Analyses of Java programs over weak memory

by

David Anthony Winscom Clarke

ORCID 0000-0001-9394-8551

Submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy (with coursework component)

March 2018

School of Computing and Information Systems

The University of Melbourne
Abstract

Between 1980 and 2000, the clock speeds of CPU chips showed an exponential increase from about 1MHz to 1-2GHz. Since then the clock speed of these chips has stagnated. However, contemporary chips offer multiple cores and hardware support for multi-threading. To exploit their power the programmer must use concurrent processing. This is difficult and error-prone.

The Java language has always provided direct high-level support for multi-threading. We provide experimental evidence that these features incur significant overheads. The acquire/release paradigm provided by the synchronized construct relies on the developer to conform to the implicit access protocol. Data race errors occur where this protocol is violated.

Programs that try to avoid the costs of the synchronized construct become exposed to the lack of sequential consistency in weak memory models. Inserting memory fences into the instruction stream restores sequential consistency. Some CPU architectures require few fences, while others require many fences because they have a weaker memory model. In some cases, a memory fence can be replaced by other instruction patterns that may incur fewer overheads than the fence. In our work, we have investigated the direct placement of fences, the default placement of fences triggered by the use of volatile variables, and the elimination of unnecessary fences.

In this thesis we offer four major innovations:

• An analysis that detects data race errors caused by mis-implementation of the acquire/release paradigm. This analysis provides a valuable trade-off between a significant improvement in processing time and a decreased completeness;
• A Java-based extension of the Abstract Event Graph analysis for the optimal selection and placement of memory fences that ensures the sequential consistency of novel synchronisation patterns or lock-free algorithms;
• An improved implementation of memory fences that can reduce the overhead incurred on architectures with relaxed memory models.

• An innovative design for a thread-safe shared data-store class with minimal use of memory fences. We provide experimental evidence that the prototype implementation successfully prevents harmful data races and that its performance is superior to functionally equivalent code that uses conventional synchronisation.

While working on these innovations we noted some related problems:

• The event abstraction that we employ cannot usefully handle references to the elements of arrays or Collections

• Our algorithms have pre-requisite conditions that are not easily satisfied in a static analysis;

We have provided solutions to these problems:

• Java streams encapsulate the handling of Collections. We offer a proof that the actions of Java streams can be summarised within an Abstract Event Graph (AEG) and that the resultant Summarised Abstract Event Graph (SAEG) is no less sound and complete with respect to the detection of data races than the AEG of the program that includes the stream;

• We have developed designs and proof-of-concept that show how elements of our static analysis prototypes may be incorporated within the Java Virtual Machine.

These innovations significantly advance the correctness and efficiency of multi-threaded Java programs and, consequentially, facilitate the full exploitation of contemporary CPU hardware.
Declarations

I formally declare that:

i. This thesis contains only my original work towards the degree of Doctor of Philosophy;

ii. All other work is explicitly cited with references to an included Bibliography;

iii. The thesis contains less than 100,000 words.

David Anthony Winscom Clarke
Acknowledgements

I am grateful to the University of Melbourne. Its regulations allowed the flexibility to recognise years of practical experience and references from peer workers as a valid substitute for recently acquired academic qualifications. My referees, Barry Nugent, Jim Odell, Eric Bodger, Graeme Simsion and the late Richard Barker provided testimony of their confidence in my ability. I thank them for that confidence and hope that I have given them adequate justification.

I am grateful to the members of the School of Computing and Information Systems who patiently aligned my ideas with academic orthodoxy. In particular, I want to thank Harald Søndergaard for taking the time to guide me towards the appreciation that an abstract event graph is not a lattice.

Tony Hosking of ANU gave me a day of his time. He listened to my ideas and made the valuable suggestion that I should look to the internals of the Java Virtual Machine for ways of optimising the implementation of memory fences. Without this insight, Chapter VI would never have existed.

I am indebted to Andrew Haley of RedHat for our correspondence regarding the hardware implementation of memory fences on weak-memory architectures and the relative costs of different ways of ensuring cache coherency.

My supervisors, Tim Miller, Antonette Mendoza and, latterly, Toby Murray have been unfailingly supportive even though my research has taken them into places that are significantly removed from their individual areas of expertise. I hope that they have enjoyed the journey and I thank them for their patience with my occasional bursts of irascibility. I particularly want to thank Toby Murray for his attempt to bring a measure of orthodoxy into my use of set-theoretic arguments.

Finally, I want to thank my friends, whom I have sadly neglected while I have been studying, and my wife, who in spite of illness, has never wavered in her support of my endeavours.
Table of Contents

Chapter I Introduction ................................................................................................................. 19
  I.1. Motivational context ............................................................................................................. 19
  I.2. Research goals ..................................................................................................................... 23
  I.3. Our contributions ............................................................................................................... 23
      I.3.1. Statically detecting data races ....................................................................................... 24
      I.3.2. Restoring sequential consistency ................................................................................. 26
      I.3.3. Optimisations within the Java Virtual Machine .......................................................... 27
      I.3.4. Avoiding data races by construction .......................................................................... 29
      I.3.5. Conclusions ................................................................................................................ 29
  I.4. Structure of thesis .............................................................................................................. 31

Chapter II Related work .............................................................................................................. 33
  II.1. Concerning sequential consistency and data races .......................................................... 33
       II.1.1. Sequential consistency and data races ..................................................................... 34
       II.1.2. Hardware memory models ....................................................................................... 38
       II.1.3. Java memory model .................................................................................................. 45
       II.1.4. Java implementations ............................................................................................... 52
       II.1.5. Java Virtual Machine - components ...................................................................... 53
       II.1.6. Abstract Event Graph ............................................................................................... 55
       II.1.7. Escape analysis ......................................................................................................... 60
       II.1.8. Analysis of parallel loops .......................................................................................... 61
       II.1.9. Thread-safe objects ................................................................................................... 62
  II.2. Other related work ............................................................................................................ 62
       II.2.1. Languages ................................................................................................................ 63
       II.2.2. Reasoning about sequential consistency .................................................................... 63
       II.2.3. Compiler intrinsics ................................................................................................... 64
  II.3. Summary ............................................................................................................................ 65

