the public methods of the encapsulated (original) class.

Implementation of TCH versions of some native classes requires techniques that are similar to but somewhat more complex than delegation. We identify two main categories of these classes: (i) classes that are structurally incompatible with the delegation approach and (ii) classes with native methods that invoke or implement language-specific mechanisms. (The latter can be perceived as classes semantically incompatible with delegation).

The Java class java.lang.Class belongs to the first category. It is structurally incompatible with delegation because it has a package-private method getPrimitiveClass(String). This method returns the class that represents the primitive type indicated by its parameter. Package-private methods can be invoked only by a class from the package of the target class. Since the instrumented classes are defined outside the original package, they cannot delegate calls to package-private methods of the original class. The solution is context-specific: upon invocation of the
CHAPTER 3. INSTRUMENTATION OF STANDARD LIBRARIES

// Returns the current time in milliseconds.
public static long currentTimeMillis()
{
    return java.lang.System.currentTimeMillis();
}

(a) JS.java.lang.System.currentTimeMillis

// Gets the name of the local host
public TCH.java.lang.String getLocalHostName()
{
    // origImpl_ is a private field of type java.net.InetAddress
    java.lang.String name = origImpl_.getLocalHostName();
    // convert the name into a TCH.java.lang.String and return it
    return TCH.java.lang.String.__JS__convertFromJavaString(name);
}

(b) JS.java.net.InetAddress.getLocalHostName

// Calculates the square root of its parameter
public static strictfp double sqrt(double arg)
{
    return java.lang.StrictMath.sqrt(arg);
}

(c) JS.java.lang.StrictMath.sqrt

// Determines if the specified Class object represents a primitive type
public boolean isPrimitive()
{
    // origImpl_ is a private field of type java.lang.Class
    return origImpl_.isPrimitive();
}

(d) JS.java.lang.Class.isPrimitive

Figure 3.7: Implementation of several originally native methods
3.1. TWIN CLASS HIERARCHY

wrapper method, the requested class is obtained from the public field TYPE of the class representing the boxed primitive type. For instance, the class representing a long is read from the field TYPE of java.lang.Long. A more general but less secure solution is to modify the permission of the original method.

The class java.lang.String belongs to the second category. The semantics of its native method String intern() do not allow simple delegation. This method must return the canonical representation of a given string. In Java, for two equal strings $s$ and $t$, $s$.intern() and $t$.intern() must always return the same string object whose value is equal to that of $s$ and $t$. If implemented by simple delegation, the intern method of the wrapper class would wrap the canonical string returned by the original method into a new TCH string. Therefore, even two consecutive calls to the intern method on the same TCH string would not return the same object. To support the semantics of the intern method, the TCH string class should make sure that each canonical string is always wrapped into the same TCH string. This is achieved by maintaining a mapping between canonical strings and the TCH strings that are used to wrap them. This however, is a deviation from the delegation pattern.

Normally, only a small portion of the system classes are native. Most of these are related to reflection, GUI, I/O, and networking. In Java they constitute about 3% of the system classes. A much smaller portion is required to run most programs that do not contain a GUI. We have successfully executed various applications, including SPECjbb, and applications that perform I/O and networking.

3.1.9 Applicability Discussion

Some techniques used to alleviate the side effects of the TCH approach are language specific. Therefore, in theory there can be a language to which TCH would be inapplicable.

TCH is a general methodology and not an algorithm. Thus, the techniques to alleviate its side effects should be perceived as guidelines rather than specific instructions. If a certain issue cannot be resolved by the proposed techniques, then language-specific solutions should be sought. The strength of TCH is in the fact that once all issues are resolved for a particular language, it can be automatically employed by any general purpose instrumentation in that language.
3.2 Facilitating Instrumentation of System Classes

Code instrumentation can be performed statically, before the execution begins, or dynamically at run time. The special status of system classes makes their transformation problematic in both modes. TCH eliminates most of the difficulties, because, after renaming, the instrumented versions of system classes are no longer part of the standard class libraries but are rather user classes, which are much easier to instrument. In contrast to the instrumentation frameworks that allow arbitrary transformations of all system classes, TCH does not require modification of any components of the language infrastructure, e.g., the compiler or the runtime.

In this section, due to a variety of language-specific mechanisms and issues, we focus our discussion on Java. However, most of it is valid for other frameworks with dynamic loading, including the .NET platform.

3.2.1 Static Instrumentation

If the instrumentation does not modify the class names, static instrumentation of system classes must force the runtime to use the instrumented versions instead of the originals. This may not always be possible, because the system classes may be deeply integrated into the runtime.

In Java, most popular JVMs, e.g., Sun and IBM JDKs, provide a command line option that allows the user to specify the path from which system classes should be loaded. In the JDKs mentioned above, it is also possible to change the implementation of system classes by modifying the contents of the file `rt.jar` in which most of the system classes are stored. Unfortunately, neither option is standardized. Therefore, in theory, there may be a valid JVM that does not allow static instrumentation of system classes.

With TCH class name transformation, the instrumented system classes become user classes. Therefore, they are loaded as ordinary user classes, not instead of, but in addition to their original versions, rendering the need to replace them or update their loading path obsolete. Thus, TCH avoids potential portability problems.

Runtime environments can make assumptions regarding the structure and loading order of system classes. If these assumptions do not hold, a runtime may terminate abnormally, often without a comprehensive error message. For example, most JVM implementations make assumptions about
3.2. FACILITATING INSTRUMENTATION OF SYSTEM CLASSES

the size of the classes java.lang.Object, java.lang.Class and java.lang.String. If the instrumentation process augments one of these classes with a field, the JVM crashes. It will also crash if the loading order is changed as a result of instrumenting the above classes. Note that these problems arise not only in static but also in dynamic instrumentation. TCH lets the JVM keep the original definitions and the loading order of the system classes, thus avoiding these difficulties.

3.2.2 Dynamic Instrumentation

The main advantage of dynamic instrumentation is that it does not require a priori knowledge of the classes used by a program (closed world assumption). Since reflection allows loading classes whose identity is determined at run time, it may be impossible to determine the transitive closure of classes used by a program. Moreover, classes created at run time can only be instrumented dynamically.

The most important challenge in dynamic instrumentation is to intercept all the classes employed by an application. In most runtimes, it is difficult to intercept system classes. In Java, there are two main obstacles. Both are related to the Java class loading mechanism, which is used by most contemporary frameworks to implement dynamic instrumentation. First, a subset of system classes (approximately 200 in Sun JDK) is already loaded by the JVM, before the class loading mechanism can be modified to enable rewriting. Since most of these preloaded classes are used extensively by non-trivial applications, it is important that they can be instrumented. (Among the preloaded classes are: java.lang.Integer, java.lang.String, java.util.HashMap, java.lang.Thread.) Second, the class loading mechanism attempts to ensure that the system classes are defined by the bootstrap class loader. Since the bootstrap class loader is integrated into the JVM, the user cannot gain any control over it without modifying the JVM, which is highly undesirable. Consequently, it is hard to modify the definition of a system class.

Portable load-time instrumentation frameworks, such as JMangler [58], Javassist [18], JOIE [24], BCEL [26], AspectWerkz [2], and JBoss AOP [4], do not instrument system classes (due to the problems mentioned above). The frameworks that are capable of instrumenting system classes, e.g., [57] and [27], compromise portability by modifying the JVM or the underlying DLLs. Personal communication with representatives of JMangler, AspectWerkz, and JBoss AOP has revealed their desire to perform dynamic transformation of system classes as well as their inability to accomplish this.
The TCH approach supports dynamic instrumentation of Java system classes. TCH does not suffer from the problems mentioned above because it renames system classes, thus transforming them into user classes. Since most runtimes (with dynamic loading) allow dynamic instrumentation of user classes, TCH effectively allows all system classes to be instrumented without any modification of the runtime infrastructure. To the best of our knowledge, TCH is the only technique that achieves this in Java.

TCH-based Dynamic Instrumentation in Java

In Java and similar frameworks, such as .NET Common Language Runtime, TCH supports dynamic instrumentation in the following way. Let AppMain be the class that contains the main method of the application to be executed. At the beginning of the execution, we convert the string parameters of the main method into TCH form. Then, we hook into the class loading system either by installing a custom class loader, as is done in Javassist, or by replacing the definition of the system class loader (java.lang.ClassLoader), as is done in JMangler. After that, we instruct the adapted class loading system to load TCH.AppMain, and then employ reflection to execute its main method.

When asked to load a class whose name begins with ‘TCH.’, e.g., TCH.somePackage.SomeClass, the adapted class loading mechanism fetches the definition of the corresponding original class (somePackage.SomeClass) from the loading path, and then sequentially applies to it the TCH and purpose-specific transformations, as described in the beginning of Section 3.1. When asked to load a class whose name does not begin with ‘TCH.’, the adapted class loader loads the class without applying any transformations to it. The above procedure is summarized in Figure 3.8.

The entire twin hierarchy, including the system classes, is produced on the fly. The implementation of TCH versions of native system classes described in Section 3.1.8 is hardcoded into the
3.3. TCH OVERHEAD ANALYSIS

Transformer and thus can also be generated dynamically. Alternatively, the TCH versions of system classes can be produced statically, while the TCH versions of user classes are produced dynamically.

3.3 TCH Overhead Analysis

We estimate the overhead of TCH using sequential applications from the Java Grande Forum (JGF) Benchmark Suite (version 2.0) [15] and the SPECjbb benchmark [5]. We compare the throughput of the original programs with their TCH counterparts, which are produced statically. The measurements were performed on Intel’s dual-processor machine, 2x1.7 GHz with 1 GB memory, using Sun JDK 1.4.2.

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.

With few exceptions, the comparison shows that the performance of the rewritten bytecodes is close to their original performance. Tables 3.1 and 3.2 summarize the results. The former presents the throughput of microbenchmarks from the first section of the JGF benchmark suite. The latter presents the results of SPECjbb and of the remaining applications from the JGF benchmark suite. The “Difference” column shows the difference between the original and instrumented benchmarks. Let $A$ and $B$ be the throughputs of the original benchmark and its TCH version respectively. The corresponding value in the “Difference” column is $100 \times (A-B)/A$. Consequently, a positive value indicates higher throughput of the original application.

The most significant performance difference in Table 6.1 is observed for object creation benchmarks (denoted by the prefix “Create:Object”), and exception benchmarks (denoted by the prefix “Exception”). In both cases, the difference is caused by the increased cost of creating rewritten objects. This increased cost is due to the fact that these objects’ inheritance chain is augmented with an additional class at the top of the hierarchy (java.lang.Object or java.lang.Throwable; see Figure 3.1). As a result, an additional constructor needs to be called during their creation.
Table 3.1: JGF Benchmark Suite – microbenchmark results. Due to space limitations we present only a subset of benchmarks. The throughput difference of the omitted benchmarks is insignificant.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>TCH</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith: Add: Int</td>
<td>864733630</td>
<td>852847490</td>
<td>1.37</td>
</tr>
<tr>
<td>Arith: Add: Double</td>
<td>1485906.2</td>
<td>1481910.2</td>
<td>0.27</td>
</tr>
<tr>
<td>Arith: Mul: Int</td>
<td>116097232</td>
<td>116747128</td>
<td>0.21</td>
</tr>
<tr>
<td>Arith: Mul: Long</td>
<td>59741112</td>
<td>60856160</td>
<td>-1.87</td>
</tr>
<tr>
<td>Arith: Mul: Double</td>
<td>1464426.1</td>
<td>1457029</td>
<td>0.51</td>
</tr>
<tr>
<td>Assign: Same: Scalar: Local</td>
<td>2362722050</td>
<td>2344758530</td>
<td>0.76</td>
</tr>
<tr>
<td>Assign: Same: Scalar: Instance</td>
<td>911725950</td>
<td>9173885100</td>
<td>-0.62</td>
</tr>
<tr>
<td>Assign: Same: Scalar: Class</td>
<td>737395200</td>
<td>736152770</td>
<td>0.17</td>
</tr>
<tr>
<td>Assign: Other: Scalar: Instance</td>
<td>295890272</td>
<td>296040640</td>
<td>-0.05</td>
</tr>
<tr>
<td>Assign: Other: Scalar: Class</td>
<td>268851840</td>
<td>267262064</td>
<td>0.59</td>
</tr>
<tr>
<td>Cast: Int: Float</td>
<td>50712684</td>
<td>50683232</td>
<td>0.15</td>
</tr>
<tr>
<td>Cast: Int: Double</td>
<td>50532808</td>
<td>50579816</td>
<td>-0.45</td>
</tr>
<tr>
<td>Create: Array: Int: 16</td>
<td>7181555</td>
<td>6800600.15</td>
<td>4.34</td>
</tr>
<tr>
<td>Create: Array: Int: 32</td>
<td>4960039</td>
<td>4688108</td>
<td>5.48</td>
</tr>
<tr>
<td>Create: Array: Long: 2</td>
<td>9238229</td>
<td>9014805</td>
<td>2.43</td>
</tr>
<tr>
<td>Create: Array: Long: 64</td>
<td>1286189.8</td>
<td>1214997.6</td>
<td>5.54</td>
</tr>
<tr>
<td>Create: Array: Object: 1</td>
<td>9900894</td>
<td>93503014</td>
<td>3.75</td>
</tr>
<tr>
<td>Create: Array: Object: 4</td>
<td>9270639</td>
<td>9013093</td>
<td>2.78</td>
</tr>
<tr>
<td>Create: Array: Object: 8</td>
<td>8566349</td>
<td>8227791</td>
<td>3.95</td>
</tr>
<tr>
<td>Create: Object: Simple</td>
<td>2048000</td>
<td>1788802.5</td>
<td>12.66</td>
</tr>
<tr>
<td>Create: Object: Simple: Constructor</td>
<td>2032149.2</td>
<td>1788802.5</td>
<td>11.97</td>
</tr>
<tr>
<td>Create: Object: Simple: 4Field</td>
<td>1927529</td>
<td>1676527.1</td>
<td>13.02</td>
</tr>
<tr>
<td>Create: Object: Subclass</td>
<td>2038216.5</td>
<td>1764986.6</td>
<td>13.41</td>
</tr>
<tr>
<td>Create: Object: Complex</td>
<td>1678138.4</td>
<td>1481374.4</td>
<td>11.73</td>
</tr>
<tr>
<td>Exception: New</td>
<td>196409.3</td>
<td>141303.75</td>
<td>28.07</td>
</tr>
<tr>
<td>Exception: Method</td>
<td>185749.5</td>
<td>135227.94</td>
<td>27.20</td>
</tr>
<tr>
<td>Loop: For</td>
<td>511600320</td>
<td>503679008</td>
<td>1.16</td>
</tr>
<tr>
<td>Loop: ReverseFor</td>
<td>511201248</td>
<td>482392032</td>
<td>5.60</td>
</tr>
<tr>
<td>Math: Abs: Long</td>
<td>37215220</td>
<td>37261768</td>
<td>-0.13</td>
</tr>
<tr>
<td>Math: Abs: Double</td>
<td>31678268</td>
<td>31813592</td>
<td>-0.43</td>
</tr>
<tr>
<td>Math: Max: Float</td>
<td>28885754</td>
<td>28957228</td>
<td>-0.25</td>
</tr>
<tr>
<td>Math: Max: Double</td>
<td>29534024</td>
<td>28959788</td>
<td>1.94</td>
</tr>
<tr>
<td>Math: Min: Long</td>
<td>33553144</td>
<td>31863088</td>
<td>5.04</td>
</tr>
<tr>
<td>Math: Min: Double</td>
<td>20999084</td>
<td>28429638</td>
<td>5.23</td>
</tr>
<tr>
<td>Math: Sin: Double</td>
<td>8395163</td>
<td>8399036</td>
<td>-0.05</td>
</tr>
<tr>
<td>Math: Atan: Double</td>
<td>4082094.5</td>
<td>4050867.2</td>
<td>0.56</td>
</tr>
<tr>
<td>Math: Floor: Double</td>
<td>4026740</td>
<td>3768169.2</td>
<td>6.42</td>
</tr>
<tr>
<td>Math: Pow: Double</td>
<td>750348.06</td>
<td>788845.25</td>
<td>-5.13</td>
</tr>
<tr>
<td>Math: Rint: Double</td>
<td>4026344.2</td>
<td>3810941.5</td>
<td>5.35</td>
</tr>
<tr>
<td>Math: Round: Float</td>
<td>2788671</td>
<td>2745308.2</td>
<td>1.55</td>
</tr>
<tr>
<td>Math: IEEE: Remainder: Double</td>
<td>422791.1</td>
<td>422895.84</td>
<td>-0.02</td>
</tr>
<tr>
<td>Method: Same: Instance</td>
<td>159090944</td>
<td>158750580</td>
<td>0.76</td>
</tr>
<tr>
<td>Method: Same: Synchronized: Instance</td>
<td>5423729</td>
<td>5422652</td>
<td>0.02</td>
</tr>
<tr>
<td>Method: Same: Class</td>
<td>175755072</td>
<td>175511520</td>
<td>0.13</td>
</tr>
<tr>
<td>Method: Other: Instance</td>
<td>31480450</td>
<td>31538018</td>
<td>-0.18</td>
</tr>
<tr>
<td>Method: Other: Instance: Of: Abstract</td>
<td>31528914</td>
<td>31538018</td>
<td>-0.03</td>
</tr>
</tbody>
</table>

Average of all JGF microbenchmarks: 3.01
Table 3.2: Benchmark application results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>TCH</th>
<th>Difference (%)</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECJbb</td>
<td>6727</td>
<td>6524</td>
<td>3.02</td>
<td>(Operations/s)</td>
</tr>
<tr>
<td>Series</td>
<td>488.05</td>
<td>533.15</td>
<td>-9.24</td>
<td>(Coefficients/s)</td>
</tr>
<tr>
<td>LUFact</td>
<td>192.70</td>
<td>189.64</td>
<td>1.59</td>
<td>(Mflops/s)</td>
</tr>
<tr>
<td>HeapSort</td>
<td>634678.90</td>
<td>632111.25</td>
<td>0.40</td>
<td>(Items/s)</td>
</tr>
<tr>
<td>Crypt</td>
<td>2235.01</td>
<td>2239.02</td>
<td>-0.18</td>
<td>(Kbyte/s)</td>
</tr>
<tr>
<td>FFT</td>
<td>129814.42</td>
<td>139070.45</td>
<td>-6.36</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>SOR</td>
<td>16.84</td>
<td>16.14</td>
<td>4.15</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>SparseMatmult</td>
<td>12.72</td>
<td>12.76</td>
<td>-0.42</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>Euler</td>
<td>4.44</td>
<td>4.67</td>
<td>-5.18</td>
<td>(Timesteps/s)</td>
</tr>
<tr>
<td>MolDyn</td>
<td>181487.31</td>
<td>187443.61</td>
<td>-3.28</td>
<td>(Interactions/s)</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>406.16</td>
<td>277.48</td>
<td>31.68</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>RayTracer</td>
<td>1183.73</td>
<td>1249.95</td>
<td>-5.59</td>
<td>(Pixels/s)</td>
</tr>
<tr>
<td>AlphaBetaSearch</td>
<td>798061.56</td>
<td>798356.90</td>
<td>-0.04</td>
<td>(Positions/s)</td>
</tr>
</tbody>
</table>

The instrumentation overhead in SPECJbb (Table 3.2) is only 3%. The most significant throughput difference among the macrobenchmarks is observed in the Monte Carlo benchmark, whose throughput decreases by 32% as a result of the instrumentation. The Monte Carlo benchmark extensively uses the class `java.util.Random`, whose methods are often inlined by the just-in-time compiler (JIT). However, the JIT does not inline the counterparts of these methods in the instrumented code, which causes a decrease in performance.

### 3.4 Application of TCH in profiling

To demonstrate the capabilities of TCH, outside the context of JavaSplit, we have implemented a Java memory profiler that employs TCH to instrument system classes. The profiler gathers memory allocation statistics. It can be used to explore memory usage and detect memory leaks in any Java program. The bytecode instrumentation is performed dynamically, using the Bytecode Engineering Library (BCEL) [26] and its custom class loader.

The profiling transformation intercepts all bytecode instructions used in object and array creation (i.e., `new`, `newarray`, `anewarray`, and `multinewarray`). It also intercepts the system API calls that create new object instances, i.e., `java.lang.Object.clone()` and `java.lang.reflect.Constructor.newInstance(Object[])`. In the bytecode, after each such allocation
CHAPTER 3. INSTRUMENTATION OF STANDARD LIBRARIES

Table 3.3: Profiler output – object and array allocations in SPECjbb

<table>
<thead>
<tr>
<th>Class</th>
<th>User classes</th>
<th>System classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count</td>
<td>%</td>
</tr>
<tr>
<td>char[]</td>
<td>567268</td>
<td>22.27</td>
</tr>
<tr>
<td>TCH.java.lang.String</td>
<td>8000041</td>
<td>85.35</td>
</tr>
<tr>
<td>TCH.java.util.Hashtable$Entry[]</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>TCH.java.util.Hashtable$Entry</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>int[]</td>
<td>76235</td>
<td>99.65</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>700428</td>
<td>99.98</td>
</tr>
</tbody>
</table>

event, the profiler transformer inserts a call to a special handler. This handler may record any interesting data associated with the event, e.g., the class of the created object, its size, the time of its creation, etc. (The newly created object is passed as a parameter to the handler.) The profiling transformation augments each class with a method that returns its instance size.