Chapter III Cost of synchronisation ............................................................................................ 67
  III.1. Measuring relative costs on x86 host .............................................................................. 69
       III.1.1. Cost of synchronized ............................................................................................... 69
       III.1.2. Using SynchroBench - our adaptations ................................................................. 72
       III.1.3. Uniprocessor performance ...................................................................................... 76
       III.1.4. Multi-threaded performance ................................................................................... 77
Chapter IV Slaty detecting data races ............................................. 91

IV.1. Causes of data races ................................................................. 92
IV.2. Finding data races ................................................................. 96
  IV.2.1. De-limiter patterns ............................................................ 96
  IV.2.2. Pre-requisites for finding data races ................................... 99
  IV.2.3. Comparison with Chord ....................................................... 103
  IV.2.4. Commonly taught patterns ............................................... 109
  IV.2.5. Approximations for scalability ......................................... 111
  IV.2.6. Soundness and completeness of data race detection ............. 117
  IV.2.7. Other related work .......................................................... 126
IV.3. Handling access to elements of a Collection ............................. 127
  IV.3.1. Background to Java streams ............................................. 128
  IV.3.2. Our representation of a stream as an Abstract Event Graph .... 131
  IV.3.3. Proving SAEG correctly encodes the AEG ......................... 132
  IV.3.4. Applicability of SAEG to standard iteration ...................... 135
  IV.3.5. Summary of benefits provided by SAEG ............................ 136
IV.4. Implementing our algorithm ................................................... 137
  IV.4.1. Identifying events and their operands ................................ 138
  IV.4.2. Finding critical section de-limiters .................................... 141
  IV.4.3. Associating events with critical sections ............................ 146
  IV.4.4. Grouping critical sections, guards and guarded variables ...... 147
  IV.4.5. Finding data races across critical sections ......................... 148
  IV.4.6. Summary of data race detection algorithm ......................... 149
IV.5. Reusable Framework ............................................................. 149
  IV.5.1. Method invocation hierarchy ............................................ 150
  IV.5.2. Class and method cache .................................................. 151
  IV.5.3. Finding critical sections .................................................. 153
  IV.5.4. Extracting data races from a method summary .................... 154
IV.6. Limitations of prototype ....................................................... 154
IV.7. Observations ....................................................................... 155
  IV.7.1. Measurements of the performance of our prototype .............. 157
  IV.7.2. Comparison with JavaRaceFinder ....................................... 162
  IV.7.3. Comparison with Chord .................................................... 165
IV.7.4. Summary of results........................................................................................................168
IV.8. Conclusions....................................................................................................................169

Chapter V Restoring sequential consistency .................................................................171
V.1. Memory models and fence placement........................................................................172
V.2. Manual analysis.............................................................................................................174
  V.2.1. Alglave’s analysis of AEG cycles.................................................................175
  V.2.2. JSR 133 Cookbook .......................................................................................179
V.3. Our extension of automation to the Java environment........................................182
  V.3.1. Our transformation of bytecode to an Abstract Event Graph (AEG).........182
  V.3.2. Resolving addresses.........................................................................................183
  V.3.3. Our derivation of competing pair relationships.............................................184
  V.3.4. Finding critical cycles - our novel implementation......................................184
  V.3.5. Alglave’s heuristics .........................................................................................186
  V.3.6. Evaluation ........................................................................................................187
  V.3.7. Observations......................................................................................................191
V.4. Conclusions ................................................................................................................193

Chapter VI Optimisations within the JVM .................................................................195
VI.1. Our analysis of JVM bias towards x86 architecture .........................................196
  VI.1.1. CompareAndSet and weakCompareAndSet...........................................196
  VI.1.2. Our proposed extension to VarHandle methods .......................................199
  VI.1.3. Implementation of volatile.........................................................................201
  VI.1.4. Implementing fences.....................................................................................202
  VI.1.5. Summary of current implementations.......................................................204
VI.2. JVM components......................................................................................................205
  VI.2.1. Java Virtual Machine Compiler Interface (JVMCI)..............................206
  VI.2.2. Graal compiler - Internal mechanisms.....................................................207
VI.3. Modifying the behaviour of the compiler.............................................................208
  VI.3.1. Our design for eliminating redundant fences............................................210
  VI.3.2. Our design for replacing a fence with an address dependency...................212
VI.4. Hosting our algorithms within Graal.................................................................214
  VI.4.1. Hosting data race detection.......................................................................215
  VI.4.2. Restoring sequential consistency..............................................................219
VI.5. Conclusions.............................................................................................................220

Chapter VII Avoiding data races by construction..................................................221
VII.1. Motivation...............................................................................................................222
VIII.1.5.  DataStore class........................................................................................................257

VIII.2.  Future directions........................................................................................................258
  VIII.2.1.  Benchmarking synchronisation techniques.........................................................258
  VIII.2.2.  Finding data races ...............................................................................................258
  VIII.2.3.  Optimal selection and placement of fences.........................................................259
  VIII.2.4.  Implementation of memory fences ......................................................................259
  VIII.2.5.  Avoiding data races............................................................................................259

VIII.3.  Final thoughts............................................................................................................260
List of Figures

Figure 1 - acquire/release paradigm ................................................................. 19
Figure 2 - Sample synchronisation .................................................................... 20
Figure 3 - Erroneous maintenance ...................................................................... 20
Figure 4 - Example of MP synchronisation ......................................................... 22
Figure 5 - MP example with fences ..................................................................... 22
Figure 6 - MP send method with fences ............................................................... 23
Figure 7 - AEG for Thread 0 ................................................................................ 27
Figure 8 - Summary of Memory Ordering (McKenney 2010) ........................... 39
Figure 9 - Message passing pseudo-code (Maranget, Sarkar et al. 2015) .......... 44
Figure 10 - MP Litmus test (Maranget, Sarkar et al. 2015) ................................ 44
Figure 11 - Valid inter-leavings for MP Litmus test (Maranget, Sarkar et al. 2015) .................................................................................................................. 44
Figure 12 - Weak memory effects for POWER and ARM (Maranget, Sarkar et al. 2015) ........................................... 44
Figure 13 - Data race (Manson, Pugh et al. 2005) .................................................. 47
Figure 14 - Happens-Before with volatiles (Manson, Pugh et al. 2005) ............ 47
Figure 15 - Out-of-thin-air result (Manson, Pugh et al. 2005) ............................. 48
Figure 16 - Original code ...................................................................................... 49
Figure 17 - After first HotSpot VM transformation ............................................ 50
Figure 18 - After second HotSpot VM transformation ........................................ 50
Figure 19 - Validity of transformations (Ševčík and Aspinall 2008) .................. 50
Figure 20 - Java Virtual Machine components ................................................... 54
Figure 21 - AEG for a Litmus test (Alglave 2010) .............................................. 57
Figure 22 - An execution derived from the AEG (Alglave 2010) ....................... 57
Figure 23 - Abstracted AEG ............................................................................... 58
Figure 24 - Comparison of synchronisation techniques ...................................... 70
Figure 25 - Comparison of techniques with contention ...................................... 71
Figure 26 - Wrapper class example ..................................................................... 74
Figure 27 - Sub-classes for de-limiter patterns ................................................... 74
Figure 28 - No benefit from multi-threading ....................................................... 75
Figure 29 - Vector versus un-sync ArrayList ...................................................... 76
Figure 30 - Java code for CAS pattern ................................................................. 77
Figure 31 - Reentrant use of synchronized ......................................................... 78
Figure 32 - Improved use of synchronized .......................................................... 79
Figure 33 - Java code for ANY lock .................................................................... 80
Figure 34 - Multi-threaded execution with dummy load ...................................... 81
Figure 35 - Multi-threaded execution, lower workload per operation ............... 82
Figure 36 - TreeSet results ................................................................................ 82
Figure 37 - TreeSet with increased contention .................................................. 83
Figure 38 - ArrayList, List interface, Collection of 500 elements ...................... 84
Figure 79 - Message-passing one variable .................................................................................. 146
Figure 80 - Implementation schematic..................................................................................... 150
Figure 81 - Method cache schematic ......................................................................................... 151
Figure 82 - Building classes ....................................................................................................... 152
Figure 83 - Time to process classes (milliseconds) .................................................................... 152
Figure 84 - Build class from bytecode ........................................................................................ 153
Figure 85 - Data races detected by classification ......................................................................... 158
Figure 86 - Sample data race report ........................................................................................... 159
Figure 87 - Processing time (ms) v LOC ...................................................................................... 160
Figure 88 - Processing Time (ms) v Number of critical sections .................................................. 161
Figure 89 - Processing time (ms) v branch statements per method ............................................. 161
Figure 90 - Comparative performance against JRF ...................................................................... 163
Figure 91 - JPF Elapsed time v Number of threads ..................................................................... 164
Figure 92 - JPF Elapsed time v Critical sections ......................................................................... 164
Figure 93 - Typical worker thread pattern .................................................................................. 165
Figure 94 - Chord v Prototype performance .............................................................................. 166
Figure 95 - Chord v Prototype performance chart ....................................................................... 167
Figure 96 - Chord v Prototype (log scale) ................................................................................... 167
Figure 97 - AEG for MP Litmus test ............................................................................................ 173
Figure 98 - Java code fragments for MP pattern .......................................................................... 175
Figure 99 - MP pattern with guarded events ............................................................................. 175
Figure 100 - List of cycles for MP example AEG ......................................................................... 176
Figure 101 - Program order edges with their cycles .................................................................... 176
Figure 102 - JEP 193 memory fence methods ........................................................................... 177
Figure 103 - Required fence types by edge type ......................................................................... 177
Figure 104 - JSR 133 Cookbook recipe ......................................................................................... 179
Figure 105 - AEG for two iterations in Thread 0 .......................................................................... 180
Figure 106 - AEG for MP with common object .......................................................................... 181
Figure 107 - Analysis framework ............................................................................................... 183
Figure 108 - Litmus tests ............................................................................................................ 188
Figure 109 - Results for Litmus tests .......................................................................................... 189
Figure 110 - Fences for Litmus tests for VarHandle architecture ............................................... 189
Figure 111 - AEG for Dekker's mutual exclusion algorithm ......................................................... 190
Figure 112 - Test result for iterated Dekker’s algorithm in two threads ...................................... 191
Figure 113 - Result for single instances of Dekker's algorithm .................................................. 191
Figure 114 - Pseudo-code for compareAndSet ........................................................................... 196
Figure 115 - CompareAndSet from LL/SC ................................................................................ 197
Figure 116 - Conventional CompareAndSet spinlock ............................................................... 198
Figure 117 - Improved weakCompareAndSet spinlock .............................................................. 198
Figure 118 - Decision table for CAS/wCAS v architecture ......................................................... 199
Chapter I Introduction

We divide this introductory chapter into three sections. The first presents a number of pieces of background information that, together, form the motivation for our research. The second section provides a brief description of our contributions. Finally, there is a section that describes the remaining narrative structure of this thesis.

I.1. Motivational context

The present evolution of hardware development provides a good reason why multi-threaded execution must become increasingly main-stream.

Multi-threaded programs inter-communicate most efficiently through shared objects. However, access to these shared objects must be controlled to avoid data races that cause unpredictable results when the program is executed. This control is usually achieved through an acquire/release paradigm.

\begin{verbatim}
Thread 0                     Thread 1
  1: synchronized(c) {
  2:    c.a = 42;
  3:    ...
  4:    x = c.a;
  5: }
  6:    c.a = 99;
\end{verbatim}

This is illustrated by the code of thread 0 shown in Figure 1. The instruction at line 1 acquires the lock \texttt{c}, which is released at line 5. Lines 2, 3 and 4 form a critical section. Thread 0 relies on the guarantee that no other program will access \texttt{c.a} unless it has acquired the lock \texttt{c}. Data race errors occur where the program violates this implicit access protocol. This is illustrated by the code of Thread 1, where line 6 is not within a critical section. Because line 6 has not been synchronised, its execution may or may not interpose between lines 2 and 4. After line 4, \texttt{x} may hold 42 or 99. We say that a data race on \texttt{c.a} exists between lines 2 and 6.