The profiler handler accesses the internal profiler logic, which is implemented in pure Java. The implementation of the profiler logic extensively uses system classes, e.g., java.lang.System, java.util.Hashtable, java.util.LinkedList, java.util.Iterator, java.io.PrintStream, and java.io.FileOutputStream. For example, an instance of class java.util.Hashtable is used for mapping between a class name and a counter of allocated class instances. An instance of java.util.LinkedList is used to record the times of allocation events. Instances of class java.io.FileOutputStream are used to spool the collected data to files. (The files are used during the execution because the accumulated data may be too large, especially in long-running applications.)

TCH benefits the profiler in two ways. First, it allows the profiler to explore applications that use the same system classes that it uses in its implementation. While the application uses the instrumented system classes, the profiler logic employs their original counterparts. Thus, the original functionality of system classes remains available to the profiler. Without TCH, the profiler would have to use the instrumented system classes, which would result in infinite recursion. Second, TCH enables the profiler to instrument all system classes (dynamically). Consequently, it can collect more accurate results. If the profiler instrumented only user classes, then allocations performed within the system classes would not be detected. If system classes are not modified, then allocations...
of arrays and system class object instances that are performed within the code of system classes are impossible to intercept. Allocations of user class instances within system classes can still be intercepted by modifying the constructors of the former.

Table 3.3 illustrates the importance of the second feature in profiling of the SPECjbb benchmark. The table presents the final values of creation counters of several classes used by the benchmark. The columns “User classes” and “System classes” indicate the number of instances created in user classes and system classes respectively. The table shows that a large number of allocations occur within the code of the system classes. For example, most character arrays (char[]) are allocated within the system classes. Moreover, the creation of the system classes java.util.Hashtable$Entry[] and java.util.Hashtable$Entry occurs only within the system code. Without the ability to instrument system classes, all these allocations would remain undetected.

The Cougar Memory Profiler (CMP) [3] is a bytecode instrumentation-based tool for memory profiling of Java programs. The developer selects which classes should be tracked and runs an automated class file editor (using BCEL) to add profiling instructions to the constructors. The profiler maintains pointers to the live instances and can display various useful debugging information, e.g., the total number of allocations of a profiled class, including the number of live and garbage-collected instances. In contrast to our memory profiler, the instrumentation is performed statically, i.e., before the execution.

Until recently, CMP did not have any support for profiling Java system classes. Currently, it allows the user to (statically) transform the classes in system packages. The instrumented classes are loaded into the JVM by means of the command line option -Xbootclasspath, which allows an alternative location of the system classes to be specified when starting the JVM.

The CMP manual advises users to avoid modifying system classes as much as possible in order to prevent the potential loading errors that occur if the profiler’s code uses system classes that are being analyzed. The manual states that “... a call to `new HashSet()` will fail if HashSet is profiled, due to a stack overflow caused by the circular reference. Similarly, if all of java.lang will be profiled, then Strings should be carefully handled to avoid string allocations, including any calls to System.out.” By contrast, TCH allows our memory profiler to easily analyze any system class, even when this system class is used in the profiler’s logic.
3.5 Load-time Instrumentation Frameworks

In this section we overview the existing frameworks for dynamic instrumentation of Java bytecode, focusing on their ability to intercept system classes. Figure 3.9 illustrates the various approaches to class interception. Our main observation is that, without the use of TCH, only nonportable techniques allow instrumenting system classes at run time.

3.5.1 Custom Class Loaders

The Java class loading mechanism allows users to install custom class loaders to intercept class files at load-time. A custom class loader must subclass the system class java.lang.ClassLoader. This strategy is employed by Javassist [18], JOIE [24], and BCEL [26]. The applicability of this approach is limited to applications that do not use their own custom class loaders. This is because only one class loader can affect the definition of a class when it is being loaded. Without TCH, a custom class loader cannot intercept all system classes because of the difficulties described in section 3.2.2, such as the preloaded classes problem.
3.5. LOAD-TIME INSTRUMENTATION FRAMEWORKS

3.5.2 JVM Dependent Interception

The behavior of the class loading mechanisms can be affected by modifying the implementation of the JVM. Binary Component Adaptation (BCA) [57] introduces an adaptation module for transforming the internal JVM data structure that represents a loaded class. Unlike most instrumentation frameworks, BCA allows the system classes to be redefined. Unfortunately BCA requires a custom JVM, thus compromising portability.

3.5.3 Library-based Interception

Duncan and Hölzle [27] introduce library-based load time adaptation. They intercept and modify the class files as they are being fetched from the file system. This is achieved by modifying a dynamically-linked standard library that is responsible for reading files. Like BCA, this approach allows instrumentation of system classes at the expense of portability. It requires that a custom DLL be provided for every operating system.

3.5.4 Class Loader Independent Interception

J Mangler [58] provides a portable interception facility, which, unlike Javaassist, BCEI, and JOIE, allows the application to use custom class loaders. This is achieved by providing a modified version of the final method defineClass() in the class java.lang.ClassLoader. Because the modified behavior is enforced for every subclass of java.lang.ClassLoader, JMangler is activated whenever an application-specific class is loaded. In contrast to BCA and DLL-based load-time adaptation, this approach is limited because it cannot transform system classes without employing TCH.
Chapter 4

Transparent Distributed Runtime for Java

In a stable and controlled environments there might be no need in fault-tolerance. To enable a more efficient execution under these circumstances, JavaSplit allows the user to disable fault-tolerance support and thus avoid the unnecessary instrumentation and checkpointing overheads. The mechanisms related to distribution, namely the distributed memory management and distributed locking logic, are designed incrementally with respect to fault-tolerance; both modules have non-fault-tolerant stand-alone base protocols. When resilience to failures is disabled, these base protocols are employed instead of their fault-tolerant extensions. In this setting, the checkpointing instrumentation is deactivated as well. Consequently, when disabled, the support for fault-tolerance does not incur any additional overhead.

This chapter presents the JavaSplit components which are not related to resilience to node failures. It describes bytecode instrumentation that enables distributed execution of a multithreaded program written for a single JVM. It also describes the base distributed memory protocol and the distributed locking protocols mentioned above. Chapter 5 complements this description of JavaSplit by presenting the checkpointing instrumentation and the fault-tolerance extensions of the above protocols.
4.1 Distributed Memory Management Protocol

Any shared memory, distributed or not, is a subject to a set of constraints, which constitute the memory model. The memory model defines rules that concern propagation of data modifications from one execution context to another. The Java Memory Model (JMM) describes how threads in the Java programming language interact through memory. Together with the description of single-threaded execution of code, the memory model provides the semantics of the Java programming language.

The new JMM [53] is based on the principles of the Lazy Release Consistency (LRC) [55] memory model. The JavaSplit distributed consistency protocol complies with LRC and incorporates the necessary adjustments to be fully compatible with the JMM. For instance, since there is a natural mapping for Java volatile variables to the release-acquire semantics of LRC in the revised JMM, we encapsulate accesses to volatile variables with acquire-release blocks.

The basic memory coherency unit in JavaSplit is a Java object. This approach fits well on top of the JVM object-based memory management. Employment of the page-based approach used in traditional distributed shared memory systems would be quite unnatural in the current context, since the hardware paging support for detection of memory accesses cannot be attained without modifying the JVM. In addition, allocation of multiple objects on the same page would result in false sharing.

The JavaSplit distributed memory management protocol is inspired by the state-of-the-art Home-Based Lazy Release Consistency (HLRC) [112] protocol. Like HLRC, the JavaSplit protocol implements the LRC memory model. Both protocols are home-based, i.e., each application protocol is assigned to node which maintains the object’s master copy. They employ similar mechanisms to inform remote threads about local modifications. While HLRC is primarily designed for page-based distributed shared memory, JavaSplit is designed for object-based systems. Like HLRC, our protocol is a multiple-writer protocol, i.e., it allows more than one process write into the same shared memory unit simultaneously. The advantage of multiple-writer approach is in the fact that a writer not need to obtain exclusive ownership of the object. JavaSplit protocol improves the scalability of HLRC and reduces its communication and memory overhead.

The JavaSplit memory management protocol differs from HLRC in several important aspects.
4.1. DISTRIBUTED MEMORY MANAGEMENT PROTOCOL

First, the memory synchronization operations, which occur during every acquire and release operation in HLRC, take place only when one thread passes lock ownership to another. This significantly reduces the number of the above synchronization operations. Each acquire-release pair triggers the memory synchronization logic at most one time. Moreover, a sequence of local acquire and release operations never triggers the memory synchronization logic. Second, the timestamps used for modification bookkeeping are no longer integer arrays whose length equals to the number of application threads (vector timestamps) but rather are pairs of integers. Finally, unlike HLRC, the proposed protocol manages the consistency information in a way that allows to dispose a large portion of outdated data without cooperation between nodes. Consequently, despite the general similarity, on the low level the JavaSplit protocol is essentially different from HLRC.

4.1.1 Lazy Release Consistency

In Lazy Release Consistency, lock operations induce memory synchronization. The two lock operations, acquire and release are equivalent to the JMM primitives lock and unlock, respectively [38]. The release-acquire pairs define a partial order among operations performed by the participating processes. This partial order, called happened-before I [55] and denoted by →, is defined as follows:

1. If $a_1$ and $a_2$ are accesses by the same process, and $a_1$ occurs before $a_2$ in program order, then $a_1 \rightarrow a_2$.

2. If $a_1$ is a release by process $p_1$, and $a_2$ is a subsequent acquire of the same lock by process $p_2$, then $a_1 \rightarrow a_2$.

3. If $a_1 \rightarrow a_2$ and $a_2 \rightarrow a_3$, then $a_1 \rightarrow a_3$.

LRC requires that when a process $p$ performs an operation $op$ (which can be a read, write, acquire or release), all operations that precede $op$ in the happened-before $I$ partial order appear completed to $p$. 
4.1.2 JavaSplit Implementation of LRC

A JavaSplit node is a JVM process which executes one application thread. The fact that a node executes a single thread significantly simplifies the instrumentation and allows to avoid synchronization when accessing frequently used runtime data structures.

In JavaSplit, the node that executes the thread that created an object is considered to be the home node of that object. The primary purpose of the home node is to maintain the object's master copy. Let $N$ be the home node of object $X$. Henceforth, we will say that $X$ is mapped to the node $N$. Although in JavaSplit the object-to-node mapping is static, the memory management protocol can be easily extended to accommodate runtime changes of this mapping.

The application objects are dynamically classified into local and shared. The former are accessed by a single thread while the latter can be accessed by multiple threads. A newly created object is classified as local. A local object has a single local copy accessible only by the local thread. An object $X$ becomes shared if a fetch request for $X$ is received from a remote node. (Thus, a local object may be referenced by a shared object.) The distributed memory management protocol is concerned only with the shared objects.

A node stores master copies of shared objects mapped to it in a data structure which we call the Shared Object Repository (SOR). An object copy is stored in SOR in a serialized form, as a byte array. A thread that wishes to access an object but does not have a valid cached copy fetches the object contents from the object’s home SOR. A cached copy is a regular Java object that is used for both reading and writing. The fact that the object master copy is stored as a byte array facilitates the process of fetching the object from its home by eliminating the need to serialize it when replying to a fetch request. The local application thread never uses an SOR object in its computation, but rather accesses it similarly to remote application threads, as described above. When a local object $X$ becomes shared, a master copy of $X$ is created in the shared object repository of its home node. Upon creation of the master copy, the local copy that was being used by the local thread becomes a cached copy which is equivalent to a cached copy used by a remote thread.

The execution of a thread is divided into intervals delimited by lock ownership transfer operations performed by the thread. (This is unlike the HLRC intervals which are delimited by acquire
and release operations.) At the end of an interval, a thread sends the modifications it has performed during the interval to the homes of the modified objects. Each home applies the received modifications to the master copies of the objects.

Since several writers may modify the object simultaneously, a writer must be able to identify its modifications in order to merge them with modifications of other writers. To achieve this, like most multiple writer schemes, the JavaSplit protocol uses twinning and differencing as follows. Before modifying an object that has not yet been modified during the current interval, a thread makes another copy of the object, called a twin. In the end of an interval, the modifications are calculated by comparing (differing) a cached copy with its twin, field-by-field. The contents of the modified fields are sent to the homes of the corresponding objects where they are used to update the master copies. Then, in order to avoid sending the same modifications twice, the contents of the twin are updated with the contents of the cached copy.

The happened-before-1 ordering is maintained by passing consistency data along with the lock ownership. During lock transfer, the releasing thread $R$ sends to the acquiring thread $A$ the list of objects that have been modified before $R$’s current release, according to happened-before-1. This list does not include objects of whose modifications $A$ is already aware. An entry of this list is called a write notice. On the basis of the received write notices, the acquiring process invalidates the local copies of the corresponding objects. Subsequently, when accessing an invalidated copy, a thread may...
need to obtain a newer version of the object from the object's home.

Figure 4.1 illustrates the JavaSplit memory management protocol in a simple example. The thread $A$ modifies the object $X$ in an acquire-release clause. The thread $B$ that wishes to read $X$ in an acquire-release clause protected by the same lock $L$ requests the lock from $A$. $B$'s lock request arrives to $A$ after $A$ has already performed a release and therefore it can grant $B$'s request. Hence, $A$ calculates the diffs and the write notices for all the objects it has modified during the current interval. The write notices are sent to $B$, while the diffs are sent to the homes of the modified objects. In the example, only the diff and the write notice belonging to $X$ are shown. Upon receiving the write notices, $B$ invalidates $X$. Consequently, when trying to read $X$, $B$ discovers that the local copy is invalid and fetches $X$ from its home.

**Avoiding Ending an Interval on Acquire Operation**

Unlike existing HLRC schemes, the JavaSplit protocol does not require flushing local modifications to the corresponding home nodes during each acquire and release operation. In JavaSplit, local modifications are flushed only when the current thread passes lock ownership to a remote node. This reduces the computational and communicational overhead associated with this operation, also known in the literature as ending an interval. Thus, the interval end is not triggered by local acquire and release operation, but is rather performed lazily.

The main reason existing HLRC protocols choose to end an interval during acquire operation is the difficulty in maintaining local modifications of objects that are invalidated as a result of lock acquisition. The JavaSplit memory management protocol solves this problem through employment of a modification merging technique described below.

Consider an object $X$ that has been modified by the local thread, but whose modifications has not been flushed yet. If $X$ is invalidated (due to a write notice received as a result of a local acquire operation) and then is accessed by the local thread, $X$ will be fetched from its home. However, if the contents of the local cached copy are replaced with the fetched, then local modifications will be lost, which violates the LRC program order requirement. To prevent the loss of local modifications, we determine the modified fields by comparing the cached copy of $X$ with its twin. Then, while copying the contents of the fetched copy to the cached copy, we do not update the modified fields, but rather leave them as they are. Assuming there are no race conditions on the modified fields,
their local values are in the most updated state in the entire system. These values will be flushed when the current interval ends.

When using the above technique, a special care needs to be taken in a situation in which the current thread ends an interval while fetching the contents of a modified object $X$ from its home. When the fetch reply arrives, $X$’s modifications may already be flushed and hence the contents of its cached copy is identical to the contents of its twin. Consequently, according to the above merging technique, the contents of the cached copy will be overwritten with the fetched copy of $X$. This may violate LRC program order if the local thread accesses $X$’s field which it has previously modified. The program order is violated because the local thread will not observe a value it has written, while no other remote thread has done any modifications to that field. To prevent this scenario we postpone transferring locks while fetching objects from their home. Thus, when a fetch reply is received, a proper merge between the cached copy and the fetched copy can take place.

**Refining Version Bookkeeping**

There is a coherency problem supporting LRC in a home-based protocol: the home (master) copy must contain the updates required by the consistency model when the object is accessed or fetched from its home. For instance, in Figure 4.1, the $B$’s data fetch request could arrive to $\text{home}(X)$ before $\text{home}(X)$ has updated $X$’s master copy, based on the diff transmitted by $A$. If left untreated, $B$ would receive a version of $X$ which does not contain $A$’s modification, which breaks the LRC memory model.

Most home-based protocols, including HLRC and the JavaSplit distributed consistency protocol, solve the problem by attaching versions to coherency units (memory pages or objects) and to write notices. In a write notice, a version is essentially a logical timestamp of the modification. Attached to a valid cached copy of a coherency unit, it indicates the current version of this copy. Attached to an invalid cached copy, it indicates the version the thread must obtain before accessing the coherency unit.

In HLRC, a version is a vector whose entries correspond to the application threads. The versions are managed as follows. At the end of an interval during which a thread $R$ has modified a page (or an object) $X$, the entry that corresponds to $R$ in $X$’s version is increased. The new version is attached to $X$’s diffs that are transmitted to the home. Upon receiving the diffs the home sets the
version of $X$'s master copy to the per-entry maximum of the current and the received versions. $R$ also attaches the new version to the $X$'s write notice that $R$ sends out when releasing a lock in favor of another thread $A$. Upon receiving a write notice for $X$ with version $v_{\text{recv}}$, the thread $A$ compares it with the version $v_{\text{local}}$ attached to the local cached copy of $X$. If $v_{\text{recv}}$ is newer than $v_{\text{local}}$, the cached copy is invalidated and the local version is replaced with the received which designates the coherency unit version that $A$ must fetch from the home the next time it accesses $X$. If indeed $A$ accesses $X$, the version of an invalid cached copy is attached to the fetch request message. If the requested version is not yet available, the home delays the reply until the necessary updates (from $R$) are received. In the reply to a fetch request, the home includes the current version of the sent copy, since the master copy could have been updated by other threads in the meantime. Upon receiving a fetch reply, $A$ updates both the contents and the version of the cached copy.

Vector timestamps do not scale well because their size is proportional to the number of threads. Since a version is attached to each valid coherency unit and each write notice, in a large-scale environment vector timestamps can occupy a considerable amount of space. Moreover, since nodes often include versions in protocol messages, the timestamp size has a non-negligible impact on communication overhead. In a object-based system such as JavaSplit, both the space and communication overheads are even more significant, because there are much more coherency units (i.e., objects) and consequently a greater number of write notices.

We resolve the above scalability issues by using version whose size is independent of the number of threads in the system. In addition to scalability, the advantage of our scheme lies the fact that a single timestamp specifies modification of many objects. Therefore, unlike vector timestamps versioning schemes, which attach a timestamp to each transmitted write notice and diff, the JavaSplit protocol sends only a single timestamp with a list of write notices or diffs. Thus, the communication overhead is even further reduced.

In JavaSplit, a version represents the interval during which a modification took place rather than the number of modifications performed by each thread on an object. At the end of an interval, i.e., when a thread $R$ transfers ownership of a lock $L$ to a thread $A$, $R$ creates a new version which is applicable to all the modifications in the ending interval. The new version $V$ is a pair $(L, c)$, where $c$ is a counter associated with the lock $L$ and which is incremented each time $L$ is transferred from one thread to another. $R$ attaches $V$ to all the diffs that it sends outs to their corresponding homes.
4.1. DISTRIBUTED MEMORY MANAGEMENT PROTOCOL

R also attaches V to the write notices it transmits along with the lock ownership to A.

The home node H may maintain several versions of the object X; one version per lock during interval of which the object has been modified. Upon receiving the diffs and the version V transmitted by R, the home node H updates X’s version as follows. The version \((L,k)\), attached to X, is replaced with \(V=(L,c)\) (assuming \(c > k\)). Any other version \((L_2,j)\) attached to the master copy of X remains unchanged.

Several versions can be also attached to an invalid copy of X. Upon receiving the lock ownership along with the version \(V=(L,c)\), the node A invalidates its cached copy of X unless one of the local versions attached to X is \((L,m)\) such that \(m \geq c\). If the object is indeed invalidated, the local version belonging to the lock \(L\) is replaced with \(V\). When fetching X from its home, upon access miss, A puts all the versions associated with \(X\) into the fetch request. The home H serves the request only after it receives all the modification designated by the attached versions, and by \(V\) in particular. The fetch reply includes all the versions attached to the master copy of \(X\). Upon receiving the reply, A replaces the old versions attached to \(X\) with those received from \(H\).