Consider the more practical synchronisation example, Figure 2, which is taken from the Oracle online documentation (Oracle Corp. 2017).
1:public class MsLunch {
  2:    private long c1 = 0;
  3:    private long c2 = 0;
  4:    private Object lock1 = new Object();
  5:    private Object lock2 = new Object();
  6:
  7:    public void inc1() {
  8:      synchronized(lock1) {
  9:        c1++;
10:      }
11:    }
12:    public void inc2() {
13:      synchronized(lock2) {
14:        c2++;
15:      }
16:    }
17:}

Figure 2 - Sample synchronisation

Suppose that a programmer is asked to maintain this class by adding a new variable c3 that should be incremented in both methods inc1() and inc2(). He/she might submit the revised class shown in Figure 3:

1:public class MsLunch {
  2:    private long c1 = 0;
  3:    private long c2 = 0;
  4:    private long c3 = 0;
  5:
  6:    private Object lock1 = new Object();
  7:    private Object lock2 = new Object();
  8:
  9:    public void inc1() {
10:      synchronized(lock1) {
11:        c1++;
12:        c3++;
13:      }
14:    }
15:    public void inc2() {
16:      synchronized(lock2) {
17:        c2++;
18:        c3++;
19:      }
20:    }
21:}

Figure 3 - Erroneous maintenance
Our data race detection system would reveal that even though the events referring to c3 occur within critical sections, there is a data race on c3, because it is accessed under two different guards, lock1 and lock2. The report would identify the instructions at lines 12 and 18 as being responsible for the error.

There is anecdotal evidence that accessing shared variables under the wrong guard is a common coding error particularly when code is maintained. We hold, intuitively, that, in general, the elimination of such data races cannot be achieved by testing alone. It requires the assistance of automated analysis.

In this thesis we provide experimental evidence that the original synchronisation mechanisms provided by the Java language have been implemented in a way that incurs significant execution overheads. Because of the commendable desire for a continuity of language support, we do not expect that these features or their implementation will be changed. The Java Virtual Machine that implements these features must provide them despite the weak memory models provided by contemporary CPU hardware. Programs that seek to avoid the overheads are exposed to and must deal with the effects of these memory models. Ensuring sequential consistency in the face of weak memory models requires the judicious placement of memory fences. Reasoning about the optimal selection of fence types and their placement is not easy. We have designed and developed an algorithm to automate the placement of fences in Java bytecode, building on the contributions of Nimal (2014) in the C language environment.

Here is an example of the way in which the automated fence placement tool might be of practical assistance. Suppose a programmer with some awareness of the weak memory properties of the X86/AMD processor has elected to use critical section de-limiters based on the single-variable message-passing pattern. Figure 4 shows a plausible class based on this design. The send and receive methods would be invoked in separate threads.
1:public class MP {
2:    public volatile int u = 1;
3:    public volatile boolean stop = false;
4:
5:    public void send() {
6:        while (!stop) {
7:            while (u == 0){Thread.yield();}
8:                ...
9:                u = 0;
10:            }
11:        }
12:    }
13:    public void receive() {
14:        while (!stop) {
15:            while (u == 1){Thread.yield();}
16:                ...
17:                u = 1;
18:        }
19:
20:    }
21:

Figure 4 - Example of MP synchronisation

Suppose that, seeking to reduce the cost of the invoked fence instructions, the class is modified to explicitly place fences as shown in Figure 5.

1:public class MP {
2:    public int u = 1;
3:    public boolean stop = false;
4:
5:    public void send() {
6:        while (!stop) {
7:            while (u == 0){Thread.yield();}
8:                ...
9:                u = 0;
10:            VarHandle.fullFence();
11:        }
12:    }
13:    public void receive() {
14:        while (!stop) {
15:            while (u == 1){Thread.yield();}
16:                ...
17:                u = 1;
18:            VarHandle.fullFence();
19:        }
20:    }
21:

Figure 5 - MP example with fences
This code will function correctly on an X86/AMD architecture. However, on an ARM architecture it will not behave reliably. Use of the fence placement system described in Chapter V would show that the Load actions implied by the code at lines 6, 7, 14 and 15 of Figure 5 must be followed by an invocation of VarHandle.acquireFence(). A genuinely universal version of the send method would have to be similar to that shown in Figure 6.

```
1:    public void send() {
2:        int xu;
3:        boolean xstop = stop;
4:        VarHandle.acquireFence();
5:        while (!xstop) {
6:            xu = u;
7:            VarHandle.acquireFence();
8:            while (xu == 0){
9:                Thread.yield();}
10:           xu = u;
11:           VarHandle.acquireFence();
12:       }
13: ...
14:       u = 0;
15:       VarHandle.fullFence();
16:       xstop = stop;
17:       VarHandle.acquireFence();
18:   }
19: }
```

*Figure 6 - MP send method with fences*

The same pattern would be required for the receive method.

**I.2. Research goals**

This thesis considers the following questions:

- What can be done to facilitate the writing of multi-threaded Java programs that are free from data races? and
- How best to minimise the consequential overheads?

**I.3. Our contributions**

We begin the presentation of our work with a description of our systematic investigation of the costs incurred by different ways of synchronising multi-threaded Java programs. We describe how we benchmark the different
techniques by using them to synchronise invocations of the methods of classes from the Collections package. This work is presented in Chapter III.

Then our contributions fall under four major headings:

- Statically detecting data races;
- Restoring sequential consistency;
- Optimisations within Java Virtual Machine;
- Avoiding data races by construction.

I.3.1. Statically detecting data races

We first address the problem of detecting data races in programs that are intended to be free from data races, but contain faulty implementations of the implicit access protocol of the acquire/release paradigm. The algorithm that deals with the finding of data races recognises that some developers may chose to use different de-limiter patterns for implementing the acquire/release paradigm. It assumes that these patterns are known and proven to work correctly, and concentrates on the finding of data races caused by errors in implementing the implicit access protocol. Our basic algorithm for finding data races cannot easily handle programs that access the elements of Collections. We show that the abstract event concept used to underpin our algorithm can be extended to encompass the actions of Java streams. This allows us to extend our algorithm to process programs that use the Java streams feature to handle access to the elements of Collections.