It is important to note that although in theory, the number of versions attached to an object can reach the number of locks in the system, it rarely exceeds two, in practice. This is due to the fact that access to an object is typically protected by a single lock (which is released after the object is modified). Consequently, despite the above rather intimidating upper bound, the JavaSplit version bookkeeping seems to be advantageous in comparison to the traditional vector timestamps scheme.

Another downside of the proposed protocol is the fact that JavaSplit versions are not always comparable to each other, unlike vector timestamps. If an object \(X\) is modified during two intervals that end with transfer of two different locks \(L_1\) and \(L_2\), it is impossible to determine the ordering of the two resulting versions \((L_1,i)\) and \((L_2,j)\). Consequently, a cached copy with a version \((L_1,i)\) that already contains the modifications associated with \((L_2,j)\) can be invalidated by the receipt of a write notice timestamped as \((L_2,j)\). This may result in unnecessary data misses.

Disposal of Outdated Write Notices

During the lifetime of an application, the HLRC-like protocols [12][89] accumulate all the received write notices, which is required for preserving the happened-before-1 ordering. Thus, if not collected, the number of write notices stored on a node is unbounded. Since, as already claimed above, the
number of write notices in a fine-grained system is much greater than in a page-based one, their storage can cause memory overflow (especially, in long-running applications).

When one thread $R$ transfers lock ownership to another thread $A$, $R$ must supply $A$ not only with the newly created write notices, but also with any write notices $R$ is aware of but $A$ is not aware of. Therefore, a write notice can be discarded only after every thread in the system received it. In order to determine the obsolete write notices all threads must participate in a distributed (garbage collection) protocol.

In large-scale systems, it is desirable to reduce the amount of system-wide cooperation between the nodes. Therefore, instead of using a distributed write notice collection protocol, JavaSplit limits the number of write notices stored per object, as follows. Whenever a new write notice for an object $X$ with a version $(L, i)$ is to be stored in the local write notice repository, any write notice with an earlier (comparable) version associated with $X$ is discarded. Consequently, the number of write notices per object is limited by the number of incomparable versions associated with $X$ (which is rarely more than one).

**Correctness Proof**

A memory management protocol is considered correct if it implements the desired memory model and preserves the liveliness of the computation.

**Lemma 4.1** JavaSplit memory management protocol preserves the liveliness of the computation.

**Proof:** The memory management operations that do not require interaction with the shared object repository, e.g., twinning, diffing, calculation of write notices and invalidation, eventually terminate. There are two types of interaction with SOR: fetch requests and update requests. An update request, i.e., the request to update a master copy of an object is served without a delay and therefore always terminates. The processing of a fetch request by an SOR can be delayed until all the update requests required by the fetch request version have been processed. The required updates are guaranteed to arrive eventually and therefore a fetch request eventually terminates. Since the termination of an update request does not depend on the termination of any fetch request, a deadlock cannot occur. As all the memory management operations eventually terminate, the protocol preserves the liveliness of the computation.

$\blacksquare$
4.1. DISTRIBUTED MEMORY MANAGEMENT PROTOCOL

Lemma 4.2 JavaSplit memory management protocol implements Lazy Release Consistency

Proof: In order to comply with LRC, our protocol must satisfy the requirements of the happened-before-I partial ordering that are enumerated in Section 4.1.1: (i) program order, (ii) release-acquire relation and (iii) transitivity.

The program order requirement is violated if a local read operation of a field \( f \) in some object \( X \), denoted by \( R_l(X,f) \), does not observe the value written by a preceding write operation \( W_l(X,f) \) and there is no remote write operation \( W_r(X,f) \) so that \( W_l(X,f) \rightarrow W_r(X,f) \rightarrow R_l(X,f) \).

The operation \( R_l(X,f) \) can observe a value different from the one written by \( W_l(X,f) \) only if there has been an access miss between the two operations, and the cached copy of \( X \) is updated with the data fetched from \( X \)'s home. Moreover, this is only possible if at the time of an access miss the modification of the field \( f \) has already been flushed to \( X \)'s home. (Otherwise, according to Section 4.1.2, \( f \) would not have been overwritten when processing the fetch reply.) However, if the value written by \( W_l(X,f) \) has already been flushed at the time of an access miss, then the version attached to the fetch request designates the version of \( X \) which includes the modification produced by \( W_l(X.f) \). Consequently, the fetch reply must include this modification (or a modification produced by some remote write \( W_r(X.f) \), so that \( W_l(X,f) \rightarrow W_r(X,f) \)). It follows that the outcome of a local write operation cannot be canceled as a result of an access miss and hence program order requirement is satisfied.

The relation between a release operation and subsequent acquisition of the same lock is ensured by the distributed synchronization protocol.

The transitivity requirement between lock and memory operations in the same thread is satisfied due to the fact the application thread does not reorder local operation.

The transitivity requirement between a write operation \( W_l(X,f) \) in thread \( T_l \) and a read operation \( R_2(X,f) \) in another thread \( T_j \) so that \( W_l(X,f) \rightarrow \text{rel}(L) \rightarrow \text{ack}(L) \rightarrow R_j(X,f) \) is satisfied as follows. When \( T_j \) passes ownership of the lock \( L \) to a thread \( T_a \) (which may be \( T_j \)) after completing \( \text{rel}(L) \), it also sends \( T_a \) a write notice indicating modification of \( X.f \). \( T_i \) also flushes the new value of \( X.f \) to \( X \)'s home node. The write notice passing logic guarantees that by the time a thread \( T_j \) completes the operation \( \text{ack}(L) \) it receives a write notice that contains a version associated with \( W_i(X.f) \) or another remote write operation \( W_k(X.f) \) so that \( W_i(X,f) \rightarrow W_k(X,f) \). Upon performing \( R_j(X,f) \), the object \( X \) is fetched from its home. The version received along with the write notice is attached
to the fetch request. Consequently, the home will reply only when the master copy contains all the updates associated with that version. Thus, the fetch reply contains the outcome of $W_i(X.f)$ or another write operation $W_k(X.f)$ so that $W_i(X.f) \rightarrow W_k(X.f) \rightarrow R_j(X.f)$

\section{4.2 Distributed Locking Protocol}

In Java, each object has a lock associated with it. An object’s lock does not necessarily synchronize accesses to the object contents but may be used to synchronize accesses to another object or a collection of objects. In addition to the conventional acquire and release primitives, a Java lock supports conditional waiting operations, wait and notify.

In JavaSplit, the distributed locking protocol treats locks independently of the objects they belong to. The implemented locking protocol is token-based, i.e., the lock ownership is represented by a token, which is passed between nodes. Naturally, to achieve mutual exclusion there is a single token per lock. Unlike traditional distributed locking protocols, the JavaSplit protocol supports conditional waiting. The thread join operation is implemented on top of the conditional waiting mechanism.

The node that has the lock token, henceforth the lock owner, does not necessarily lock it, i.e., it is not inside a critical section protected by this lock. The lock owner may acquire the lock whenever needed while any other node must first obtain the lock token and thus become the new owner of the lock. We will say that a node holds the lock if it is the lock owner and its thread is inside a critical section protected by this lock. In the application level, the lock owner may transfer the token to another node only if it (the owner) does not hold the lock.

Each lock has a home node whose primary role is to maintain the identity of the current lock owner. The home node of a lock coincides with the home node of the object it is associated with. However, this is not required by the presented protocol.

Whenever a node $R$ passes the token of a lock $L$ to another node $A$, $R$ also sends a token location message to $L$’s home notifying the home that $A$ is the new lock owner. In order to prevent home from updating the identity of the lock owner based on outdated messages, a counter of lock transfers is sent along with the token location message. The counter of a lock $L$ is attached to $L$'s token.
and it is incremented each time the token is transferred to another node. The home ignores token location messages with lock transfer counter that is not bigger than the counter associated with the latest valid information about the lock owner. Note that token location messages and usage of the lock transfer counter are not necessary to ensure correctness of the protocol. They merely reduce the time it takes for a token request to reach the lock owner. Thus, it is possible to decrease the number of token location messages by occasionally avoiding their transmission.

A node that passes a token of a lock \( L \) to another node \( A \) internally registers \( A \) as \( L \)'s new lock owner. Thus, every node that has ever owned \( L \), remembers to whom it has passed the token.

Initially, the token is located at the lock’s home node. A node that needs to obtain the token of a lock \( L \) sends a token request message to the lock’s home. The home forwards the request to the node it considers the lock owner. Each node that receives the request but does not own the lock forwards the message to the node it considers to be the owner, i.e., to the node it has passed the token to. Eventually, the request reaches the current owner.

The lock owner maintains the token request queue, which is passed along with the token. Upon receiving a token request from a node \( R \) the current owner adds the request to the request queue unless a request from \( R \) is already in the queue. Whenever the lock owner no longer needs the token, it removes the first request from the queue and passes the token along with the queue to
CHAPTER 4. TRANSPARENT DISTRIBUTED RUNTIME FOR JAVA

the corresponding requester node. If the request queue is empty the node remains the lock owner
and hence can enter critical sections associated with that lock. Thus, in the absence of contention,
critical sections inside loops can be executed quite efficiently, without communication on the critical
path of the computation.

Figure 4.2 illustrates the described above algorithm. The requester node sends a token request
for the lock $L$ to $L$'s home node. The home node forwards the request to the node that it considers
to be the current lock owner. However, since the destination node of the forwarded request has
already passed the token to the current lock owner, the request is forwarded again. The lock owner
adds the received request to the queue and when it no longer holds the lock, it transfers the token
to the requester as well as notifies the home node about the identity of the new lock owner.

4.2.1 Distributed Locking for Unreliable Channels

The algorithm described above works well under the assumption of reliable channels. However, if the
channels are unreliable, messages can be reordered, lost or duplicated, which violates the correctness
of the algorithm. The distributed locking logic implemented in JavaSplit uses the unreliable UDP
transport and therefore it takes additional precautions to ensure correctness.

All protocol messages, except the message that contains the token, are transmitted in fire-and-forget fashion. Upon receiving a new token request which cannot be served right away, the lock
owner sends a request acknowledgement message to the requester. If after a predefined timeout
period the requester does not receive either a request acknowledgement nor the token, the token
request is retransmitted. Thus, end-to-end reliability of a token request is ensured.

The message that contains the token is retransmitted until an acknowledgement from the destination, i.e., from the requester, is received. This guarantees that the token is never lost but introduces
two additional problems. First, the lock owner that is in process of transferring the token to the
requester node $R$ may forward to $R$'s own retransmitted token request messages. This is solved
by making nodes ignore their own token requests. Second, a duplicated token message can arrive
at a node after it has already received that message and has passed on the token, which potentially
violates mutual exclusion.

In order to prevent token duplication resulting from the second problem above, the counter of
lock transfers mentioned in the description of the algorithm in Section 4.2 which is attached to the
4.2. DISTRIBUTED LOCKING PROTOCOL

token message is employed. The recipient of a token message ignores it if the attached counter is not greater than the lock transfer counter in the previously received and accepted token message for that lock.

The fact that a lock request queue cannot contain more than one request from the same node plays an important role in an environment with unreliable channels. It prevents the lock owner from creating duplicated request queue entries as a result of duplicated token request messages. Nevertheless, since messages can be delayed, it is possible that an outdated token request message is received by or forwarded to the current lock owner. This results in creation of an unnecessary token request in the queue and eventually may lead to transferring the token to a node that does not need it. In order to prevent token loss in the case a token is passed to a node that no longer expects it, every node must accept the token, unless its counter of lock transfers is outdated, as described above. Note however, that in contemporary networks messages are rarely delayed or duplicated. Consequently, the situation described above does not occur very often.

4.2.2 Correctness Proof

A locking protocol is correct if it satisfies the following requirements. First, it must guarantee mutual exclusion. Second it must ensure *liveliness*. This means that if at any point of time there are nodes that request the lock ownership, one of these nodes must eventually receive it.

Although most locking schemes prevent starvation, the implemented protocol does not guarantee it, as there is no such requirement in Java. Note, however, that starvation is unlikely to occur in the scheme described above. In order for this to happen, a lock request must be indefinitely forwarded from node to node, never reaching the current lock owner.

In a token-based locking protocol the mutual exclusion requirement is implemented by ensuring there is a single token for each lock. Liveliness is achieved by preventing token loss and by ensuring that no node holds the token indefinitely.

**Lemma 4.3** The JavaSplit distributed locking protocol guarantees that at any point of time there is at most one token for each lock.

**Proof:** Initially a lock has a single token located on its home node. The token could be duplicated only due to retransmissions of the token message during token transfer. This would happen if a
node receives a token, passes it to another node, and then receives a retransmitted token.

Consider a node $A$ that has already received the token from a node $R$. If $R$ retransmits the token message due to an acknowledgement timeout, $R$ will ignore this message because the lock transfer counter attached to the token will not be greater than the lock transfer counter recorded by $R$ at the time of receipt of the first instance of the token message. It follows that a node can process the same token message only once. Therefore, the token cannot be duplicated.

\[ \textbf{Lemma 4.4} \] JavaSplit distributed locking protocol prevents token loss

\textbf{Proof:} A node $R$ that transfers the lock to another node $A$ retransmits the token message until an acknowledgement from $A$ is received. Thus, it is guaranteed that $A$ eventually receives the token message. If this is the first time that $A$ processes this message it will accept the token. Otherwise, $A$ will ignore the token message. In the latter case, the token is either owned by $A$ or it has been transferred to another node. If $A$ has transferred (or is in process of transferring) the token to another node, it will keep retransmitting the token, similarly to $R$, until the destination receives the token message. Thus, each node passing the token ensures the token message is not lost.

\[ \textbf{Lemma 4.5} \] Assuming the application thread eventually releases the lock, a node cannot hold the token indefinitely, if there is at least one other node that requests the token.

\textbf{Proof:} Assume, by contradiction, there is a node $R$ that holds the token indefinitely and there is a node $A$ that wishes to obtain the token. Eventually, $A$’s token request will reach $R$ through the chain of previous lock owners. (If the request is lost due to unreliable communication, it will be retransmitted by $A$ after a timeout.) Once the request reaches $R$, it is inserted into the lock’s request queue. Since the application thread eventually releases the lock, the token will be transferred to node $A$, in contradiction to our assumption.

4.2.3 Distributed Conditional Waiting

In Java, a \textit{wait set} is associated with each object. A thread that invokes the \textit{wait} method of an object is inserted into the wait set. Invocation of an object’s \textit{notify} method removes an arbitrary thread from the object’s wait set. Invocation of the method \textit{notifyAll} removes all threads from the object’s wait set. In order to perform the above operations a thread must be the owner of the
4.3. Instrumentation for Distributed Execution

The distribution transformation makes the bytecode aware of the distributed nature of the underlying environment. To preserve data consistency and ensure correctness of synchronization operations, all classes that can be used by the original application, including the system classes, must be instrumented. In order to support system classes we employ the TCH approach described in Chapter 3 and rename the instrumented classes. During the distributed execution, the application uses a class JS.A instead of a class A.
Figure 4.3: Conditional waiting for distributed token-based locking algorithms
4.3. INSTRUMENTATION FOR DISTRIBUTED EXECUTION

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOAD 1</td>
<td>...</td>
<td>load the instance of class A</td>
</tr>
<tr>
<td>DUP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETFIELD</td>
<td>A::byte</td>
<td>check the state field</td>
</tr>
<tr>
<td>IFNE</td>
<td>next GETFIELD</td>
<td>conditional branch</td>
</tr>
<tr>
<td>DUP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVOKESTATIC</td>
<td>Handler::readMiss</td>
<td>invoke the read miss handler</td>
</tr>
<tr>
<td>GETFIELD</td>
<td>A::myIntField</td>
<td>// read instance field</td>
</tr>
</tbody>
</table>

Figure 4.4: An instrumented read access of the field \textit{myIntField} in class \textit{A}. The instructions in bold are added by the instrumentation. If the object is valid for read, \textit{i.e.}, the value of the state field is not 0, only the first 3 added instructions are executed.

```java
public class JS.A extends JS.somepackage.C {
    // fields
    private int myIntField;
    protected JS.B myRefField;
    public JS.java.util.Vector myVectorField;
    // methods
    protected void doSomething(JS.B b, int n){...}
    public JS.B doSomethingElse(JS.java.lang.String str){...}
    // -------- JS utility fields --------
    public byte __JS__state;
    public int __JS__version;
    public int __JS__locking_statis;
    public long __JS__gloabl_id;
    ...
    // -------- JS utility methods --------
    public void __JS__serialize (__JS__ByteOutputStream out){
        super.__JS__serialize(out);
        out.writeInt(myIntField);
        out.writeIdOf(myRefField);
        out.writeIdOf(myVectorField);
    }
    public void __JS__deserialize (__JS__ByteInputStream in){
        super.JS_deserialize(in);
        myIntField = in.readInt();
        myRefField = in.readReference();
        myVectorField = in.readReference();
    }
    public __JS__Diff __JS__compare(JS.A twin){
        __JS__Diff diff = super.__JS__compare(twin);
        if(myIntField != twin.myIntField) {...}
        if(myRefField != twin.myRefField) {...}
        if(myVectorField != twin.myVectorField) {...}
        return diff;
    }
    ...
}
```

Figure 4.5: Utility class members added by the distribution transformation
The distribution transformation intercepts events that are important in the context of a distributed runtime. First, the bytecodes that start execution of new threads are replaced by calls to a handler that ships the thread to a node chosen by the load balancing function. Second, the lock-related operations are replaced by synchronization handlers. The lock related operations include the monitorenter and monitorexit instructions, synchronized methods, as well as wait, notify and join operations. Third, in order to preserve memory consistency, the rewriter inserts access checks before accesses to fields and array elements, e.g., getfield, putstatic, iaload, and lastore (see Figure 4.4). If an access check fails, a newer version of the object is obtained from another node. The rewriter does not intercept calls to I/O operations from the application classes. Instead, the system classes that perform low-level I/O are modified to achieve the desired functionality.

The instrumented classes are augmented with utility fields and methods that are used by the runtime. The class at the top of the inheritance tree is augmented with fields indicating the state of the object during the execution, e.g., access permission, version, and whether it is locked. This approach enables quick retrieval of the state information and allows the garbage collector to discard it together with the object. Each class is also extended with several utility methods, which are generated on the basis of the fields of the specific class. The most important utility methods are for serializing, deserializing, and comparing (differing). Figure 4.5 illustrates these added class members.

### 4.3.1 System Classes

An important benefit from using the TCH approach is the fact that there is no need to replace the system classes used by each local JVM. Such a replacement could seriously damage the portability of our system. Instead, an alternative set of distribution-aware system classes is created and used by the application.

Different JVM implementations may supply different implementations of system classes. Consequently, if a node creates its own instrumented set of system classes, the instrumented classes originating from different JVMs may not be interoperable. Therefore, we produce the instrumented versions using system classes of a single JVM brand and make all the participating nodes use the resulting classes. Currently, we use the system classes from Sun JDK 1.4.2 for this purpose. Since the instrumented system classes are implemented in pure Java, i.e., do not have native methods, they can be used on any JVM.
4.3. INSTRUMENTATION FOR DISTRIBUTED EXECUTION

Rewriting of certain native classes for distributed execution is not straightforward. Some native methods have special semantics which do not allow employment of the delegation pattern in the distributed context. For instance, the method `hashCode` defined in `java.lang.Object` must always return the same value for a particular instance of an object. However, unless specially treated, an invocation of this method on copies of the same object located on different JVMs will return different values. To address this issue, the problematic native methods where implemented distributively in a manner that suits their semantics. For instance, the `hashCode` method is implemented by extracting the hashcode value from the object’s globally unique id.

4.3.2 Static Fields

Static fields are variables that have a single incarnation per class in which they are defined. Semantically, they are fields of the corresponding class object. In the distributed context, the singularity of static variables require special treatment.

We support static variables by creating a special class `JS.A__static` for each original class `A` that has static variables. The instrumentation removes the static variables from `JS.A` and inserts them into `JS.A__static` as regular, non-static fields. The class `JS.A` is augmented with a constant static reference field, pointing to an instance of `JS.A__static`, which is treated as any other shared object. In the bytecode, an access to a static field of `A` is substituted with an access to the corresponding field of the `JS.A__static` instance, preceded by a regular access check that verifies the validity of the `JS.A__static` object. Thus, we use the same memory coherency mechanism for management of both static and regular fields.

4.3.3 Array Transformation

Although Java treats arrays as objects, it is harder to create distribution-aware versions for arrays than for objects. The main problem is that there is no array class definition, but merely language constructs that declare arrays based on definitions of existing classes. As a result, we cannot augment an array, as is, with the utility fields and methods needed by the JavaSplit logic.

We solve the above problem by creating a wrapper class for each utilized array type and augmenting the wrapper with the utility members. For example, a wrapper of an array of elements belonging to a class `A(A[])`, called `JS.array.A`, incorporates a reference to an array of type `A[]`
CHAPTER 4. TRANSPARENT DISTRIBUTED RUNTIME FOR JAVA

and the required utility members. An access to an array is transformed to an access to the array field of the wrapper object, preceded by an access check. Thus, we use the same coherency mechanism for managing regular objects and arrays.

Currently each array is treated as a single coherency unit. However, it is possible to divide big arrays into several coherency units. The wrapper approach allows this extension by allocating several instances of the utility fields, one for each region.

4.3.4 Rewriting Lock Operations

In Java applications there are a lot of unnecessary lock operations [19]. Often, especially in the system classes, locks protect accesses to objects that are accessed by no more than one thread during the entire execution. The overhead of the unnecessary locking may be negligible in Java. (In fact, many modern JVMs remove the unneeded locking operations from the execution path.) However, when rewriting bytecodes for distributed execution, one must be extra careful to avoid the performance degradation that may result from the increased cost of the transformed lock operations, which were unnecessary to begin with.

We reduce the cost of lock operations of local objects by avoiding the invocation of lock handlers when locking objects that are used only by one thread. Instead, a counter is associated with a local object, counting the number of times the current owner of the object has locked it since becoming its owner. (In Java, a thread can acquire the same lock several times without releasing it, and the object is considered locked until the owner performs the same number of releases.) Acquire and release operations on a local object merely increase and decrease the counter. Thus, the object is considered locked only when the counter is positive. If the object becomes shared, the lock counter is used to determine whether the object is locked. This approach bears a certain amount of similarity to the one presented in [10]. The cost of lock counter operations is not only low in comparison to the invocation of lock handlers for shared objects, but is also cheaper than the original Java acquire (monitorenter) operation.
4.4 Performance Evaluation

We have evaluated the efficiency of JavaSplit in several different settings. Our tests were performed on several combinations of operating systems, JVM implementations, and IP-compliant communication hardware. (This supports the claim of the Java-like portability of our system.)

The structure of the current section is as follows. First we present the instrumentation overheads and communication latency and throughput. Then we describe the performance of our benchmark applications on a collection of Intel Xeon dual processor (2x1.7MHz) machines running on Windows XP and interconnected by a 100Mbps Ethernet. Note that this configuration is not very advantageous in the context of distributed execution, since the ratio of bandwidth to CPU power is considerably smaller than in performance evaluations of similar systems [7, 92, 105]. The results obtained by this hardware setting when running on Red Hat Linux are not presented, due to their similarity to the results obtained under Windows XP. Finally, we evaluate the performance of an application, which does not scale on JavaSplit in the above setting. We compare its performance on a collection of Intel 2x2.4MHz dual processor machines running on Red Hat Linux in two communication settings: 100 Mbps Ethernet and 10 Gbps Infiniband in IP-over-Infiniband (IPOIB) mode. (The effective bandwidth provided in IPOIB is only 3.5 Gbps.) We observe that Infiniband significantly improves its performance, obtaining a scalable speedup.

In its current state, the system does not perform any optimizations on the bytecode rewriting process. We believe that existing optimizations [7, 104] based on flow analysis of the original bytecodes, e.g., access check elimination and batching, can reduce most of the instrumentation overhead. These optimizations have not been applied to JavaSplit yet, because, in contrast to scalability, they are not among our main research goals. Note, however, that there are certain applications for which the instrumentation overhead is negligible.

4.4.1 Instrumentation Overhead

The most significant changes, performance-wise, introduced by the JavaSplit bytecode rewriter are: (i) the addition of access checks before accesses to heap data (i.e., accesses to object fields and array elements), (ii) replacing lock operations (e.g., monitorenter and monitorexit) with distributed synchronization code.
CHAPTER 4. TRANSPARENT DISTRIBUTED RUNTIME FOR JAVA

Table 4.1: Heap access latency (nanoseconds). IBM JDK 1.3.0 optimized away the data accesses in the employed microbenchmarks.

<table>
<thead>
<tr>
<th></th>
<th>Original Sun</th>
<th>Original IBM</th>
<th>Rewritten Sun</th>
<th>Rewritten IBM</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>field read</td>
<td>0.84</td>
<td>0.07</td>
<td>1.82</td>
<td>1.63</td>
<td>2.2</td>
</tr>
<tr>
<td>field write</td>
<td>0.97</td>
<td>0.06</td>
<td>2.48</td>
<td>0.74</td>
<td>2.6</td>
</tr>
<tr>
<td>static read</td>
<td>0.84</td>
<td>0.06</td>
<td>1.84</td>
<td>1.61</td>
<td>2.2</td>
</tr>
<tr>
<td>static write</td>
<td>0.97</td>
<td>0.06</td>
<td>2.97</td>
<td>0.73</td>
<td>3.1</td>
</tr>
<tr>
<td>array read</td>
<td>0.98</td>
<td>0.09</td>
<td>5.45</td>
<td>4.99</td>
<td>5.6</td>
</tr>
<tr>
<td>array write</td>
<td>1.23</td>
<td>0.19</td>
<td>5.05</td>
<td>4.98</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Table 4.2: Local acquire cost (nanoseconds)

<table>
<thead>
<tr>
<th></th>
<th>Original Sun</th>
<th>Local Obj.</th>
<th>Shared Obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun JDK 1.4.2</td>
<td>90.6</td>
<td>19.6</td>
<td>281</td>
</tr>
<tr>
<td>IBM JDK 1.3.0</td>
<td>93.4</td>
<td>54.7</td>
<td>327</td>
</tr>
</tbody>
</table>

Table 4.3: Message latency (milliseconds)

<table>
<thead>
<tr>
<th>Msg. size</th>
<th>Windows Ethernet</th>
<th>Linux Ethernet</th>
<th>Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.66</td>
<td>0.19</td>
<td>0.04</td>
</tr>
<tr>
<td>65</td>
<td>0.65</td>
<td>0.17</td>
<td>0.04</td>
</tr>
<tr>
<td>650</td>
<td>0.84</td>
<td>0.30</td>
<td>0.04</td>
</tr>
<tr>
<td>6500</td>
<td>1.76</td>
<td>0.83</td>
<td>0.09</td>
</tr>
<tr>
<td>65000</td>
<td>7.08</td>
<td>3.97</td>
<td>0.28</td>
</tr>
</tbody>
</table>

Table 4.4: Throughput (Mbit/sec)

<table>
<thead>
<tr>
<th></th>
<th>Windows Ethernet</th>
<th>Linux Ethernet</th>
<th>Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Java</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Overhead %</td>
<td>9.56</td>
<td>9.60</td>
<td>10.24</td>
</tr>
</tbody>
</table>
4.4. PERFORMANCE EVALUATION

Table 4.1 shows the cost of heap data accesses in the rewritten code in comparison with their cost in the original Java program. These figures were obtained on a 2x1.7GHz station running on Windows.

In Sun JDK 1.4.2 for Windows, heap data accesses introduce a slowdown of between 2.2 and 5.6, whereas in IBM JDK 1.3.0 for Windows they seem to introduce a slowdown of between 12 and 55. This great difference is due to the fact that IBM JDK optimized away the repeated accesses to the same data in the original code. In the instrumented code, however, it appears that the access checks stand in the way of these optimizations. In real applications the data access patterns are not as trivial as in the employed microbenchmarks. Therefore, there are very few opportunities for such JVM optimizations. In fact, none of the tested applications has ever exhibited a slowdown greater than 2. In any case, existing techniques [7, 104] can be employed to eliminate a large portion of access checks and thus reduce the overhead of the heap data accesses.

Table 4.2 shows the cost of a local acquire operation, i.e., an acquire which does not result in communication. Although there is considerable overhead in acquiring a shared object, acquiring local objects costs less than in original Java. This is due to the lock optimization described in Section 4.3. Note, however, that there would not be much of a difference between the two metrics mentioned above, should a more recent JDK be used. This is due to the aggressive optimizations on the unnecessary locking operations in the modern JVMs.

4.4.2 Communication Latency and Throughput

The experiments presented in the following sections utilize two types of communication hardware: 100Mbit Ethernet and 10Gbit Infiniband.

Table 4.3 presents the latency of the utilized communication layer for different message sizes. During the execution, the nodes exchange messages ranging from several bytes to dozens of megabytes (depending on the application). For messages shorter than 65 kilobytes we use UDP messages, whereas longer messages are sent through the TCP protocol. In general, the latency of the utilized interconnect is important in the context of short messages, while the throughput is significant in the context of large ones.

The message latency of Sun JDK 1.4.2 is the highest. The latency of Infiniband is the lowest, one order of magnitude lower than in Sun JDK 1.4.2. The latency of IBM JDK 1.3.0 for Windows
is similar to Sun JDK 1.4.1 for Linux.

Table 4.4 describes the throughput of the interconnects and compares it to the bandwidth available from the native socket interface. The measurements show that Java sockets utilize around 90% of Ethernet bandwidth and 80% of Infiniband bandwidth. The overhead is due to the indirect access to the network interface, i.e., through the JVM. Note that, according to our tests, the bandwidth available in the IPoIB mode to the native sockets is only 3.5 Gbps, despite the fact we use a 10Gbps Infiniband.

4.4.3 Benchmark Applications

In our performance evaluation we use eight applications, the first three of which are from the Java Grande Forum Benchmark Suite [91].

1. **Series.** Computes the first $N$ Fourier coefficients of the function $f(x) = (x+1)x$. The calculation is distributed between threads in a block manner. We run Series for $N = 1000000$.

2. **SparseMatmult.** This benchmark multiplies unstructured sparse $N \times N$ matrices stored in compressed-row format with a prescribed sparsity structure. It exercises indirect addressing and non-regular memory references. We use $N = 500000$.

3. **RayTracer.** Renders a scene containing 64 spheres at resolution of $N \times N$ pixels. The worker threads of this application render different rows of the scene. We run the Ray Tracer for $N = 2000$.

4. **TSP.** The TSP application searches for the shortest path passing through all $N$ vertices of a given graph. The threads eliminate some permutations using the length of the minimal path known so far. A thread discovering a new minimal path propagates its length to the rest of the threads. During the execution the threads also cooperate to ensure that no permutation is processed by more than one thread by managing a global queue of jobs. We run TSP for $N = 21$.

5. **FileCrypt.** Encrypts a (very large) input file using Triple DES. The file is divided into blocks. In each iteration a thread gets a block, encrypts it, and writes the result into a separate file.
4.4. PERFORMANCE EVALUATION

The output of the program is an enumerated set of encrypted blocks. In our tests we encrypt a 1GB file into 5MB blocks. This benchmark demonstrates the I/O capabilities of the system.

6. **PrimeFinder.** Searches for prime numbers in the range \([2,N]\). The detected primes are written to the standard output. For better load balancing, the range is dynamically distributed among threads which cooperate to avoid checking the same number more than once. In our experiments \(N=3500000\).

7. **MaxClique.** Finds a the maximal clique in an arbitrary graph. We use a random graph with 52 vertices and edge density of 0.5. The execution of a thread is composed of iterations. In each iteration a thread checks whether a clique of a given size containing a given vertex exists. Whenever a thread finds a clique larger than known so far it propagates its size to the rest of the threads.

8. **KeySearch.** Implements a known plain-text attack on DES. The inputs of the application are an original message and the result of its encryption. The threads cooperate to find the encryption key in the range \(2^{24}\) of keys. Coordination among threads is needed to allow load balancing and in order to ensure the threads terminate as soon as the key is found.

4.4.4 Windows/Ethernet measurements

The experiments presented in this section are performed on Windows workstations interconnected by a 100 Mbps Ethernet. We employed the two most popular JVM implementation for Windows: Sun JDK (1.4.2) and IBM JDK (1.3.0). We used 2x1.7GHz dual processor workstations.

In all measurements two application threads were executed on each of the dual-processor nodes, each in an individual JVM process. We performed separate measurements for different JVMs. The executed programs were compiled using Sun JDK 1.4.2. To calculate the speedup, we divided the execution time of the original (unmodified) Java application with two threads on a single dual-processor machine by the execution time in JavaSplit. In the following graphs, the X axis indicates the number of utilized CPUs rather than the number of stations. The speedup is calculated separately for each JVM, each calculation based on the times produced by the same JVM. The results are presented in Figures 4.6, 4.7, 4.8, 4.9 and 4.10. Tables 4.5 and 4.6 characterize network traffic and memory consumption of the employed benchmarks, respectively.
Figure 4.6: Execution times (sec) and speedups
4.4. PERFORMANCE EVALUATION

Figure 4.7: Execution times (sec) and speedups
Figure 4.8: Execution times (sec) and speedups
4.4. PERFORMANCE EVALUATION

![Graph showing execution times and speedups for FileCrypt]

Figure 4.9: Execution times (sec) and speedups
Figure 4.10: SparseMatmult results in the Windows/Ethernet setting
### 4.4. PERFORMANCE EVALUATION

Table 4.5: Network Traffic (32 processes)

<table>
<thead>
<tr>
<th>Application</th>
<th>Bytes</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeySearch</td>
<td>690387</td>
<td>5159</td>
</tr>
<tr>
<td>PrimeFinder</td>
<td>2929838</td>
<td>20961</td>
</tr>
<tr>
<td>RayTrace</td>
<td>3250866</td>
<td>20485</td>
</tr>
<tr>
<td>FileCrypt</td>
<td>603993</td>
<td>4540</td>
</tr>
<tr>
<td>Series</td>
<td>4420462466</td>
<td>621</td>
</tr>
<tr>
<td>MaxClique</td>
<td>3256265</td>
<td>30630</td>
</tr>
<tr>
<td>TSP</td>
<td>16654791</td>
<td>73703</td>
</tr>
<tr>
<td>SparseMatmult</td>
<td>1542528177</td>
<td>804</td>
</tr>
</tbody>
</table>

Table 4.6: Memory Consumption (MB)

<table>
<thead>
<tr>
<th>Application</th>
<th>Uninstrumented app. with 2 threads</th>
<th>Single JavaSplit app. thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used</td>
<td>Heap</td>
</tr>
<tr>
<td>KeySearch</td>
<td>1.4</td>
<td>2.0</td>
</tr>
<tr>
<td>PrimeFinder</td>
<td>1.2</td>
<td>2.0</td>
</tr>
<tr>
<td>RayTrace</td>
<td>49.0</td>
<td>58.2</td>
</tr>
<tr>
<td>FileCrypt</td>
<td>10.5</td>
<td>11.8</td>
</tr>
<tr>
<td>Series</td>
<td>16.4</td>
<td>22.0</td>
</tr>
<tr>
<td>MaxClique</td>
<td>1.2</td>
<td>2.1</td>
</tr>
<tr>
<td>TSP</td>
<td>0.8</td>
<td>2.0</td>
</tr>
<tr>
<td>SparseMatmult</td>
<td>92.0</td>
<td>101.1</td>
</tr>
</tbody>
</table>
In KeySearch and PrimeFinder the two JVMs exhibit similar results and produce quite scalable speedups. In RayTracer and Series we observed that the execution times of the original and the rewritten programs on one JVM differ greatly from their counterparts on the second JVM, but are proportional to them respectively. Thus, the obtained speedups are almost identical.

In FileCrypt, the reduced speedup for 32 processors by both JVMs is mostly due to the fact that the execution time has a lower bound around the 30 seconds mark. At this point the application is limited by the hard disk access speed. Note that despite the differences on a smaller number of processors, both JVMs achieve exactly the same result on 32 CPUs. Without this bound, a higher speedup would be achieved.

The relatively low speedup obtained by TSP is caused by the instrumentation overhead and lock contention between threads residing on different nodes. The instrumentation slowdown of TSP is approximately 2. (Note that the TSP speedup for 2 CPUs is around 0.5.)

Figure 4.10 shows that SparseMatmul does not scale well. Its performance degrades with the number of processors due to its unscalable communication pattern. During the execution, the worker threads fetch big pieces of matrices from the main thread that creates the matrices. Since the main thread sends these matrix parts in a sequential manner, a greater number of threads results in a greater average fetch request delay, as shown in Figure 4.10(b).

4.4.5 Linux/Infiniband measurements

Our infiniband network was limited to 5 dual-processor 2x2.4GHz stations running on Linux. In the experiments described below we utilized these stations in two different settings: (i) 100Mbps Ethernet connection, and (ii) 10Gbps Infiniband connection (in IPoIB mode). All measurements were performed with Sun JDK 1.4.1. In order to give the reader a more comprehensive view despite the few available workstations, we present results for each even number of processors in the range [2,10]

Figure 4.11 presents the performance of SparseMatmul on Ethernet and on Infiniband. With Ethernet the performance does not improve after a certain number of processors. With Infiniband, however, we obtain a speedup proportional to the number of CPUs. Figure 4.12 presents the breakdown of the execution time for both settings. (The breakdown legend is the same as in Figure 4.10(b)) The wider bandwidth of Infiniband significantly reduces the read miss overhead.
4.4. PERFORMANCE EVALUATION

Figure 4.11: Comparison of Infiniband and Ethernet runs on Linux. Infiniband exhibits linear speedup.

Figure 4.12: Breakdown of the Linux runs. The black and the gray indicate the time spent in data misses and in local computation respectively.
Chapter 5

Fault-Tolerant Distributed Runtime

The difficulty in using a network of workstations for parallel processing is that a participating workstation may suddenly become unable to continue its part of the computation. This can be caused by a failure, e.g., a power outage, or by user intervention, e.g., shutdown or restart. These events result in the immediate loss of the volatile data and processes located on the node. Henceforth, for simplicity, we will generally refer to the above events as node failures. Unless the application or the runtime that executes it are fault-tolerant, the above actions will surely violate the integrity of the computation. This problem is especially acute in non-dedicated environments, where the runtime system has little control over the participating machines.

Under certain circumstances, fault-tolerance becomes crucial for successful completion of the execution. In environments that consist of a large number of nodes there is a non-negligible probability that some node fails before the computation is complete. Moreover, this probability is especially high for long running applications. Hence, in some cases, an application may never be able to complete in the absence of fault-tolerance.

JavaSplit transparently provides fault-tolerance to the applications and environments that require it. It is an optional feature that can be enabled, whenever needed. JavaSplit can preserve the integrity of the computation despite multiple, possibly simultaneous, node failures. The number of tolerated failures is configurable.

Fault-tolerance is accomplished by checkpointing and replication. At certain local execution points an application thread independently checkpoints itself. A snapshot of a thread does not
CHAPTER 5. FAULT-TOLERANT DISTRIBUTED RUNTIME

include the application’s shared objects, i.e., objects that can be accessed by more than one thread. Shared objects and thread states are made persistent by replication in the volatile memory of other nodes. Unlike many checkpointing algorithms, ours keeps only a single checkpoint per thread.

The key idea of our fault tolerance scheme is to capture the states of the application threads in such a way as to guarantee that rollback of any thread to its latest saved checkpoint will not violate system consistency. If this requirement is fulfilled, a failed thread recovers by simply restarting from its most recent checkpoint.

In contrast to providing persistence by flushing the data to a disk, replication in transient memory allows quicker access to the persistent data. Moreover, using disks of participating machines is unacceptable since they may become unavailable in most types of failures. An external persistent storage service is not always available and often has the same performance overhead as flushing the data to a disk.

Capturing and reestablishment of the thread state is not supported by the Java platform. In order to preserve portability, JavaSplit employs bytecode instrumentation to implement these features. This is accomplished by adapting the publicly available code of the Bukses project [102] to suit the needs of JavaSplit. Due to the instrumentation approach, the resulting thread checkpointing scheme can be employed on any standard JVM.

Scalability considerations played an important role in the design of the JavaSplit fault-tolerance scheme. Consequently, neither failure-free execution nor recovery from a failure require global cooperation of nodes. Moreover, recovery does not result in a rollback of non-failing nodes. During recovery, a non-failing node continues its normal execution, unless it wishes to acquire a lock held by a failed thread. In that case, it may need to wait until the failed thread is restored.

The implemented fault-tolerance protocol has a minor effect on the management of shared objects. It prevents optimizations that make a thread’s modifications visible on other nodes at times other than lock transfer. Nevertheless, it does not prohibit prefetching of shared data and therefore allows the number of data misses to be reduced.

In JavaSplit each thread executes in a separate JVM process. For simplicity, the reader can assume that each application thread runs on a separate node. In reality, failure of a node that executes a number of threads is equivalent to simultaneous failure of several virtual nodes.

The fault-tolerance scheme is implemented by extending the distributed memory management
5.1 System Model

We assume a fail-stop model [19], i.e., when a node fails, it stops communicating. Failures may occur simultaneously and even during recovery. We assume the existence of a mechanism which allows verification of node liveness.

Node failures may be permanent or transient. However, since the system is designed to treat a node recovered from a transient failure as an entirely new node, we can assume that all failures are permanent. Transient failures are modeled as permanent by assigning a new incarnation ID to every recovering node and ignoring messages destined to or sent by previous incarnations of the failed nodes.