Our algorithm incorporates a grammar-directed parser that can handle critical sections de-limited by a variety of different instruction patterns, but relies on the assumption that these de-limiter patterns effectively provide a guarded section within which there are uniprocessor execution conditions.

The algorithm is a static analysis that searches for data races within sequentially consistent Java bytecode. The essential characteristic of a data race is that there are conflicting read and write actions. The data race exists irrespective of the values that are read and written. The abstract event concept (Alglave 2010) correctly abstracts this "no-values" characteristic. For this reason, we chose to use the abstract event concept and its associated Abstract Event Graph (AEG) notation to reduce a program to an abstraction
that records only a control-flow graph linking events that access shared data. We rely on Alglave's proof that an AEG soundly and completely represents the effects of the execution of a program with respect to weak memory interactions. In our analysis we ignore all calculations and the values that are read and written. All that matters is that a read action or write action on a shared variable has occurred.

In general, static analysis is known to suffer from the challenge of path explosion. We address this in several ways:

- Method-based summaries using *must* and *may* summarisation. In a control-flow with "IF" statements, if an access to a shared variable occurs in both branches of a conditional we say that it *must* occur. Otherwise, it *may* occur;
- Forcing guarded sections to conform to the syntactic block-structuring; and
- Simplifying the resolution of addresses of variables.

Taken together these measures eliminate the need to evaluate all the execution traces of a program and allow an analysis based on the independent examination of methods.

We acknowledge that, where a program has a complicated control-flow graph, our summarisation algorithm will tend to produce many *may* summarisations. Large numbers of *may* categorisations are an indication that at least some of the occurrences will be very rare. There are also a number of particular coding patterns that will generate false positive reports or that are specifically eliminated early in our analysis. We describe these in greater detail in section IV.2.6 of Chapter IV.

Our analysis algorithm has been specifically tuned so that it can exploit the benefits of parallel concurrent execution across multiple threads. In Chapter IV, we show experimentally that this has significant benefits even on platforms with support for only limited numbers of real threads. On our test platform, changing a part of our implementation to use multiple threads reduced its execution time by a factor of five.
Performing static analysis on programs that access the elements of arrays or Collections is known to present significant difficulties because, in general, the number and type of elements is not known at compilation time. The abstract event concept (Alglave 2010) increases the problem because we can no longer distinguish individual accesses to different elements. To facilitate the analysis of programs that encapsulate the access to individual elements within the Java streams feature, we define extensions to the Abstract Event Graph (AEG) notation to represent the actions performed by the lambda expressions passed to the action methods of Java streams. This extended notation is used to describe a Summarised Abstract Event Graph (SAEG). Our analysis assumes that the developer has taken advantage of the typed Collections feature so that the type of elements is known statically. Given this restriction, we provide a proof that such an SAEG soundly and completely represents the stream actions and can be integrated within the overall AEG for a program. We use this result to allow our data race detection algorithm to handle programs that use the Java streams feature. The proof relies on the ability to identify code that behaves as a function similar to that identified in the streams paradigm as a lambda expression. We show that where a loop body satisfies these conditions, an SAEG can be used to analyse the processing of the elements of a Collection.

In the next section, we consider how de-limiter patterns can be optimised and the ways in which the restoration of sequential consistency can be made more efficient.

I.3.2. Restoring sequential consistency

Within a critical section, the developer can rely on uniprocessor conditions. However, the critical section de-limiters themselves must expect to be executed in a multi-threaded circumstance. This means that they must cope with the lack of sequential consistency caused by weak memory models. Appropriately placed memory fences restore sequential consistency. The concepts of strong and weak memory models, and of memory fences are explained in Chapter II. The application of these concepts to the Java environment was documented in Manson, Pugh et al. (2005) and the
consequential advice for the implementation of the volatile construct documented by Lea (2008). We have extended this work to consider more closely the efficiency of those recommendations when applied to the implementation of the Java Virtual Machine (JVM) on weak memory architectures such as ARM (ARM_Holdings 2014).

To restore sequential consistency to a program, we have adapted to the Java environment the analysis of sequential consistency defined by Alglave (2010) and similarly adapted the fence placement algorithm developed for the C language environment by Nimal (2014).

Figure 7 shows the AEG for Thread 0 in Figure 1.

We validate the algorithm for the optimal placement of memory fences by applying it to Java programs that are re-implementations of the Litmus Tests (Alglave, Maranget et al. 2011). By inspection, we confirm that the placed fences are the same as those correspondingly chosen by the experimenters and verified to provide sequentially consistent behaviour. We harmonise this technique with the Java environment by modifying the placement algorithm to use the fence methods specified in Java Enhancement Proposal (JEP) 193 (Lea and Sandoz 2015) rather than specific hardware fence instructions from the target architecture. This conforms more directly to the Java principle that, as far as possible, the Java Virtual Machine (JVM) should isolate the developer from the details of the target environment. This work forms the subject of Chapter V.

I.3.3. Optimisations within the Java Virtual Machine

We have a number of reasons for investigating the integration of our algorithms within the Java Virtual Machine (JVM):

• The present implementation of the JVM is biased towards the x86 target architecture. We provide examples where this would be inefficient if simply ported to a weak-memory architecture.

• The substitution of different machine code sequences for memory fences cannot be performed effectively by an external static analysis.
Any such optimisations that are performed externally may be invalidated by the optimisations performed within the Java Virtual Machine (JVM). These optimisations properly reside within the implementation of a JVM for a specific architecture;

- The algorithms for finding data races and for investigating sequential consistency have certain pre-requisite conditions that are not easily satisfied within a static analysis. Our palliative approximation techniques impair the completeness of the analysis and force the premature rejection of certain coding patterns.