We assume that the system incorporates a directory service which allows to determine network addresses of computation nodes and application threads. If a node fails and then is restored, the directory service distributes information about the new location of the recovered entities. Thus, a thread $A$ that retransmits a message to a thread $B$ which has failed and then restored on another workstation will eventually receive an update of $B$'s new network address and redirect the message to the new location.

5.2 Replication Groups

The participating nodes are organized into roughly equal-sized groups. Any data that needs to be preserved despite node failures is replicated in the transient memory of nodes belonging to the same group. Whenever a node belonging to a group fails, the remaining group members cooperate in order to ensure the safety and consistency of the objects, locks and threads mapped to the group.

The minimal size of a group is a parameter of the runtime. In order to guarantee that the runtime is resilient to $F$ simultaneous node failures, the size of a group should be at least $F+1$. In practice, the runtime can handle any number of failures within a group as long as the members do not all fail simultaneously. To improve fault tolerance, the system should add new nodes to groups whose size
CHAPTER 5. FAULT-TOLERANT DISTRIBUTED RUNTIME

drops below a predefined minimum. Note that a node joining a group must copy the group state from some member.

The policy of partitioning nodes into groups in wide area networks is important. There is an inherent trade-off between efficiency and fault tolerance. On one hand, geographic dispersal of group members can improve fault tolerance because it prevents dependencies between failures of members of the same group. For instance, in the context of idle cycle utilization, workstations situated in the same room might be reclaimed simultaneously, when their users return from a coffee break. On the other hand, since members of the same group frequently exchange messages (to keep the replicas consistent), their geographical proximity might increase overall system efficiency. The investigation of this tradeoff is beyond the scope of this work, which assumes a uniform network of workstations.

5.3 Persistence of Shared Objects

As described in Section 4.1.2, each node maintains a shared object repository which contains the master copies of the shared objects that consider the node to be their home. A master copy of an object is not accessed directly by the local thread. Conceptually, the local thread accesses a master copy using the same protocol as the remote thread.

There are two types of access to the shared object repository: fetch request and update request. The former is a request to observe the contents of a particular master copy. The latter is a request to modify the contents of a particular master copy. (Multiple update requests are typically batched into the same message.)

The contents of a master copy are visible to all threads in the system. Thus, if some thread $W$ fails after updating the master copy of an object and then is restored in a state in which it has not locally performed the corresponding writes yet, then the state of the system becomes inconsistent. A thread $R$ may observe the modifications performed by $W$ on the master copy, while $W$ may not even perform the same modifications due to non-determinism of the parallel execution.

To address this inconsistency problem, the JavaSplit checkpointing scheme guarantees that the a thread that sends out update requests never rolls back beyond that point. Moreover, it ensures safe delivery of the update requests to their destinations. Consequently, the update requests that arrive to a shared object repository remain valid despite any node failures. It follows that the shared
5.3. PERSISTENCE OF SHARED OBJECTS

object repository can be regarded as an independent database whose resilience to node failures is orthogonal to the fault-tolerance scheme of the other components.

The contents of the shared object repository are not included in the snapshot of the local thread. This significantly reduces both the time required to capture the thread state and the size of the resulting snapshot. The persistence of a shared object repository is ensured by replication in the transient memory of the nodes that belong to the same replication group as the current node. Henceforth we will refer to the group that contains the replicas of the object’s master copy, as the object’s home group. (Note that the object’s home group contains the object’s home node.)

The consistency of the shared object repository replicas is implemented using the primary-backup approach. The primary replica of the shared object repository is located on the home node of the objects it contains. As before, the fetch and update request are sent to the primary replica of the shared object repository. The fetch requests are served normally, whereas the update requests are forwarded the backup replicas. The state of the primary replica is updated only after the backups have acknowledged successful receipt of the modifications. Only then, an acknowledgement to the source of the update request is sent. When a master copy of an new shared object is added to the shared object repository, its contents are propagated to the backup repositories. The above scheme ensures that the modifications of the shared object repository state become visible to the application.
threads only after they have been applied to all the replicas. Figure 5.1 illustrates the basics of this scheme.

If the node that contains the primary replica fails and then is restored, the shared object repository state is copied from one of the backup replicas. If the node has failed in the middle of state update, the backup replicas may be inconsistent, as some have received the last update and some have not. However, in this case, an acknowledgement of the update request has not be sent. Consequently, when the primary replica is recovered, possibly from a less updated replica, it will receive a retransmitted update request and propagate it to the other replicas. Since update messages are identified by versions, only the less updated replicas will apply the modifications. The recovered primary replica will acknowledge the receipt of the update message only after all the replicas receive the modification and apply it, if necessary. Thus, the consistency of the shared object repository replicas is restored. Note that the thread checkpointing policy described in Section 5.4 guarantees that an update request will be retransmitted, despite failures of its source, until acknowledgement of the request is received.

5.4 Thread Checkpointing

To preserve the integrity of the system despite failures, we must ensure that the states of every two threads are consistent. In JavaSplit, there are two ways a thread can affect other threads: by updating a master copy of a shared object (stored in a shared object repository) and through lock-related operations. In this section, we describe a thread checkpointing policy that ensures inter-thread consistency with respect to shared data. The consistency of lock operations is guaranteed by another mechanism which is described in Section 5.5.

As mentioned above, a thread may affect the application data observed by other threads only by updating the shared object repositories. The modifications are sent to shared object repositories when the current thread transfers lock ownership to another thread. Therefore, a thread affects the data observed by other threads only when it transfers a lock. Consequently, if a thread rolls back past the latest lock transfer operation, the system state becomes inconsistent.

Figure 5.2 illustrates the above observation. If we roll back the thread R to q2, this will have the same effect as stopping R at q2 and then resuming it after a while. However, if we roll back R to q1,
the system state becomes inconsistent, because: (i) R returns to a state in which it has not sent the modifications, although they have already been applied to the shared object repositories, and (ii) there is no guarantee R will recreate these modifications after the rollback. After the rollback, the execution of R may differ from the original execution, because R may fetch different (more recent) copies of shared objects or be requested to transfer a lock at a different time. Moreover, R may read its own ‘future’ modifications of the shared objects, which is clearly illegal according to both LRC and the Java memory model.

To avoid inconsistency between the state of a rolled back thread and the rest of the system, each thread captures and saves its state just before sending out the modifications. In Figure 5.2 the checkpoints are denoted by \( c_i \). When a thread fails, it restarts from its latest snapshot and retransmits the modifications. This retransmission is the only effect the failure has on the shared object repositories and consequently on the other application threads. However, since the modifications can be applied on shared object repositories only once, the repeated modification requests will be ignored. The retransmission following thread recovery is necessary because the previously transmitted modification requests may have failed to arrive to their target as a result of channel unreliability or target node failure.

The thread’s snapshot is made persistent by replication in the group to which thread’s node belongs. When a group member receives the newer snapshot, it discards the previous one. The snapshot includes not only the thread execution state but also several runtime data structures, such as write notices and lock records, that need to be made persistent.
CHAPTER 5. FAULT-TOLERANT DISTRIBUTED RUNTIME

If a node fails, its thread is restored on the least loaded member of its group from the most recent checkpoint found in the group. To avoid execution of several threads on a single processor, a failed thread can be restored on a new node added to the group especially for that purpose. It is also possible to have a backup member that does not execute any threads unless a group member fails. Note however, that if as a result of a failure, a node executes two application threads, they run in two separate JVM processes and interact as if they are located on two different nodes.

If the node of a thread $T$ fails during snapshot transmission to the group members, some replicas might not receive the latest snapshot. In this situation, $T$ must be recovered from the most recent available snapshot. To achieve this, the group members cooperate to ensure they all have the most recent snapshot before restarting $T$. If during this process, the newest version of the snapshot is lost due to failures of the other group members that have received it, the $T$ is restored from the latest available snapshot. In this case, the consistency of the system is not violated because it is guaranteed that $T$ has not leaked out the diffs calculated its last checkpoint. This is due to the fact that a thread waits until all members of its group receive the snapshot before sending out the diffs.

5.5 Fault-tolerant Distributed Locks

According to the checkpointing scheme described in Section 5.4, the thread checkpoints itself before transmitting modifications of the shared objects. This happens in the process of transferring a lock token to a remote thread. The token message is sent immediately after transmission of the shared objects updates. Consequently, the lock state at the moment of token transfer is preserved as a part of the thread snapshot. This means that if a node that executes a thread $T$ fails after $T$ has transferred the token to another thread, $T$ will still remember to which thread it has passed the token when $T$ is restored on another node. In a failure-prone environment, this “memory” property is crucial for the correctness of the distributed locking protocol described in Section 4.2. Given this property, there are only a few modifications that need to be applied to the base protocol to make it fault-tolerant. In fact, the only problem that needs to be taken care of to ensure correctness is token loss.

In token-based mutual exclusion protocols, the token might be lost in transit or as a result of the lock owner failure. In the JavaSplit locking protocol, the token message is guaranteed to reach
its destination, i.e., the next owner thread, despite node failures. First, it is retransmitted until the destination acknowledges its receipt. Second, if the source fails, the token message is recreated and retransmitted, because the thread is always checkpointed just before token message creation and transmission. Finally, if the destination thread fails, the directory service will update the current node with the address of the node on which the destination thread is restarted. Then, the following retransmissions of the token message will be sent to the new location of the destination thread. It follows that the token may be lost only if the current lock owner node fails.

When the node of the lock owner fails, the token is not necessarily lost. If a snapshot has been taken between the receipt of the token message and the failure, then the failed thread does not forget that it is the current owner of the lock. Otherwise, after recovery the thread has no recollection of ever receiving the token message and thus the token is lost.

Token loss is detected in the following way. When passing the token to another thread $Q$, the thread $R$ records not only $Q$'s identity but also the current value of the lock transfer counter (see Section 4.2). The current value of the lock transfer counter and the identity of the owner is also attached to the token location message and then recorded by the home alongside the identity of the most recent known lock owner. When any thread $A$ (including the home) forwards the token request to another thread $B$, $A$ always attaches the identity of the thread that has passed the token to $A$ as well as the lock counter of that transfer. (If $A$ is not the home, it attaches its own identity and its latest recorded lock transfer counter.)

In the absence of a node failure resulting in a token loss, the value of the lock transfer counter on a thread $B$ that receives a forwarded token request should match the value of the lock transfer counter attached to the token request. If the received value is smaller than the local, it means that $B$ has failed after receiving ownership but before checkpointing. Upon detecting this situation, the thread $B$ requests the thread specified as the previous owner (before $B$) in the token request to reproduce the token and resend it to $B$.

Figure 5.3 illustrates token loss detection. The ownership of the lock $L$ is transferred from the thread $A$ to $B$, then to $C$, and then to the thread $D$ (see Figure 5.3(a)). The threads $A$, $B$ and $C$ checkpoint when passing on the ownership and hence make the identity of the next owner and the value of the lock transfer counter persistent. The thread $B$ updates the home of $L$, the thread $E$, with the lock transfer data when transferring ownership to $C$. When $D$ receives the ownership, it
Figure 5.3: Token loss detection by monitoring of the lock transfer counters
5.6 Correctness Proof

In this section we show that the integrity of the computation is not violated despite node failures. Our presented proof consists of lemmas analogous to those proved in Sections 4.1.2 and 4.2.2. Here, however, we take into the account the possibility of node failures.

5.6.1 Correctness of the Distributed Locking Protocol

**Lemma 5.1** The distributed locking protocol guarantees that at any point of time there is at most one token for each lock, despite node failures

**Proof:** In the absence of node failures the claim holds due to the correctness of Lemma 4.3. Therefore, token duplication may occur only if during token transfer the sender of the token R or/and the receiver of the token A fail.

The node A may receive the token message more than once if R retransmits the message due to a timeout or if R fails at any time between the current token transfer and the next time R checkpoints. Since A records the lock transfer counter associated with the token, it cannot accept the same token message more than once unless it fails itself and forgets the recorded lock transfer counter. Since the latest received lock transfer counter is made persistent along with the node's checkpoint, A may forget it only if it fails after receiving the token message, but before it checkpoints. If indeed A fails during this period of time, it will rollback to a state in which it has not received the token message and therefore it will accept the first token message received from R after the failure. However, in this situation, it is guaranteed that A has not passed on the token to another node yet, because we have determined that A has failed before checkpointing and therefore before transferring any token to another node. It follows that token cannot be duplicated despite node failures.
Lemma 5.2  \textit{JavaSplit distributed locking protocol prevents token loss, despite node failures.}

\textbf{Proof:} In the absence of node failures the claim holds due to the correctness of Lemma 4.4. In the presence of failures, a token may potentially be lost in transit or if the current lock owner fails.

Consider token transfer from a node $R$ to a node $A$. Before sending out the token $R$ checkpoints and makes persistent its decision to transfer the token to the node $A$. If $R$ fails at any point of time before taking the next checkpoint it will retransmit the token to $A$ as soon as it recovers. ($R$ cannot take the next checkpoint before receiving acknowledgement from $A$.) Even if $A$ fails before completing the transaction and, as a result, its network address changes (because it is restored in another process), the token messages transmitted by $R$ will reach it, since $R$ will eventually find out $A$'s new location. It follows the token cannot be lost in transit.

Now consider a node $A$ that has successfully received the token from a node $R$ and has acknowledged token receipt, so that $R$ will not retransmit the token message in the future. $A$ is the current owner of the lock. If $A$ checkpoints after receiving the token but before failing, it will be restored to a state in which it is still the current owner of the lock. However, if $A$ fails before checkpointing, it will be recovered to a state in which it does not see itself as the current owner of the lock. In this situation the token is indeed lost. However it is recovered by the token recovery procedure described in Section 5.5, as follows. A token request issued by one of the nodes (which can also be $A$) will eventually reach $A$ through the chain of previous owners (which have made the token propagation path persistent by including the next owner in snapshot taken during lock transfer). When the request arrives at $A$, it will discover that its local lock counter is inconsistent with one received from the previous owner and thus conclude that it has lost the token due to a failure. Subsequently the token is restored in the previous owner.

Lemma 5.3  \textit{Despite node failures, a node cannot hold the token indefinitely, if there is at least one other node that requests the token.}

\textbf{Proof:} In the absence of node failures this claim holds, by correctness of the Lemma 4.5. Assume, by contradiction, that there is a node $A$ that holds the token and does not pass it to any other node. Since the identity of the current owner is made persistent when transferring the token from node to
node, there is always a chain of previous owners through which the token request eventually reaches 
A. Assuming that the application thread releases the lock at some point, A will eventually decide 
to pass the lock to the requester.

5.6.2 Correctness of the Memory Management Protocol

**Lemma 5.4** JavaSplit memory management protocol preserves the liveliness of the computation, 
despite node failures

**Proof:** In the absence of node failures the statement is correct according to Lemma 4.1. Here, we 
show that any memory related operation eventually terminates, despite node failures.

The memory management operations that do not require interaction with the shared object 
repository, e.g., twinning, diffing, calculation of write notices and invalidation, eventually terminate. 
There are two types of interaction with SOR: fetch requests and update requests. An update request 
is acknowledged by the primary replica of the SOR, when all the replicas have applied the received 
modifications. Since this is done eventually, an update request always terminates. The processing 
of a fetch request by an SOR can be delayed until all the update requests required by the fetch 
request version have been processed. The required updates are guaranteed to arrive eventually, 
despite failures, because the update requests are made persistent before their transmission. Thus, 
a fetch request eventually terminates. Since the termination of an update request does not depend 
on the termination of any fetch request, a deadlock cannot occur. As all the memory management 
operations eventually terminate, the protocol preserves the liveliness of the computation.

**Lemma 5.5** JavaSplit memory management protocol implements Lazy Release Consistency despite 
node failures.

**Proof:** In the absence of node failures the claim holds due to the correctness of Lemma 4.2. Below, 
we show that the execution in the presence of failures is equivalent to a possible execution without 
node failures. Consequently, the correctness of the current lemma follows from Lemma 4.2.

As described in Section 5.3, the shared object repository is made fault-tolerant through the 
primary-backup approach. According to the description of this scheme, an update request cannot 
be lost due to the failure of the primary replica. The incoming update requests are first applied
to the backup replicas, then to the primary replica. An acknowledgement to the source of the update request is sent only after the primary replica is updated. If the primary fails before sending an acknowledgement, the source of the update request will retransmit the update. Otherwise, if the primary fails after sending an acknowledgement, it is guaranteed that all the backup replicas have received the update. Therefore, in the latter case, when the primary is restored it will have the above updates already applied to it. In the case the primary fails when not all backups have successfully received the update, it does not matter whether the primary is restored from the most updated replica. If the primary is restored from a replica that has not received the update yet, it will delay serving fetch requests that require the most updated version, similarly to the base memory management protocol. Eventually, the missing and yet unacknowledged update request will be received and applied to the primary. It follows from the above, that the observed behavior of the shared object repository is the presence of failures is equivalent the behavior of the SOR in the base memory management protocol when there are no failures.

Now we consider the behavior of the application thread with respect to the shared object repository and hence with respect to the other threads. A thread affects SOR only by sending it update requests during lock ownership transfer. A thread cannot affect the data observed by other threads in a way other than by sending updates to SOR. Consequently, a thread may potentially violate memory consistency only if it sends out update requests, fails and then is recovered in a state in which it will not send out exactly the same updates. Below we show why this is impossible.

Before sending out update requests, a thread $T$ checkpoints itself and sends the resulting snapshot to its group members. Only after all group members have successfully received the snapshot, $T$ transmits the updates. If $T$ fails and is recovered, it will retransmit the updates until it receives an acknowledgement from the SOR, which ensures they are delivered. It follows, that the updates will eventually be delivered to the SOR, despite $T$ failures. Thus, the behavior of a thread that sends out update requests is equivalent to its behavior in the absence of failures.

Thus, the replication scheme of the shared object repository and checkpointing before transmitting update requests guarantee that the state of the application threads is consistent with the state of the SORS, despite node failures. It follows that the behavior of a thread with respect to SOR is equivalent to a possible behavior of a thread in the absence of failures.

\[\square\]
5.7 Portable Checkpointing for Java Threads

The Java platform does not support capturing and reestablishment of a thread state, which is required by our fault-tolerance scheme. Fortunately, there are several frameworks that use instrumentation to augment Java with this functionality [88, 97, 102]. This allows us to support fault-tolerance without compromising portability. In our implementation, we employ the Brakes project [102].

In the JVM level, the execution state of a Java thread is a stack of method frames. A method frame contains the method’s private stack and its local variables. For each method invocation a new method frame is created. Whenever a method $B$ is invoked from a method $A$, $B$’s frame is put on top of $A$’s frame. Upon return from a method, the associated frame is popped from the stack. Thus, the frame of the currently executing method is always on top of the stack. The main problem in capturing the thread state in a standard JVM is that the contents of a method frame are not accessible outside the method.

To capture the state of a thread, Brakes unrolls the stack of frames while saving the frame contents. Upon receiving a command to make a snapshot, it saves the contents of the current (topmost) frame and returns from the associated method (and thus removes the topmost frame from the stack). The frame saving procedure is repeated, until there are no more application frames on the stack. By the end of this process the contents of all method frames are saved. Along with each frame Brakes also saves the frame’s local program counter. To implement thread state capturing as described above, the Brakes transformer inserts code after each method invocation (see Figure 5.4(a)). The inserted code inspects a certain Boolean variable to find out whether the thread is currently in the process of capturing its state. If indeed this is the case, the code saves the current frame and program counter and then returns from the current method. Otherwise, the execution of the method continues normally.

To reestablish the thread state Brakes invokes the method that has been at the bottom of the execution stack during checkpointing. Immediately after entering the method, it reconstructs the method’s frame according to the snapshot. Then, it jumps to the saved program counter, i.e., to the invocation of the method from which it has returned during checkpointing. The frame reconstruction procedure is repeated until the stack of frames is fully reconstructed. After this, the execution continues normally. To implement thread state reestablishment, the Brakes transformer inserts code
Figure 5.4: Java thread state capturing and reestablishment with Brakes
5.7. **PORTABLE CHECKPOINTING FOR JAVA THREADS**

at the beginning of each method (see Figure 5.4(b)). The inserted code inspects a Boolean variable to find out whether the thread is currently restoring its state. If this is the case, the current frame is reconstructed and the execution jumps to the saved value of the program counter, thus skipping the already executed instructions.

The instrumentation overhead of a normal execution is 4 bytecode instructions per method call. The time overhead of state capturing and reestablishment greatly depends on the application.

Although only the methods that might be involved in the stack manipulations described above must be instrumented, JavaSplit does not analyze the application to determine these methods and transforms all user code. Such an analysis could improve the speed of local execution. The methods of the JavaSplit runtime classes, however, are instrumented only if their frame may be located on the execution stack during checkpointing.