I.3.3.1. Java Virtual Machine background

The standard Java Virtual Machine (JVM) (Oracle_Corp. 2014) comprises an Interpreter and two Just-in-Time (JIT) compilers, C1 and C2. The C2 compiler uses profile information to transform the bytecode to form a "normal" execution path. It is expected that, during normal execution, all conditional instructions in this normal path will "fail", that is they will "fall through" to the next instruction in-line. Most of the successful branches lead to code that returns control to the JVM, though some return to the main instruction stream. This configuration of the code is optimal for its efficient execution by pipelined and cached processors. The Graal project (http://openjdk.java.net/projects/graal/) has developed a Just-in-Time (JIT) compiler, written in Java, that provides an extensible replacement for the C2 compiler.

I.3.3.2. Our work

The "normal path" sequences of instructions generated by JIT compilers, including Graal, satisfy the conditions for the resolution of addresses needed for the efficient use of our data race detection and fence placement algorithms. We have used the Graal compiler together with an early-access release of Java 9 to build proof-of-concept implementations that:

- Omit an acquireFence if the necessary address dependencies already exist or replace it with alternative instruction patterns to provide improved performance;
• Show the feasibility of incorporating parts of our static analysis prototypes within the compiler environment;

I.3.4. Avoiding data races by construction

Data races are caused by failure to respect the access protocol implicit in the use of the acquire/release paradigm. Building on the knowledge gained through our research, we offer the novel thread-safe DataStore class as a way of avoiding this problem. This work is described in detail in Chapter VII. We have applied the minimal set of fences proposed by the AEG-based analysis for sequential consistency to locking algorithms based on the atomic CompareAndSet technique. We show experimentally that this implementation offers superior performance to those based on the features of earlier Java releases.

The goal of the DataStore class is that it shall be impossible for a user of the class to violate the implicit access protocol it imposes. We offer here a description of the way this is achieved by using the standard features of the Java language.

The DataStore class demands that its stored object is an instance of a class that extends the DSObject class. The attributes of this class and its subclasses are made immutable by using the final construct. There is a companion class, DSObjectMutable that has the same attributes as its immutable variant. These attributes are non-final. The DataStore, DSObject and DSObjectMutable classes mechanise a CopyOnWrite paradigm while allowing un-fettered access for readers. Our empirical results show that this design offers good performance over a surprisingly wide range of operating conditions. When normalised against a conventional implementation using the synchronised construct, our DataStore class shows improved performance by a factor that ranges from two to eight, depending on the operating conditions.

I.3.5. Conclusions

We have addressed the problem of finding data races in an otherwise correct program by using a static analysis. We provide an extension to the Abstract Event Graph notation to accommodate programs that access elements of a
Collection using the Java *streams* feature and show that this extension can also be applied to iterations that do not explicitly enumerate elements of a Collection. We use the SAEG notation to extend our algorithm for finding data races so that it can handle programs that access the elements of Collections in these ways. The algorithm is successful, relying on our approximations and simplifications to achieve adequate scalability. Our experimental results show that it offers significantly shorter processing times when compared to systems based on JavaRaceFinder (JRF) (Kim, Yavuz-Kahveci et al. 2012) or the static analysis of Chord (Naik, Aiken et al. 2006).

We have investigated the relative performance of different de-limiter patterns and offer an effective static analysis algorithm for verifying that a candidate de-limiter pattern efficiently restores sequential consistency.

We have shown that it is possible to use the features of the Just-in-Time (JIT) compiler environment to restore sequential consistency with different instruction patterns rather than memory fences. This offers the possibility of some improvements in efficiency. This analysis includes the replacement of memory fences with alternative instruction sequences and the elimination of unnecessary memory fences where existing address dependencies already provide sequential consistency.

We have shown how our static analysis algorithms can be simplified when incorporated within the JVM.

We have used the knowledge acquired during this work to design and implement an efficient thread-safe DataStore class as a way to avoid data race errors.

Taken together, these innovations significantly improve the ease with which multi-threaded programs may be built and, consequentially, facilitate the full exploitation of the power of contemporary hardware.
I.4. Structure of thesis

Including this chapter, the thesis is divided into eight chapters.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>Related work</td>
</tr>
<tr>
<td>III</td>
<td>Cost of synchronisation</td>
</tr>
<tr>
<td>IV</td>
<td>Statically detecting data races</td>
</tr>
<tr>
<td>V</td>
<td>Restoring sequential consistency</td>
</tr>
<tr>
<td>VI</td>
<td>Optimisations within the JVM</td>
</tr>
<tr>
<td>VII</td>
<td>Avoiding data races by construction</td>
</tr>
<tr>
<td>VIII</td>
<td>Conclusions</td>
</tr>
</tbody>
</table>

Chapter II, which deals with related research, is divided into two parts. The first part describes the prior research on which we rely within our work. The second describes other research, which, though related, bears less directly on our work.

Chapter III provides the experimental evidence that justifies the attempt to find more efficient substitutes for the conventional synchronized construct and sets out the argument for shorter, less-costly critical sections. We report the results of our extensive investigation into the relative costs of different synchronisation techniques.

Chapter IV gives a description of our algorithm for finding data races within an otherwise correct multi-threaded Java program and presents our extension to the Abstract Event Graph notation to accommodate accesses to elements of a Collection. This chapter assumes that the critical section delimiters operate correctly in a weak memory environment.

Chapter V describes our investigation of critical section delimiters. We use the AEG analysis of sequential consistency to investigate the improvements in performance that may be achieved by a less conservative approach to the use of fences than that presently used within Java Virtual Machine (JVM) implementations. We extend this to a consideration of the use of architecture-specific instruction sequences that can preserve sequential consistency without the use of explicit fence instructions. Finally, we describe the way in which our developed code base was re-used to implement a prototype of the fence selection and placement algorithm in and for Java.
The algorithm described in Chapter V for the optimal selection and placement of memory fences delivers amended bytecode with inserted generic memory fences. Chapter VI begins by describing how the features of the JVM may be used to improve the implementation of the generic memory fences on weak memory architectures and provides a proof-of-concept implementation of these principles. In the later part of the chapter, we examine the potential benefit of incorporating elements of our static analysis algorithms described in Chapter IV and Chapter V within the JVM and conclude by describing our design for this task.

In Chapter VII we describe the novel DataStore class. This class is a successful response to the question "Is it possible to avoid data races by enforcing an implicit access protocol using only the standard features of the Java Language?".

Finally, there is Chapter VIII that records our conclusions and provides some indications of a direction for future research.
Chapter II Related work

"Caesar primum suo deinde omnium
ex conspectu remotis equis
ut aequato omnium periculo
spem fugae tolleret
cohortatus suos proelium commisit "

"De Bello Gallico"

Julius Caesar

We divide this chapter into two parts. The first part describes previous and on-going research that forms the essential background to our work. The second part describes other important work that is less directly related to our research.

II.1. Concerning sequential consistency and data races

We begin with the papers that define sequential consistency and data races. These are described in section II.1.1. This is followed by a review of prior work that has produced tools intended to find data races in multi-threaded programs. The research on informal synchronization leads to an examination of alternative synchronisation techniques. These techniques are exposed to the effects of the weaker memory models provided by contemporary CPU architectures and an examination of those techniques requires a level of knowledge of memory models. We provide a review of the work on these models in section II.1.2. The Java Memory Model (JMM) tries to insulate the developer from these matters, but the implementers of the Java Virtual Machine (JVM) must efficiently implement the specification of the JMM by using the features of the hardware models. We provide a full review of the theoretical work on the JMM in section II.1.3 with a review of the subsequent Java implementations in section II.1.4. The specification of Java 9 exposes an interface to the Java Virtual Machine that allows an external Just-in-Time (JIT) compiler, written in Java, to work together with the JVM. This opens the possibility of further optimisations of the way in which memory fences are implemented for architectures such as ARM and
POWER. In section II.1.5 we provide an overview of the Graal compiler, an open source Java project that uses this interface.

Any replacements for the existing synchronisation techniques must be more efficient while retaining all the necessary correctness. Ensuring the correctness is difficult. In section II.1.6, we review the work done in analysing programs for sequential consistency, finding a lack of sequential consistency and restoring it by using appropriate features of the target hardware architecture. Next, we provide a brief review of existing work on the escape analysis of Java programs and indicate its relevance to our work. In section II.1.8, we review the work of Radoi and Dig (2015) in searching for data races in Java programs with parallel processing loops.

We introduce the Java streams feature and explain how we extend our work to handle programs that process Collections by using the streams feature.

Finally, we provide a series of introductions to related work that has influenced but has not been directly used within our work.

II.1.1. Sequential consistency and data races

We begin this section by introducing the research that defined a sequentially consistent process and provide definitions of data races, of critical sections and of their de-limiters.

Dijkstra’s seminal paper (Dijkstra 1971) defined one of the important functions of an operating system as building layers of abstraction so that the programmer is insulated from the indeterminism inherent in the management of peripheral devices. His work carries the implicit assumption that reasoning about programs relies on a belief that the results observed are repeatable and identical with those that would have been observed had the instructions been executed in the order written by the programmer. The term sequential consistency to describe this behaviour first occurs in a discussion by Lamport (1979) of the necessary characteristics of a usable multi-processor system. This paper pre-dates the era of CPUs that can support parallel processing through the concurrent execution of multiple threads, but all subsequent work has accepted the premise that reasoning
about programs is possible only if it can rely on a guarantee of sequential consistency.

A data race occurs when several threads simultaneously make unregulated access to a shared memory location and at least one of them makes a write access. We have adopted this simple definition, though the matter was more extensively researched in (Netzer and Miller 1992). For example, we have not made any special provision for the case where a program ensures that a shared variable is only ever updated in a single thread, though there is at least one reader in a different thread. Although we would regard this situation as a data race, it is not, provided that the variable is declared as volatile, so that there can be no detrimental weak-memory effects.

Most of the techniques for avoiding data races rely on regulating the access to the shared data so that only one thread can access it at a time using an acquire/release paradigm. We refer to these regulated sections of code as critical sections. Within a critical section, the thread is assured uniprocessor conditions. All contemporary CPU architectures guarantee sequential consistency under uniprocessor conditions. Thus, the code within a critical section can rely on sequential consistency. We note here that the acquire/release paradigm does not ensure freedom from data races unless the participating threads conform to the implicit access protocol, which is that a thread must not access the guarded data unless it is executing within a critical section that uses the guard that protects the data.

Although the C language environment has previously provided library support for multi-threaded working, it is only with the definition of C++11 (ISO 2014) that the language provided formal support for memory fence operations. Accordingly, most of the effort in this environment has concentrated on the detection of a lack of sequential consistency in weak memory environments. We have concentrated our work on the Java environment, which has always provided formal language support for multi-threading and synchronisation.

Although the amount of open source code is increasing, in a commercial environment there are still many programs where the source code is a proprietary secret. However, by definition of the Java environment, Java
bytecode is always available for analysis. Accordingly, we have concentrated our research on tools and techniques that start with Java bytecode, rather than the source code.

We began our research by looking for existing tools that can find data races in code. The FindBugs (Ayewah, Pugh et al. 2008) and Keshmesh (Vakilian, Negara et al. 2013) tools rely on pattern matching to detect common errors. This technique can find many other patterns as well as those that give rise to data races. The approach has proved effective but is limited by the ability of the operator to configure the tool. It cannot find unforeseen error patterns, a limitation acknowledged by the authors of these tools.

Yin (2013) discusses synchronisation patterns that use volatile variables, and describes the use of the Chord tool (Naik, Aiken et al. 2006) as a framework for identifying these paradigms in Java programs. From our viewpoint, his work is of interest primarily because he analysed existing programs that used informal synchronisation techniques and identified a number of common patterns. He applied his analysis to seventeen programs whose source code was generally available.

They were:

- tsp, a travelling salesman problem solver;
- jtpcc, a TPC-C benchmark (http://jtpcc.sourceforge.net/);
- Eleven programs from the Java Grande Benchmark Suite (http://www.epcc.ed.ac.uk/research/java-grande/);
- raja, a ray tracer with a graphic user interface (http://raja.sourceforge.net/);
- jbb, the benchmark SPEC JBB2000 (http://www.spec.org/osg/jbb2000/);
- avrora, an AVR micro controller simulation program, which comes from the Dacapo Benchmarks (http://dacapobench.org/); and
- jigsaw, W3C’s leading-edge Web server (http://www.w3.org/Jigsaw/).