Brakes bytecode transformation must be applied after the distribution transformation described in Section 4.3. Performing transformation in the opposite order would make the code inserted by Brakes distribution-aware and therefore incorrect. Note that this ordering restriction applies to the instrumentation of a specific class and, therefore, the checkpointing transformation can be applied to one class before the distribution transformation is applied to another.

As mentioned in Chapter 3, unless the JVM is modified, a class can be instrumented before the execution begins (*static instrumentation*), or while it is being loaded into the JVM (*dynamic instrumentation*). The advantage of static instrumentation is that the rewriting is performed offline and therefore does not affect the execution time. The main advantage of dynamic instrumentation is that it does not require *a priori* knowledge of the set of classes used by the program (*closed world assumption*). Since reflection allows the loading of classes whose identity is determined at runtime, it may be impossible to determine the transitive closure of classes used by a program. Moreover, classes created during the execution can only be instrumented dynamically. Both distribution and checkpointing transformations can be performed statically or dynamically, as long as the aforementioned ordering restriction holds.
5.8 Incremental Checkpointing

When using the Brakes checkpointing framework, the entire heap of the application thread is serialized by the standard Java serialization facilities. In JavaSplit, the heap includes the local cached copies of the application objects and a number of JavaSplit runtime data structures (such as the lock database). Unfortunately, the Java serialization has a considerable computational overhead. Consequently, the thread checkpointing time can be significant when the number of serialized objects is great. Since checkpointing is performed on the critical path of the execution, it can have a non-negligible impact on the overall system performance.

To address this problem, we reduce the number of serialized objects by implementing an incremental checkpointing scheme. In the heart of this optimization lays the idea of excluding previously checkpointed data from the current snapshot. Only the data items that have changed since the last snapshot was taken are included in the new snapshot. This applies not only to the cached copies of application objects but also to the elements of the JavaSplit runtime data structures.

To realize incremental checkpointing, JavaSplit monitors creation, modification and removal of data items. The monitoring of the new and modified cached copies fits naturally into the present monitoring of application data accesses. Disposal of application objects is intercepted through the APIs provided by the standard java.lang.ref package, which allow to observe the outcome of Java garbage collection. The monitoring of JavaSplit runtime data structures is incorporated into the implementation of the runtime logic, rather than accomplished by means of bytecode instrumentation.

When checkpointing, only the new and modified data items are inserted into the snapshot. A list of deleted data items is also included, in order to allow their removal from the previously saved snapshot. The new and modified data items that are marked as deleted are not inserted into the new checkpoint.

The contents of the cached copies and the runtime data elements are written into the snapshot by custom serialization methods, rather than using Java standard serialization. This further reduces the thread checkpointing time. Custom serialization is accomplished by addition of custom serialization methods to the classes that need to be inserted into the checkpoint. These methods are incorporated by the instrumentation into the application classes and are added manually to the
5.9. PERFORMANCE EVALUATION

relevant JavaSplit runtime classes. (Note, although the checkpointing time is significantly reduced by custom serialization alone, i.e., without incremental checkpointing, our experience shows that in many cases this improvement does not achieve the desired performance.) When serializing a cached copy, its reference fields are replaced with the ID of the referenced objects.

Upon receipt of the thread snapshot each member of the replication group merges the received snapshot with the locally stored snapshot, which results in the most recent snapshot of the thread.

The incremental checkpointing scheme does not only decrease the checkpointing time, but also minimizes the snapshot size. As a result, snapshot transmission time to the group members and the associated networking overhead are also significantly reduced.

It should be noted that support for the described above scheme requires non-trivial modifications in the internal constructs of the Brakes framework. A considerable effort has been invested into augmenting Brakes with the desired functionality.

5.9 Performance Evaluation

We have evaluated the performance of JavaSplit with fault-tolerance support using a collection of Intel Xeon dual processor (2x1.7GHz) machines running on Windows XP and interconnected by a Gigabit Ethernet. In our experiments we employed SUN JRE 1.4.2. We used five applications, the first two of which are from the Java Grande Forum Benchmark Suite [91]: Series(N=2500), RayTracer(N=100), TSP(N=14), KeySearch(2^14) and PrimeFinder(N=3,000,000). A detailed description of these benchmarks can be found in Section 4.4.4.

We execute these applications in the following settings:

1. **Original Java**: An original (not instrumented) Java program running on a standard JVM.

2. **JavaSplit**: A program instrumented to be distribution-aware running on JavaSplit.

3. **JavaSplit-FT**: A program instrumented for distribution-awareness and checkpointing, running on JavaSplit.

Since JavaSplit and JavaSplit-FT are distinguished mainly by the fault-tolerance support, we can learn about the overhead of our fault-tolerance scheme by comparing their performance.
To explore how instrumentation affects local execution time, we ran each benchmark application with a single worker thread. Since JavaSplit and JavaSplit-FT do not perform any networking when running on a single node, the observed overhead is due to instrumentation. The results of this experiment are presented in Figure 5.5.

In PrimeFinder, the instrumentation overhead is negligible. This is because most of its run time PrimerFinder spends printing prime numbers on screen, rather than in computation, which is instrumented more heavily. In Series, JavaSplit execution time is smaller than the original. The reason for this phenomena is the aforementioned optimization of a local lock acquisition. Since JavaSplit executes a single application thread, all lock acquisitions are local, which, according to Table 4.2, gives JavaSplit an advantage over the original application. The overhead of Brakes transformation is not very significant, because there are few method invocations in Series. TSP and RayTracer exhibit overhead of approximately 130% on JavaSplit. These applications frequently access arrays and object fields. Since JavaSplit instruments heap accesses to make them distribution-aware, it introduces non-negligible overhead into these benchmarks. The overhead of Brakes transformation
5.9. PERFORMANCE EVALUATION

in RayTracer is quite significant, because of the frequent use of method invocations in the original application. Brakes introduces overhead of 4 bytecode instructions per each method invocation.

It is important to understand that in its current state, the system does not perform any optimizations on the bytecode rewriting process. The existing optimizations [7, 104] based on flow analysis of the original bytecodes, e.g., access check elimination and batching, can reduce most of the JavaSplit instrumentation overhead. The Brakes instrumentation can also be optimized. Although Brakes transformation needs to be applied only to the methods that are involved in stack capturing and reestablishment (see Section 5.7), it is currently applied to all application methods. Bytecode flow analysis can be employed in order to determine the exact set of methods that need to be rewritten. The above optimizations have not been implemented, because, in contrast to scalability, they are not among our main research goals.

Figures 5.6, 5.7 and 5.8 present the results of our scalability-related experiments. In these experiments, two threads are executed on each of the dual-processor nodes. Each thread runs in a separate JVM process. The bytecode instrumentation is performed statically, before the execution begins. In general, both systems exhibit good scalability. As expected, the scalability of JavaSplit is better than that of JavaSplit-FT, however, not by far. In the latter, the increased overhead is mostly due to the checkpointing and instrumentation, as described below.

Figure 5.9 presents the breakdown of an application thread time when executing the benchmarks on 28 processors using JavaSplit and JavaSplit-FT. The lock fetch time on JavaSplit-FT is greater then on JavaSplit. This is due to the fact that a thread that passes ownership of a lock to another thread must first checkpoint and distribute the snapshot to the replication group members. Although this checkpointing overhead is not significant in comparison to the application execution time, it introduces a non negligible delay on each lock fetch operation, especially when a number of threads contend for the same lock. In Series, the data fetch time increases significantly when running on JavaSplit-FT, because a considerable amount of local objects become shared throughout the execution. Since the application thread must checkpoint whenever another thread fetches the contents of a newly shared object, a delay is introduced on the data fetch path. The difference in the breakdown of the RayTracer benchmark can hardly be used for comparison due to somewhat irregular behavior of worker threads in this application. The data fetch time and the wait operation time varies from worker to worker and from one execution to the next.
Figure 5.6: Benchmark execution times (in seconds). The dashed line indicates the execution time of the original benchmark using two threads in a single JVM running on a dual-processor machine. The horizontal axis of each graph represents the number of processors.
Figure 5.7: Benchmark execution times (in seconds).
Figure 5.8: Benchmark execution times (in seconds).
5.9. PERFORMANCE EVALUATION

Figure 5.9: Time breakdown of JavaSplit and JavaSplit-FT on 28 processors
Chapter 6

Speculative Locking

The overhead of lock operations often becomes a performance bottleneck preventing parallel applications from achieving the desired degree of scalability. In the presence of lock contention, i.e., when several threads compete to acquire a lock, the execution of critical sections protected by the same lock is serialized. In this situation, a thread may block for a long period of time, waiting for its turn to execute a critical section. In the absence of lock contention, the overhead of lock acquisition may still be relatively high, because the acquiring thread often needs to cooperate with other threads to ensure mutual exclusion.

In parallel applications executing in a distributed environment, such as JavaSPlit, the performance problems induced by the locking are especially acute. In order to ensure mutual exclusion, a process acquiring a lock needs to communicate with remote nodes. This may result in long delays on the critical path of the execution even when no other process competes for the ownership of the same lock. Furthermore, since the number of available processors in a distributed runtime is potentially larger than that in commonly used multiprocessors, the scalability requirements are higher.

In many cases, critical sections protected by the same lock can be executed in parallel without creating a data conflict and thus violating application correctness. Threads entering critical sections guarded by the same lock do not necessarily access the same data, because application programmers tend to protect multiple data elements by a single lock. Moreover, even if the processes access the same data element, the correctness of the application may be violated only in the case one of the processes modifies the data.
Speculative locking is an optimistic concurrency control mechanism that exploits the above observation. During lock acquisition, instead of waiting for a lock to be properly acquired, a thread optimistically continues executing. This removes the acquisition overhead from the critical path and allows concurrent execution of critical sections protected by the same lock. In case a data conflict is detected, the thread rolls back to a state preceding the lock operation. Speculative locking has been previously explored in the context of shared-memory microprocessors [74, 84, 86]. These works suggest speculative locking schemes that are implemented as hardware extensions of existing mechanisms. They rely on cache coherence protocols to preserve data integrity and to detect data conflicts. The thread state is restored using micro-architecture support for thread-level speculation [23, 37, 61], reorder buffer and custom hardware additions.

To reduce the performance overhead of lock operations in JavaSplit, we devise a novel speculative locking scheme for a fault-tolerance distributed runtime environment. This scheme is generic, i.e., it can be employed not only by JavaSplit but also by other fault-tolerant distributed runtime systems. Support for fault-tolerance is used to implement local rollback required in case of misspeculation. To the best of our knowledge, we are the first to suggest employment of speculative locking in a general-purpose distributed runtime. Our speculative locking protocol differs from the multiprocessor schemes [74, 84, 86] mentioned above. It does not necessitate taking a checkpoint before each speculative acquisition and it allows thread to speculatively acquire an arbitrary number of locks simultaneously rather than acquiring one lock at a time. It supports both speculative and blocking acquisitions; a process may decide whether to speculate on a particular lock based on the history of misspeculations.

In this chapter we present our generic speculative locking scheme for a fault-tolerant distributed runtime (Section 6.1) and describe its implementation in JavaSplit (Section 6.2). In Section 6.3, we show that speculative locking can yield a significant performance improvement in the above setting.

### 6.1 Speculative Lock Acquisition

To implement speculative locking in a distributed environment, we replace the blocking lock acquisition with a non-blocking lock request operation. After requesting the lock, the process optimistically continues its execution as if the lock has already been acquired, while the lock ownership is being
fetched in parallel to the computation. If at some point a data conflict is detected, the process rolls back to some state preceding the speculative acquisition. Figure 6.1 compares blocking lock acquisition with speculative.

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 *speculating process*.

### 6.1.1 Data Conflict Detection

A data conflict occurs when 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 own data accesses and modifications performed by other processes. The data collision check takes place when the lock ownership is properly acquired. The speculating process compares the list of speculatively accessed data elements against the modifications it should have observed had it performed a blocking lock acquisition. If a match is found, the process rolls back.

In some systems, it may be possible to detect data conflicts before the process becomes the rightful owner of the lock it has speculatively acquired. The proposed scheme does not prohibit a process from performing such an early conflict discovery.
CHAPTER 6. SPECULATIVE LOCKING

Many distributed runtime environments, e.g., systems based on software shared memory, 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 those systems and often require only a slight adjustment to support speculative locking. Moreover, the memory model of most recent systems is Release Consistency[34], implementation of which usually requires passing a list of modifications along with the lock ownership. This greatly facilitates the implementation of data conflict detection.

6.1.2 Managing Recoverable States

In a fault-tolerant runtime, a failed process can be restored in a state consistent with the rest of the system. Although in most cases it is desirable to restore a process in the most recent reconstructible state, it may be possible to restore it in a number of less recent states without violating system consistency. Henceforth, we will call such process states recoverable states. In checkpointing-based fault-tolerance schemes, recoverable states usually correspond to the saved checkpoints; in logging-based schemes there are ranges of recoverable states.

A process that detects a data conflict between a local data access and a remote modification needs to roll back to a state preceding the local access. To ensure that such a rollback is possible, we must make certain that a recoverable state preceding the conflicting data access exists. Since a data collision can only occur in a speculative mode, it is sufficient to guarantee that there is always a recoverable state that precedes the beginning of the speculative mode.

In a fault-tolerant system, a process has at least one recoverable state. Consequently, there is always a recoverable state when entering the speculative mode. Hence, the challenge is to ensure that the recoverable states created before the speculation are not all discarded during the speculative mode. For instance, consider a checkpointing-based scheme in which a process keeps only the latest checkpoint (which comprises its recoverable state). If a new checkpoint is taken while in speculative mode, the previous checkpoint is discarded and the process is no longer able to roll back to a pre-speculation state.

The conclusion from the above observation is that in some fault-tolerant schemes we may need to take action to preserve a recoverable state that precedes the speculation. This can be achieved either
by preventing its disposal or by precluding creation of newer recoverable states while the process is in speculative mode. In most fault-tolerance schemes these methods require minimal changes in the original protocol.

In checkpointing-based 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. The optimal checkpointing frequency is derived from checkpointing cost and rollback probability. An alternative method to minimize the amount of lost computation is to enforce a checkpoint prior to speculation if the most recent checkpoint has been taken too long ago.

6.1.3 Implementing Efficient Local Rollback

The local rollback procedure required to support the presented speculative locking scheme in a fault-tolerant system can be implemented using the existing process recovery facility. However, since the process recovery logic is designed for restoring the process state in the case of a node failure, it induces unnecessary overheads when employed to implement local rollback. In the presence of frequent data conflicts, the overhead of a local rollback should be minimized. Therefore, we propose to compose the rollback procedure from the building blocks of the recovery logic, as described below. In any case, it is desirable that during local rollback the process network address does not change. This allows remote requests to arrive to the restored process without the delay associated with updating process whereabouts in a distributed directory service.

Typically, resilience to node failures is achieved by saving the data necessary for process recovery in a remote device, be it volatile memory of another node or a disk-based stable storage. When a node fails, process recovery data is transferred to a new node on which the process is restored. To avoid unnecessary communication, recovery data required for a rollback of a process can be maintained locally, e.g., in the process heap. For example, in checkpointing-based schemes, process checkpoints needed only for the purpose of speculative locking can be stored in local memory rather than in a remote location.

The existing fault-tolerance mechanisms assume that the runtime state of a failed process is lost. Therefore, the entire process runtime state is made persistent and the recovery procedure is designed to restore the process from scratch. However, when a process rolls back due to a data conflict the
CHAPTER 6. SPECULATIVE LOCKING

states of both the process and the associated system logic are still available. It follows that (i) portions of the process and system state may not need to be reconstructed and that (ii) a part of the persistent recovery data is unnecessary to support local rollback. Consequently, we suggest that only the data required for the purpose of local rollback is maintained and that reusable parts of process runtime state are not reconstructed.

6.1.4 Rollback Dependencies

Under certain circumstances, a rollback of a misspeculating process can result in rolling back of additional processes. This may significantly increase the misspeculation overhead and thus hinder the benefit from speculative locking.

In some fault-tolerant systems, process recovery requires a subset of non-failed processes to roll back in order to maintain a valid execution state. Since the local rollback operation is based on the process recovery procedure, misspeculation may also result in a rollback of more than one process. Usually, such fault-tolerance schemes assume that process failures are infrequent. Since this assumption does not project on the likelihood of misspeculations, the rollback overhead may saturate the performance gains from speculative locking. Both the number and the employment of such schemes is limited as they are considered less efficient and not very scalable.

A similar inter-process rollback dependency occurs if a speculating process affects other processes. For example, if a speculating process $A$ makes its data modifications visible to another process $B$, then $B$ will have to roll back if $A$ detects a data conflict. There are two approaches to deal with this problem. One way is to allow the above *speculation dependencies*, monitor them, and rollback dependent processes along with the misspeculating process. Alternatively, we may prevent them by postponing the operations that cause such a dependency until the process leaves speculative mode.

If this is impossible due to the specifics of the target system, a speculating process that wishes to perform a dependency inducing operation should block waiting until the lock is properly acquired.

The method to deal with speculative dependencies should be chosen based on the fault-tolerance scheme as well as on the application properties. For example, if data conflicts are rare and the number of processes affected by a single misspeculation is relatively small, then speculation dependencies may be allowed, monitored and treated accordingly. In contrast, the prevention approach seems more suitable when the cost of cascading rollbacks is relatively high. Although prevention may yield better
6.1. SPECULATIVE LOCK ACQUISITION

performance, it requires modifying the original runtime logic in order to delay dependency-inducing events.

6.1.5 Speculative Deadlocks

In many systems, transferring lock ownership to a remote process while in speculative mode results in a rollback dependency described in Section 6.1.4. Hence, in order to prevent speculation dependencies in these systems, the lock transfer operation is delayed until the process leaves the speculative mode. However, since a process leaves speculative mode only after receiving ownership of all the locks it has speculatively acquired, a system-level deadlock may occur, as described below.

Henceforth, we will say that a process holds a lock if it is the lock owner and it is inside a critical section protected by this lock. A process that owns a lock is allowed to acquire and release it but does not necessarily hold it. On the application level, the lock owner may transfer lock ownership to another process only if it (the owner) does not hold the lock.

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. Assuming the application is deadlock-free, at some point at least one of the processes will stop holding the lock speculatively acquired by the other process. However, neither process can transfer lock ownership to the other process, because they are both in speculative mode. Figure 6.2 illustrates this situation. In general, any number of processes can be involved in a circular dependency analogous to this, which we refer to as speculative deadlock.

Unless a speculative deadlock is treated, a process can remain stuck in the speculative mode. There are several ways to detect and resolve speculative deadlocks. The simplest method is based on timeouts. Upon speculative acquisition, 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. 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 processes with the smallest ID in the loop is rolled back.
6.1.6 Speculation Preclusion

Under certain data access patterns speculative lock acquisition is likely to result in a rollback and therefore can have a negative impact on performance. We employ a number of run time heuristics to detect such situations. When detected, the system acquires the lock in a non-speculative (blocking) fashion.

To avoid repetitive misspeculations, we monitor the rollbacks associated with a particular lock or a critical section. Based on these statistics, in each given case, the process decides whether it should perform a speculative acquisition. A possible policy is to prevent speculative acquisition of a 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 period of time or and in a number of next acquisition operations. Since the speculation preclusion logic may be invoked on each lock acquisition, it should be as lightweight as possible.

Application processes that frequently acquire a large number of locks may spend most of their execution time in speculative mode, rarely exiting it. This has two negative effects. First, such processes are likely to become involved in a speculative deadlock. Second, they may rarely be able to create a recoverable state, because, as described in Section 6.1.2, in certain fault-tolerance schemes processes cannot checkpoint while speculating. Consequently, processes may lose a significant portion of their computation in case of a rollback.
6.2 Speculative Locking in JavaSplit

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

6.2.1 Implementing Data Conflict Detection

The implementation of data conflict detection necessary to support speculative locking consists of monitoring local read accesses, monitoring remote write accesses and identifying conflicts between the two access types.

Monitoring Local Read Accesses

In order to monitor the data read 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 have been speculatively read.

Figure 6.3 presents the inserted logic in pseudo-code. If the current thread is not speculating, 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 resetting 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.

**Monitoring Remote Write Accesses**

The memory model employed in JavaSplit is *Lazy Release Consistency* (LRC) [55]. In the implementations of LRC, including the JavaSplit protocol, 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 (memory pages or objects) and then fetches their updated versions the next time it accesses them.

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 speculating. It follows that the write notices contain information about all the remote write accesses that may conflict with the read operations performed in speculative mode.

**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 JavaSplit memory management protocol, a write notice consists from 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

---

Figure 6.3: Pseudo-code inserted before a read access

IF in speculative mode
  IF object is not marked speculatively read
    MARK object speculatively read
    ADD object to the list of speculatively read objects
  ENDIF
ENDIF
is not invalidated based on a write notice, it means that there were no conflicting remote write accesses. Consequently, a data conflict may occur only for invalidated objects.

If an object marked as speculatively read is invalidated based on a received write notice, the current thread rolls back. Note that despite a decision to roll back there is a possibility that there was no actual data conflict. A remote process that performed the write access and the local process might have accessed different fields of the same object. In general, the frequency of false positives depends on the granularity of the memory coherency unit and the access pattern.

6.2.2 Implementing Consistent Rollback

In JavaSplit, fault-tolerance support, described in Chapter 5 is accomplished by checkpointing and replication. The states of the application threads are captured in such a way as to guarantee that rollback of any single thread to its latest saved checkpoint does not violate system consistency. Consequently, JavaSplit keeps only a single checkpoint per thread. Process recovery never results in rollback of non-failing processes. This allows efficient implementation of local rollback and therefore makes JavaSplit well-suited for accommodating speculative locking. It should be noted that although the number of concurrent node failures tolerated by JavaSplit depends on the number of maintained thread snapshot replicas, the number of concurrent process rollbacks is unlimited.

Ensuring Recoverable State Before Speculation

As described above, JavaSplit checkpointing scheme guarantees that the most recent process checkpoint constitutes a recoverable state, as defined in Section 6.1.2. According to the same section, it remains to ensure that a process is not checkpointed while in speculative mode. Otherwise, it will not be able to roll back to a state preceding the speculation in case of a data conflict.

In JavaSplit, a process checkpoints itself only when it transfers lock ownership to another process. Thus, if the process transfers lock ownership while in speculative mode, it will not be able to roll back to the snapshot preceding the speculation. Hence, in order to maintain a recoverable state before the beginning of the speculative mode, we delay ownership transfer until the speculation ends. Coincidentally, this also prevents a speculating process from affecting other processes, because a process exposes its modifications only when it passes lock ownership to another process. Thus, by avoiding ownership transfer in speculative mode, we not only maintain a recoverable state for the
CHAPTER 6. SPECULATIVE LOCKING

case of misSpeculation but also prevent speculative dependencies described in Section 6.1.4. Those dependencies are undesirable, because a rollback of one process may require other processes to rollback to maintain consistency.

The inability of a speculating process to transfer lock ownership results in the speculative dead-lock phenomenon presented in Section 6.1.5. To resolve speculative deadlock we have implemented a timeout-based deadlock discovery described in the same section.

JavaSplit fault-tolerance scheme does not prohibit taking checkpoints which are not induced by lock transfer operations. This allows to increase checkpointing frequency in order to minimize the amount of computation lost due to data conflicts, as suggested in Section 6.1.2.

Efficient Local Rollback

The JavaSplit process recovery procedure is sufficiently lightweight to be employed as is in the implementation of the local rollback operation. Nevertheless, we follow the optimization guidelines presented in Section 6.1.3 in order to reduce the rollback overhead. First, each process locally stores its latest snapshot and therefore a rollback does not require fetching the snapshot from a remote location. Second, certain system data structures, such as the lock management logic are not reconstructed from scratch but rather updated during rollback. Finally, unlike in 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.

6.2.3 Conditional Waiting for Speculative Locking

The Java programming language provides conditional waiting primitives wait and notify. A wait set is associated with each lock. A thread that performs the the wait operation is inserted into the wait set of a lock, releases the lock and blocks. Invocation of the notify operation removes an arbitrary thread from the lock’s wait set. The notified thread continues its execution but must re-acquire the lock in order to proceed. In order to perform the above operations a thread must be the owner of the object’s lock.

In JavaSplit, the wait set is passed along with the lock ownership. Consequently, the owner of the lock can always access the wait set locally. However, if locks are acquired speculatively, the wait set 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 set and then applying the logged operations to the wait set when it is received. The semantics of conditional waiting operations permit delaying their effect until the lock is released.

Multithreaded Java programs tend to encapsulate the wait operation in conditional loops. An example of this programming pattern is shown in Figure 6.4. 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.

6.2.4 Correctness Proof

In this section, we provide a correctness proof for speculative locking in JavaSplit. The proof is coupled to the memory consistency model implemented in JavaSplit. 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.

**Lemma 6.1** 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 local and shared objects and acquiring 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 the speculation period. 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 which does not violate the happens-before order.

**Lemma 6.2** 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 Lemma 6.1, after the process leaves speculative mode for the
6.2. SPECULATIVE LOCKING IN JAVASPLIT

(i + 1)-th time, it is in a state which does not violate the happens-before order. Thus, by induction, the claim is correct. ■

Lemma 6.3 When the process completes execution, it is in a state which 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, 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 Lemma 6.2, 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. ■

Lemma 6.4 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. ■

Lemma 6.5 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 6.4, 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 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., transferring locks to the requesting processes. Handling pending requests ensures that no dependencies on this process exist, breaking the dependency loop and ensuring forward progress.

\[\text{\textbullet} \]

6.3 Performance Evaluation

6.3.1 Evaluation Goals

This section 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.

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 ranges between 0.2ms and 0.6ms.
6.3. PERFORMANCE EVALUATION

... public void run() {
    while(...) {
        DataObject dataObject = selectRandomObject();
        synchronized(dataObject) {
            dataObject.read();
            if (shouldWrite()) {
                dataObject.write();
            }
            compute(T1);
        }
        compute(T2);
    }
}

Figure 6.5: Model application - worker thread code

6.3.2 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 6.5. 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:
CHAPTER 6. SPECULATIVE LOCKING

$L$ Number of shared objects and their associated locks
$W$ Number of worker threads
$P_{\text{write}}$ Write probability per loop iteration
$T_{\text{chk}}$ 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)
$T_1$ compute(T1) execution time with a bare JVM
$T_{1,JS}$ compute(T1) execution time with JS bytecode instrumentation
$T_{1,SL}$ compute(T1) execution time with JS and SL bytecode instrumentation
$Q$ Average lock request queue size

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 calls to 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 $Q \cdot T_1$. Overall, the time required per loop iteration is $Q \cdot T_1 + T_1 + T_2$.

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

$$F_{\text{bare}} = \frac{W}{Q \cdot T_1 + T_1 + T_2} \quad (6.1)$$

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:

$$Q = \max\left(0, \frac{W}{L} \cdot \frac{T_2}{T_1}\right) \quad (6.2)$$
6.3. PERFORMANCE EVALUATION

Throughput on JavaSplit without Speculation

When the application is distributed using JavaSplit, the time breakdown of each loop iteration is as follows. The thread sends a message to the lock owner ($T_{net}$ time units). Then, it waits until the lock passes through all $Q$ requesters in the lock queue. Before the lock ownership is obtained, 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 6.6 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). These requests arrive at an average rate of one request per worker per iteration.

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

$$F_{JS} = \frac{W}{2 \cdot (T_{net} + T_{chk}) + Q \cdot (T_{net} + T_{chk} + T_{1,JS}) + T_{1,JS} + T_{2,JS}}$$ (6.3)

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.
Figure 6.6: Flow of a queued lock acquisition

The second factor is the overhead of speculation failure, either due to a data conflict or due to a speculative deadlock. In both cases, the failure overhead is equal to the rollback time ($T_{roll}$ time units) and the lost application time ($S * (T_{1,SL} + T_{2,SL})$ time units). We annotate the probability for any speculation failure by $P_{fail}$, which consists of the probability of a conflict ($P_{conflict}$) and the probability for a deadlock ($P_{deadlock}$).

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

$$S/F_{SL} = T_{compute} + (T_{chk} + T_{net}) + P_{fail} * (T_{roll} + T_{compute})$$  \hspace{1cm} (6.4)

And therefore, the throughput of speculative distributed JavaSplit is:

$$F_{SL} = \frac{W}{T_{1,SL} + T_{2,SL} + (T_{chk} + T_{net})/S + P_{fail} * (T_{roll}/S + T_{1,SL} + T_{2,SL})}$$  \hspace{1cm} (6.5)

**Data Conflict Probability**  The probability for a data conflict ($P_{conflict}$) depends on multiple factors.
6.3. PERFORMANCE EVALUATION

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. Amongst 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
CHAPTER 6. SPECULATIVE LOCKING

beneficial, but also increases the probability for a speculative deadlock, which reduces overall performance. For each application execution, an optimal value for $M$ exists, denoted by $M_{max}$.

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 (low $P_{conflict}$). 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 (low $P_{deadlock}$). 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 $Q \rightarrow W$ 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 $T_{1,SL}$, $T_{2,SL}$).
6.3. PERFORMANCE EVALUATION

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

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 $M_{\text{max}}$.

The round-trip delay effects the speedup gained by speculative locking significantly. First, it reduces the throughput of non-speculative systems, because as $T_{\text{net}}$ 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.

6.3.3 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.

SPECjbb

We have implemented a minimal version of the SPECjbb2000 [5] 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 serves as a good test case for evaluating how speculative locking handles applications which use fine-grained locking with thousands of objects and locks.
CHAPTER 6. SPECULATIVE LOCKING

**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 6.7 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.

Correlating the results to the theoretical analysis, this execution has $Q = 0$ and therefore speculative locking only reduces the lock acquisition overhead ($T_{net} + T_{chk}$) which is associated with every lock fetch. Additionally, this execution has $T_2 >> T_1$, reducing the probability for speculative deadlocks ($S = 1$).

**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 6.3.4 performs
6.3. PERFORMANCE EVALUATION

Figure 6.7: Throughput and worker breakdown for SPECJBB* (low acq. frequency)
Figure 6.8: Throughput and worker breakdown for SPECJBB* (high acq. frequency)
6.3. PERFORMANCE EVALUATION

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 6.8 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 increase, 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.

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

![Graph showing throughput versus workers]

Figure 6.10: Hashtable throughput

**Hashtable**

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 Hashtable 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 6.3.2).
Figure 6.10 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 limited by \( 1/T_1 \).

### 6.3.4 Micro-Benchmark Results

**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 6.11 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 message and the time required to distribute the checkpoint in the process which transfers lock ownership.
6.3. PERFORMANCE EVALUATION

![Graph showing lock acquisition latency with varying network latencies]

Figure 6.11: Average lock acquisition latency with varying network latencies

**Data Conflicts**

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

Figure 6.12 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 recovering from speculation failures, reducing the overall throughput proportionally.

The time spent by each worker to handle speculation failures is the product of the number of data conflicts by the time lost per data conflict. Therefore, the "Speculation Failure Time" line is a linear function of the data conflict probability. Consequently, the throughput decreases linearly as the data conflict probability increases.

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
Figure 6.12: SPECJBB* throughput with varying data conflict probabilities

Figure 6.13: Varying preemptive checkpointing frequency
rollback time and the lost application time.

Figure 6.13 shows the time spent on checkpointing and on recovering from failures for a micro-benchmark with some non-negligible data conflict probability ($P_{\text{conflict}} = 0.5$). 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.

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) [15]. We compared 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 6.1 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
Table 6.1: 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>13024360.00</td>
<td>35686080.00</td>
<td>35.60</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Same: Scalar: Class</td>
<td>13304760.00</td>
<td>78916248.00</td>
<td>40.60</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Same: Array: Local</td>
<td>88210512.00</td>
<td>60789680.00</td>
<td>20.88</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Same: Array: Instance</td>
<td>2632729.00</td>
<td>1959246.10</td>
<td>24.74</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Same: Array: Class</td>
<td>4927519.00</td>
<td>3815912.00</td>
<td>22.56</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Other: Scalar: Instance</td>
<td>10151696.00</td>
<td>73652504.00</td>
<td>27.47</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>1: Assign: Other: Scalar: Class</td>
<td>13305451.00</td>
<td>78916248.00</td>
<td>40.60</td>
<td>(assignments/s)</td>
</tr>
<tr>
<td>2: LUFact: Kernel: SizeA</td>
<td>52.070393</td>
<td>40.03502</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: Cyclic: Kernel: SizeA</td>
<td>35556.6094</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>134089.0</td>
<td>13.58</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>2: SparseMatMult: 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: Cyclic: Kernel: SizeB</td>
<td>3667.7058</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>15.36</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>2: SOR: Kernel: SizeB</td>
<td>4.356919</td>
<td>3.1093062</td>
<td>28.62</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: SparseMatMult: Kernel: SizeB</td>
<td>11.8845935</td>
<td>10.466272</td>
<td>11.94</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>2: Cyclic: 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: SparseMatMult: Kernel: SizeC</td>
<td>1.495171</td>
<td>1.3734094</td>
<td>8.16</td>
<td>(Iterations/s)</td>
</tr>
<tr>
<td>3: Euler: Initial: 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>0.30695935</td>
<td>0.22281292</td>
<td>39.27</td>
<td>(Timesteps/s)</td>
</tr>
<tr>
<td>3: Euler: Total: SizeA</td>
<td>0.003650928</td>
<td>0.00219909</td>
<td>39.22</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MolDyn: Run: SizeA</td>
<td>19590.025</td>
<td>15146.326</td>
<td>24.08</td>
<td>(Interactions/s)</td>
</tr>
<tr>
<td>3: MolDyn: Total: SizeA</td>
<td>0.01581027</td>
<td>0.01198639</td>
<td>24.22</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MonteCarlo: Run: SizeA</td>
<td>890.1549</td>
<td>624.3756</td>
<td>29.86</td>
<td>(Samples/s)</td>
</tr>
<tr>
<td>3: MonteCarlo: Total: SizeA</td>
<td>0.07833912</td>
<td>0.054842602</td>
<td>29.99</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: RayTracer: Run: SizeA</td>
<td>70692.947</td>
<td>450992.2</td>
<td>36.20</td>
<td>(Pixels/s)</td>
</tr>
<tr>
<td>3: RayTracer: Total: SizeA</td>
<td>0.028572245</td>
<td>0.01946508</td>
<td>31.87</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: AlphaBetaSearch: Run: SizeA</td>
<td>103456.13</td>
<td>90685.1</td>
<td>12.18</td>
<td>(positions/s)</td>
</tr>
<tr>
<td>3: Euler: Initial: SizeB</td>
<td>38539.73</td>
<td>26561.166</td>
<td>31.08</td>
<td>(Gridpoints/s)</td>
</tr>
<tr>
<td>3: Euler: Run: SizeB</td>
<td>0.16012631</td>
<td>0.09180715</td>
<td>42.61</td>
<td>(Timesteps/s)</td>
</tr>
<tr>
<td>3: Euler: Total: SizeB</td>
<td>0.00155315</td>
<td>9.17E-04</td>
<td>45.22</td>
<td>(Solutions/s)</td>
</tr>
<tr>
<td>3: MolDyn: Run: SizeB</td>
<td>231.01254</td>
<td>18562.988</td>
<td>91.65</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>
6.3. PERFORMANCE EVALUATION

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 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 8.1 we suggest several methods to reduce this overhead.

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 6.14 shows the probability for speculative deadlocks as a function of the speculation quota, when running SPECjbb* 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.

Section 6.3.2 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 6.15 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.
CHAPTER 6. SPECULATIVE LOCKING

Figure 6.14: Speculative deadlock probability as a function of speculation quota

Figure 6.15: SPECjbb* throughput as a function of speculation quota
6.3.5 Performance Evaluation Summary

Performance evaluation confirms that speculative locking may indeed boost applications which were previously executed on 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 which cause data conflicts may not be suitable for execution with speculative locking. The proposed scheme employs several heuristics to detect these situations and avoid speculating on conflicting locks. In section 8.8 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 8.7 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 8.1 suggests methods of reducing this overhead.
Chapter 7

Transactional Memory vs.
Speculative Locking

Employment of lock-based synchronization in parallel applications imposes a trade-off between programming complexity and performance. Fine-grained locking provides a higher level of concurrency but significantly increases the programming effort. In contrast, parallel programming using coarse-grained locking is straightforward. However, it cannot match the concurrency and scalability of its fine-grained counterpart.

On shared-memory multiprocessors, transactional memory is a well established alternative to lock-based synchronization. While providing a simple synchronization paradigm it can achieve performance close to that of fine-grained locking. Transactional programming replaces lock-protected critical sections with atomic regions which are executed transactionally. If a data conflict with a concurrent transaction is detected, the transaction is aborted and possibly retried. There have been a great number of works that describe hardware [39, 49, 93] and software [40, 48, 90, 71], implementations of transactional memory.

Several recent works [13, 50, 60, 70] suggest employment of transactional memory in a distributed environment. The proposed schemes appear to be drawn from the classic hardware/software transactional memory protocols in which each transaction has to be validated before it terminates. In a distributed setting, this approach results in placement of blocking remote operations, whose purpose
is to avoid/detect conflicts, on the execution critical path. Due to the relatively high network latency, the application threads spend a large portion of their time waiting for a response from remote peers. (The execution time breakdown figures in [50] clearly show this phenomenon.) Overall, it seems that these schemes are not "optimistic" enough, trying to ensure validity of one transaction before executing another. In addition to the above problem, some schemes employ centralized and broadcast-based paradigms, which significantly hinders their scalability.

There is a certain similarity between speculative locking and transactional memory. Similar to transactional memory, speculative locking reduces the programming effort without sacrificing concurrency. It allows programmer to employ coarse-grained locking while achieving fine-grained parallelism under the hood. 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.

In this chapter, we attempt to determine whether speculative locking can be a suitable alternative to transactional programming in a distributed setting. We compare its characteristics of the distributed speculative locking scheme presented in Chapter 6 to those of transactional memory systems. We find that despite conceptual similarity to transactional memory, speculative locking is capable of overlapping computation with communication and by no means requires a centralized or broadcast-based implementation. Nevertheless, acknowledging the benefits of transactional programming, we propose a simple implementation of transactional memory on top of our speculative locking protocol. The fact that this implementation does not suffer from the aforementioned shortcomings suggests that these problems are not inherent to transactional memory and that a more efficient implementation may be possible.

The structure of the rest of this chapter is as follows. Section 7.1 discusses prior work on distributed transactional memory. Section 7.2 compares transactional memory with speculative locking in a distributed setting and discusses the tradeoffs in their employment. Section 7.3 proposes a distributed implementation of transactional memory on top of speculative locking.
7.1 Distributed Transactional Memory Protocols

The main focus of transactional memory research has been shared-memory multiprocessors. Recently, the community has started showing interest in transactional memory on a cluster of interconnected processors. In the absence of physical shared memory and built-in cache coherence protocols, the distributed transactional memory systems synchronize access to shared data using message passing and remote procedure calls.

Herlihy et al. [50] describe a distributed scheme for metric-space networks. The main concern of this work is prompt object discovery and efficient routing of remote requests. The nodes are organized in a tree-like structure whose edges serve as communication paths. The distributed memory management logic mimics the operation of a multiprocessor transaction-aware cache coherence protocol. Before reading or writing a shared object for the first time, a transaction must open it. The open operation invalidates (or downgrades) remote copies of the object and ensures that the local copy is valid, possibly fetching an update from another node. Creating a local writable copy invalidates all remote copies; creating a local read-only copy downgrades the remote writable copy to a read-only copy. A transaction commits only if none of the objects that it accessed have been invalidated by remote transactions. While commit is local, the open operation often results in blocking remote requests, whose purpose is to invalidate/downgrade remote copies and/or fetch the most recent copy. Although the actual frequency of remote open operations depends on the data access pattern, it is expected to be considerably higher than the rate of data misses in a distributed shared memory system that employs a relaxed consistency model, e.g., Lazy Release Consistency (LRC) [114]. (Informally, this is due to the fact transactional memory is "strict" than LRC).

Distributed Multiversioning (DMV) [70] is a broadcast-based cache coherence scheme which distinguishes between update and read-only transactions. The key optimization in this scheme is isolation of read-only transactions from any remote conflicting updates. A read-only transaction creates the needed version of a page by selectively applying previously received incremental updates to the base version of the page. Thus, DMV manages to avoid a certain amount of read-write conflicts. Upon commit, each update transaction first creates a new memory version number and then broadcasts per-page version-tagged modifications to all other replicas. If the modification broadcast is received during a conflicting update transaction, the latter is aborted. The DMV protocol requires
that only one update transaction performs a modification broadcast at any given time. This is achieved by either obtaining a system-wide token (update-anywhere variant), or by executing all the update transactions on a single dedicated node (master-update variant). During commit, the update transaction broadcasts its modifications to all nodes and waits for their acknowledgements before proceeding. This makes commit a blocking remote operation which hinders throughput. In the master-anywhere variant, the token acquisition prior to commit introduces yet another blocking remote request. The scalability of the master-update variant is doubtful due to the fact that the master node must execute all update transactions.

Distributed Software Transactional Memory (DiSTM) [60] is built on top of the DSTM2 [46] engine. This work evaluates three broadcast-based transactional memory coherence protocols: (i) TCC, (ii) serialization lease and (iii) multiple leases. All proposed protocols detect conflicts at commit time. A committing transaction sends its updates to a system-global master node. The latter updates the global dataset it maintains and then forwards the modifications to the nodes that have cached copies of the modified objects. In TCC, each transaction that wishes to commit, broadcasts its read and write sets so that the other nodes determine whether there have been any conflicts. In the two lease-based protocols a transaction that wishes to commit must acquire a lease from the master node, which performs preliminary conflict detection. In the serialization lease protocol, only one transaction is allowed to commit, whereas in the multiple-leases protocol several transactions can commit concurrently. The above protocols place two blocking remote requests on the execution critical path. One is the pre-commit action: broadcast of read/write sets or lease acquisition. The second is the transmission of updates. The employment of master-centric approach hinders scalability, as confirmed by the presented results.

Bocchino et al [13] attempt to create a highly scalable distributed transactional memory that provides weak atomicity [73]. They identify eight dimensions in the software transactional memory design space and evaluate several design choices that they find suitable for a distributed environment. In the implemented protocol variants either the open or the commit operations are implemented as remote requests located on the execution critical path. The reported nearly linear scalability to hundreds of processors is probably due to the excessive computation/communication ratio. This speculation is confirmed by the fact that in the presented experiments a benchmark that uses distributed queue locks achieves a similar level of scalability.
7.2. TRANSACTIONAL MEMORY VS. SPECULATIVE LOCKING

In the above schemes, a transaction coordinates its accesses to the shared data with remote peers either during execution of the *open* operation [13, 50] or during commit [13, 70, 60]. In both cases, a transaction sends at least one remote request and waits for a reply. For ease of reference, we classify these schemes as *blocking-open* and *blocking-commit*, respectively. While in the latter the commit operation always induces a blocking remote request, the open operation in the former may be local, provided that the current node has a valid cached copy of the object. However, due to the transactional "strict" consistency model cached copies are often invalidated.

Clearly, in the above distributed transactional memory systems, henceforth *DTMs*, a thread spends a non-negligible portion of its time waiting for replies from remote nodes, which decreases the overall throughput of the system. The commit operation is performed once per transaction, while the open operations is performed each time a transaction accesses an object for the first time. The resulting remote requests are placed on the execution critical path due to a design that requires validating a transactions before it commits. In contrast, the distributed speculative locking scheme we present below is capable of detecting conflicts in parallel to the computation. In addition, unlike some of the above schemes, it does not employ centralized and broadcast-based techniques.

7.2 Transactional Memory vs. Speculative Locking

Transactional memory and speculative locking have a lot in common. They are both optimistic concurrency control mechanisms whose primary goal is to achieve a high level of concurrency with minimal programming efforts. They share the same basic idea of optimistic execution, followed by conflict detection and rollback in case validation fails. The implementation of both methods requires monitoring accesses to shared memory as well as buffering (or logging) modifications until they can be safely deposited.

The two methods support different programming constructs whose semantics affect implementation freedom. Transactional memory provides the *atomic* block that should appear to be executed atomically. If there is a data conflict between two concurrent transactions, one of them must be aborted. Consequently, a committing transaction must be validated against each and every transaction in the system. Speculative locking systems, however, employ the classic lock-based synchronization, i.e., they use lock-protected critical sections, which impose a relaxed ordering on the memory
CHAPTER 7. TRANSACTIONAL MEMORY VS. SPECULATIVE LOCKING

accesses.

Informally, transactional memory model appears to be "strict" than the relaxed consistency models employed under speculative locking. Consequently, its implementation potentially requires more inter-thread cooperation, which has a considerable negative performance impact in a distributed setting. This impact is not only due to remote requests on the critical path, but also due to relatively high overhead of network operations and backlog effects. Nevertheless, it has been shown [35, 36] that speculative approaches can significantly reduce the gap between the stricter and the more relaxed consistency models. Thus, it may be possible for a more speculative (optimistic) implementation of distributed transactional memory to yield much better performance.

The proposed distributed speculative locking scheme takes advantage of the underlying relaxed consistency model and, unlike DTM s discussed in Section 7.1, manages to mask communication with computation. The question is whether lock-based synchronized with speculative locking support is a better programming model for a distributed environment than transactional memory. In the remainder of this section, we address this question by describing the pros in cons of the two paradigms. Note that many of the arguments below are not directly related to a distributed setting.

Transactional memory has a number of programming advantages over lock-based synchronization. First, unlike locks, transactions are composable into larger atomic operations [41]. Second, its memory model is more comprehensive than relaxed memory models implemented in lock-based systems. Third, the transactional nature of the execution enables straightforward implementations of fault-tolerance schemes. Finally, the fact that atomic blocks are not associated with a particular object relieves the programmer from the need to choose a lock variable.

Most shortcomings of transactional memory are performance-related. As already mentioned, transactional memory model is hard to implement efficiently, especially in the absence of physical shared memory. Moreover, although transactions are commonly intended to be short, they are optimistic only within their boundaries. An additional downside is the fact that certain TM implementations prohibit data prefetching, because it may significantly increase data conflict probability. In a distributed environment the gravity of this shortcoming increases, as data prefetching is an important optimization in this setting.

Proper handling of I/O operations is a well known problem of transactional memory systems. Since normally I/O operations cannot be undone, they cannot be placed inside a transaction or must
7.2. TRANSACTIONAL MEMORY VS. SPECULATIVE LOCKING

Table 7.1: Pros and cons summary

<table>
<thead>
<tr>
<th></th>
<th>Speculative Locking</th>
<th>Transactional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pros</td>
<td>Can use relaxed memory models</td>
<td>More comprehensive memory model</td>
</tr>
<tr>
<td></td>
<td>Multiple coarse-grained locks</td>
<td>Transactions are composable</td>
</tr>
<tr>
<td></td>
<td>Limited conflict-sensitive period</td>
<td>Straightforward fault-tolerance</td>
</tr>
<tr>
<td></td>
<td>Explicit fine-grained locks support</td>
<td>Atomic blocks are not annotated</td>
</tr>
<tr>
<td></td>
<td>Legacy apps are lock-based</td>
<td></td>
</tr>
<tr>
<td>Cons</td>
<td>Application deadlocks</td>
<td>&quot;Strict&quot; consistency hinders efficiency</td>
</tr>
<tr>
<td></td>
<td>Speculative deadlocks</td>
<td>Optimistic only within transaction boundaries</td>
</tr>
<tr>
<td></td>
<td>No syntactic lock-object association</td>
<td>Data prefetching may cause conflicts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Problematic I/O support</td>
</tr>
</tbody>
</table>

be deferred until a transaction is committed. This problem is especially acute in systems that do not support non-transactional execution. A similar problem exists also in lock-based schemes with speculation support: an I/O operation cannot be executed while in speculative mode. However, there is also a simple solution. Before executing an I/O operation, a thread waits until it leaves the speculative mode. In the absence of lock contention, the thread will block for a period of time roughly equal to the message round trip latency, which is acceptable considering the overhead of the I/O operation itself.

The speculative locking approach has a number of efficiency advantages, most of which have been already mentioned. The fact that it can be used in conjunction with relaxed memory schemes yields more freedom for optimizations. Employment of multiple coarse-grained locks over relaxed memory models results in less ordering constraints between memory operations which further reduces conflict probability. Additional advantage is the fact that conflict-sensitive period may be limited to the beginning of a (long) critical section. This period spans from the beginning of a critical section and until the lock ownership is received. Consequently even if the lock ownership message invalidates an object $X$, there will be no conflict if $X$ is accessed after the lock ownership is received. In contrast, in lazy validation [13] transactional schemes all accesses of one transaction are validated against all accesses of another.

An important argument in favor of speculative locking is the fact that most legacy applications are lock-based. While there have been attempts [17] to implement lock-based synchronization over transactional memory, we are not aware of any entirely transparent implementations. The thorniest
problem seems to be the implementation of conditional waiting.

The downsides of speculative locking approach are application and speculation deadlocks. Application deadlocks are unlikely if coarse-grained locking is employed. Speculative deadlocks can be promptly resolved, but they still have a performance penalty because one of the involved threads must roll back.

Table 7.1 summarizes the above discussion. We believe that speculative locking is a suitable alternative for transactional programming. Keeping in mind the benefits of the latter, we suggest that the former is at least considered when designing a distributed runtime environment. The validity of this suggestion is confirmed by the "As Good As Fine Grained" postulate that appears in several presentations of Nir Shavit, one of the forefathers of software transactional memory [90]:

"If we could implement fine-grained locking with the same simplicity of coarse-grained, we would never think of building a transactional memory"

7.3 Implementing DTM on top of Speculative Locking

The benefits of transactional programming may make it the preferred choice for writing parallel applications in a distributed environment, despite the potential performance advantages of speculative locking. Hence, we sketch a simple implementation of transactional memory on top of distributed speculative locking.

The main idea of the proposed implementation is to translate the transactional atomic blocks to lock-protected critical sections. The transactions are partitioned into the maximal number of disjoint sets such that two transactions belonging to different sets are guaranteed not to conflict. The partitioning is performed based on static analysis of the code. A dedicated lock is assigned to each transaction set. An atomic block is translated to a critical section protected by the lock assigned to the set of the corresponding transaction. In the worst case, there is a single transactions set, which results in a system-global lock protecting all the critical sections. Note that in this case, however, there are no speculative deadlocks.

The above transformation may be performed by the compiler or in a post-compilation step. It can also be implemented as a source-level pre-processing phase followed by compilation for the lock-based environment. The resulting code is to be executed on a distributed lazy release consistent runtime system with speculative locking support, e.g., JavaSplit. We recommend eager propagation
of updates during lock ownership transfer, in order to reduce the number of data misses.

To illustrate the potential performance advantage of the proposed scheme, henceforth $DTM/SL$, over the blocking-open and blocking-commit DTMs (defined in Section 7.1), we consider the following TM scenario. Multiple threads, each running on a separate node, concurrently access a large number of shared objects. Each object is accessed within the boundaries of a transaction. We assume that true data collisions are rare, which is the case in many TM benchmarks.

Figure 7.1 presents the execution of a write accesses (by thread $B$) in blocking-commit DTM, blocking-open DTM and $DTM/SL$. In blocking-commit DTM, there is (at least one) blocking request in the end of the transaction for the purposes of validation and/or update distribution. In blocking-open DTM, there is a remote open request in order to validate against the preceding read operation in thread $A$. In $DTM/SL$ the write operation is local, assuming that it has a valid copy of $X$. This is likely to be the case due to a number of reasons. First, since write accesses are infrequent it is likely that $B$ has an up-to-date copy of $X$ if it accessed $X$ in the past. Second, if eager update scheme is employed (as shown in the figure), $B$ will receive the most updated copy of $X$ along with the lock ownership while executing a preceding critical section. Finally, if a data prefetching scheme is employed $B$ may get the most updated copy of $X$ when fetching another object.

The execution of a read access can be presented in a similar fashion, by switching between read($x$)
and write\( (x) \) operations in Figure 7.1. The blocking-commit DTM will perform a blocking request in any case. However, the blocking-open DTM is guaranteed to perform a remote operation only if it has not accessed \( X \) after the thread \( A \) has written to it (and before the current read operation). The DTM/SL will avoid a read miss using eager update propagation or data prefetching.

In the above scenario, DTM/SL is guaranteed to perform less blocking remote operations than the blocking-commit scheme and is likely to perform less blocking remote operations than the blocking-open scheme. Although far from a formal proof, we feel this scenario makes a good case in favor of both DTM/SL and distributed speculative locking in general.

7.4 Conclusion

The existing DTM schemes seem to be drawn from the design space of traditional software and hardware transactional memory protocols. Some sacrifice scalability by adapting centralized and broadcast-based patterns. The others appear to be not optimistic enough to remove blocking remote (conflict preventing) operations from the critical path and hence achieve suboptimal throughput.

The performance issues discussed in this chapter are not necessarily inherent to implementation of distributed transactional memory. This is confirmed by the fact that DTM implementation on top of speculating locking is expected to yield better throughput than existing DTM schemes. Nevertheless, we feel that complexity of a potential DTM implementation that alleviates the above issues (and does not employ SL) is significantly higher than that of the existing distributed systems.

Overall, speculative locking appears to be a good choice for optimistic concurrency control in a distributed environment. Employed over a relaxed consistency scheme, it does not only achieve the concurrency of fine-grain locking but also manages to overlap communication with computation.
Chapter 8

Future Work

This chapter describes a number of extensions that may improve the performance of the JavaSplit framework but have been beyond the scope of this work. It also presents several possible future research directions related to this work.

8.1 Instrumentation Optimizations

A significant portion of the instrumentation overhead in JavaSplit is due to access checks preceding access to object contents. It is possible to significantly reduce this overhead as described below.

Currently, an access check is added before each access to object field (or array element). A read access check that is placed before a read access ensures that object contents are valid. A write access check that is placed before a write access ensures that the object contents are valid; it also creates an object twin if it has not been created yet (in the current interval). A large portion of read access checks can be removed as follows. A read access check preceded by a read or write access to the same object can be removed, if there is no lock acquisition between the two operations. This is due to the fact that object contents may be invalidated only by write notices received during lock acquisition.

When speculative locking is employed, additional read access checks monitor objects that are read while the thread is in speculative mode. If there is no lock acquisition between two consecutive access checks of the same object, the latter access check can be removed. This is due to the fact that a thread cannot enter speculative mode between these operations.
Since Java programs usually read a number of object fields (or array elements) between lock acquisitions, a large portion of access checks can be eliminated by means of static analysis. A vivid example of access check elimination is related to code loops. Read access checks inside a loop without lock acquisitions can be placed before the loop. Thus, an access check is performed once per loop rather than in each iteration.

Another optimization related to speculative locking is creation of 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.

8.2 Array Partitioning

JavaSplit memory management protocol treats arrays as regular objects. A modification of an array element by one thread often results in invalidation of the array in another thread. Consequently, the latter thread must fetch the entire array contents from its home, even if it accesses a different portion of the array. Similarly, in the context of speculative locking, a write access in one thread often results in a data conflict in another thread, even if the latter accesses a different portion of the array. Since, unlike regular Java objects, array can be large and accesses to its different portions may be unrelated one to the other, it seems that managing portions of a large array as different consistency units may have a significant positive impact on performance. Although the instrumentation required to implement such array consistency partitioning may induce additional overhead on array access, many applications will still benefit from this optimization.

8.3 Single-Writer Consistency Protocols

JavaSplit employs a multiple-writer scheme which allows threads concurrently write to the same object. Naturally, if the application is race-free, the threads modify different portions of an object.
8.4. PASSING DATA MODIFICATIONS INSTEAD OF WRITE NOTICES

However, small objects are usually modified by one thread at a time, while the access is protected by a lock. Under this access pattern, single-writer schemes should outperform the employed multiple-writer scheme. A single-writer scheme would avoid costly memory synchronization operation in the end of each interval and would prevent access misses in the current exclusive writer. The impact of a single-writer scheme on the implemented fault-tolerance protocol and support for speculative locking is unclear.

8.4 Passing Data Modifications Instead of Write Notices

In JavaSplit implementation of Lazy Release Consistency, write notices are passed along with the lock ownership. A thread invalidates local cached copies based on the received write notices and then fetches the invalidated objects from home, if needed. A possible optimization that would prevent these data misses is to pass the contents of the modified object instead of its write notice. The receiving thread would then update the local cached copy of the object and the object would remain valid. This optimization fits well into existing memory management scheme and can be selectively applied to the smaller objects.

8.5 Home Migration and Load Balancing

The node that executes the thread that creates an object is considered to be the home node of the object. Currently, this mapping is static, i.e., it cannot be changed. Consequently, the node of a thread that creates a large number of objects can become a bottleneck when a large number of threads concurrently fetch these objects or access lock associated with them. To alleviate this problem, an adaptive home migration scheme may be implemented. This would allow mapping objects to homes in such a way that each node manages an equal number of frequently accessed objects. The proposed home migration optimization would require managing a directory service, which is currently unneeded, because home identity is derived from the object globally unique identifier.
8.6 Distributed Garbage Collection

In its current state, JavaSplit does not implement garbage collection of shared objects. (The local objects are collected by the Java built-in garbage collector.) In general, distributed garbage collection is a complex problem, whose solution may have a significant impact on system design and performance. In the presence of fault-tolerance requirement, the complexity of this problem is even greater. Since garbage collection was not the primary focus of this research and due to the fact that in well-designed parallel application the lifetime of the shared-data spans over the larger portion of the application lifetime, we do not implement a distributed garbage collector. Nevertheless, it would be interesting to explore this topic in order to determine its feasibility as well as the impact of distributed garbage collection on system performance and scalability.

8.7 Passing Lock Ownership in Speculative Mode

One of the limitation 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 8.1, a speculative deadlock is avoided because process $A$ is able to transfer ownership of lock $L_2$ to process $B$ during speculative mode.
Figure 8.1: Passing lock ownership in speculative mode
8.8 Predictive Data Push

Section 6.3.2 suggests that applications which inherently push write notices ahead of time may have fewer speculative data conflicts. Experiments show that this is indeed the case.

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 data. Then, whenever a home receives a diff for a shared object, it pushes the data to the 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 8.2 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).
8.8. PREDICTIVE DATA PUSH

Figure 8.2: Predictive Data Push
Chapter 9

Conclusion

This thesis presents and evaluates a fault-tolerant distributed runtime for standard Java applications. This work combines elements of several well-established fields in the core of distributed systems. First, it extends a line of works in the area of distributed shared memory. Second, it refines several techniques in fault-tolerant computing. Third, it incorporates a novel speculative scheme that is based on the ability of the system to capture and reestablish thread state. Finally, it is motivated by ideas in works on harnessing free cycles.

The portability of the proposed runtime is the same as portability of Java. It is designed to run over off-the-shelf components, without modifications of hardware, operating system, networking stack nor the Java Virtual Machine. Moreover, it allows a heterogenous set of nodes participate in the same execution.

The instrumentation techniques devised as a part of this research open new frontiers in rewriting applications implemented in languages, such as Java and C#, which are compiled for and executed on virtual machines. The most significant contribution in this context is the Twin Class Hierarchy approach, which allows instrumentation of library classes incorporated into the virtual machine.

The presented distributed speculative locking scheme demonstrates that speculative locking can significantly improve performance of a general-purpose distributed runtime. Its comparison to distributed transaction memory raises important issues that concern the current state-of-the-art of the latter.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


USENIX Association.


קבץ

בעודזת תחלה, פリング תחתון יחור בו ויתרון במענה ממקמה ממקמה וממקמה וממקמה וממקמה וממקמה וממקמה וממקמה.

שהובה. обязательно לשחריינו, שחלקו בחרה, עד שהשחתה בחרה. החשיבות גיוסה במקמה, ובמקמה, ובמקמה, וב מקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקма, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקма, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקма, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקма, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, ובמקמה, وفي תחנית.
תקציר

acades of experimental studies that employ the application of NLP and machine learning algorithms to tasks such as summarization, sentiment analysis, and named entity recognition. The goal of these studies is to improve the accuracy and efficiency of these tasks by leveraging the power of machine learning models.

In this thesis, we focus on the application of machine learning algorithms to the task of text summarization. We propose a novel approach that combines deep learning models with traditional text summarization techniques. Our approach is evaluated on a large corpus of news articles, and we demonstrate that it outperforms existing methods in terms of both accuracy and efficiency.

We also contribute to the field of machine learning by developing new algorithms for the task of sentiment analysis. Our algorithms are evaluated on a large dataset of movie reviews, and we show that they achieve state-of-the-art performance.

Overall, our work highlights the potential of machine learning models for natural language processing tasks, and we believe that these models will continue to play an important role in the future of NLP.

This work is part of a larger research project funded by the Israel Science Foundation. We would like to thank Dr. [Name] for his support and guidance throughout the course of this project.
איני막דימיםאתהעבדההיאאתלאומיאתקבלותשםלהודיתהגהיניס
הכ开荒 תודה

אסי מדרת הלורד' אוסף שוסטר וולזר' מייקל פקטר על התמיכתו וה助长ו במחלקה וכופה

המדח. הורובנוב היגיסו האקדמאי שבישרו לי עיני רובוט לארץ הדר.

אסי מדרת למל Conexion על התמיכתו הכספית והדרכה ב распростיליות

buahר על מחקר

לש מולי חלקי של הדרישה למקבת חותר
דוקטור לפילוסופיה

וועדה(Cs) שעגי

ורונה לסנט הטעויות - מרכז טכנולוגיה לישראל
נובמבר 2008 היפה
חוש' ט进场
بيةטרב של הורכנות ג'אורה על מספר רב של
החותם עבדרה לא ייעודיות

קורסנטים ישיג

Technion - Computer Science Department - Ph.D. Thesis PHD-2009-11 - 2009