From these he extracted five patterns of informal synchronisation:
• Barrier - used by barrier, lufact, moldyn and raytracer;
• Flags - used by jbb;
• Status variable - used by raja;
• Avrora-style - used by avrora;
• Lazy - used by sync, tsp, jtpcc and jigsaw,

and quoted by Pugh (2000) as a common example of flawed code.

This survey illustrates the extent to which developers have historically tried to avoid the overheads of the synchronized construct. Xiong et al. (2010) note that although attempts at informal synchronisation are quite common, the great majority of them are flawed. Often, they achieve signalling between threads but do not implement an effective acquire/release protocol.

We rejected the use of the Chord framework, itself, because it uses the SOOT package to read bytecode. This package has not been updated to handle the Stack Frame Maps that were made mandatory in Java 8. We were particularly keen that our work should embrace the features of Java 8 and Java 9.

JavaRaceFinder - Extended (JRF-E) offers an extended analysis of the data races discovered by JavaRaceFinder (Kim, Yavuz-Kahveci et al. 2012), which is based on the JavaPathFinder (JPF) tool (Visser and Mehlitz 2005). The JPF uses a framework around the standard Java Virtual Machine to perform the controlled execution of multiple threads. The JRF-E development employs the extensibility features of JPF to search for data races and thence, to offer suggestions for their elimination. This approach, by definition, correctly reflects the behaviour that would be observed under operational circumstances. However, the algorithm is computationally intensive and consumes large amounts of memory. The reported processing times range from 2 seconds to 600 seconds for programs whose size is of the order of hundreds of lines of code. In their reported results, a significant number of tests could not be completed because the analysis program ran out of heap space memory.

Ferrara (2013) used abstract interpretation (Cousot 1996) with standard abstractions, such as zero/nonzero, sign and numeric range, to investigate the finding of defects in Java programs including the detection of
data races. His published results demonstrated that abstract interpretation makes practical the static analysis of relatively large Java programs. His work was restricted to critical sections identified by the `synchronized` keyword, and did not address the problem of critical sections de-limited by read and write actions on `volatile` variables, or other de-limiter patterns. The published performance measurements indicate that adjusting the abstractions used to improve completeness would detract from scalability.

We chose to start with bytecode rather than with source code because, by definition, bytecode must be available, whereas source code may be commercially secret. We rejected JavaRaceFinder (JRF) because of our intuition that a static analysis approach might yield significantly lower processing times albeit at the expense of an increased number of false positive reports. Ferrara's work showed that using conventional abstractions gave acceptable performance but was incomplete because the abstractions used were unrelated to the behaviour that caused data races. Accordingly, we searched for an abstraction that was more closely related to memory accesses. In section II.1.6, we discuss the Abstract Event Graph (AEG), an abstraction that fulfills this criterion.

II.1.2. Hardware memory models

Within a critical section, there is an implicit guarantee that the code is executing in uniprocessor circumstances. Conversely, the code of the patterns that de-limit critical sections must expect to be multi-threaded. Contemporary CPU architectures can deliver some bizarre results when a multi-threaded program accesses variables that are shared across threads. Reasoning about programs that are exposed to such results is difficult, because of the large number of potential inter-leavings possible between the execution paths in multiple threads. In this section, we introduce and explain:

- the differences between `strong` and `relaxed` or `weak` memory models;
- the notion of `memory barriers` or `fences` as the way to restore sequential consistency to a program executing under a weak memory model; and
• _atomic_ operations as the in-built hardware support for locking operations.

Most contemporary CPU architectures guarantee sequential consistency for execution within a single thread. This is called a _strong_ memory model.

However, as a consequence of hardware and software optimisation techniques, the actual execution order of instructions within a thread may be very different from that written by the programmer. Instructions may be re-ordered or may be re-written to avoid redundant read operations. Write operations may be omitted if there is no subsequent read of the same variable. Variables may be mapped into registers so that they never become manifest within the main memory. Different threads may have local views of a shared variable. Instructions may be speculatively executed in advance of their written sequence, and so on. This is termed a _weak_ or _relaxed_ memory model. It provides no guarantee of determinism when a thread observes values of variables shared with other threads.

McKenney (2010) provides a useful and informative introduction to the hardware mechanisms that speed execution but cause weak memory effects.

![Figure 8 - Summary of Memory Ordering (McKenney 2010)](image-url)
The characteristics of different processor architectures are described in Figure 8, which is quoted from McKenney. An earlier view of the problems of memory consistency was provided by (Adve and Gharachorloo 1996). The detail of these mechanisms varies widely between processor architectures and, indeed, between implementations of those architectures. What is important is that the weak memory effects exist and make the execution lack sequential consistency. McKenney’s paper also provides an accessible explanation of the way memory barrier or memory fence instructions can be used to restore sequentially consistent execution.

Today, most laptop, desktop and server CPUs use the x86 architecture or the AMD64 architecture, which for our purposes is equivalent. Implementations of the ARM architecture are dominant for smartphones and tablets. It is reported (Trader 2016) that Fujitsu, who have been contracted to develop a super-computer for the Japanese government, have elected to base its design on the ARM architecture even though previously their developments used Fujitsu’s implementation of the SPARC architecture. Accordingly, we have focussed our efforts on the x86 and ARM architectures.

The figure shows that, of the popular contemporary CPU architectures, the x86/AMD64 architecture comes closest to being sequentially consistent, while the ARM architecture has a very relaxed memory model.

Most published hardware memory models concentrate on behaviour of the hardware. They do not, generally, provide any firm indication of the implications of this behaviour for programs. The earliest definition of a memory model specifying the semantics of a program was provided for the SPARC processor (SPARC International 1992).

It presented three memory models that could be selected by a program:

- **SPARC-TSO**
  A total store order providing generally sequentially consistent behaviour. Only Store-Load edges need a fence action to provide full sequential consistency;

- **SPARC-PSO**
  A partial store order which allows some re-ordering of instructions and does not guarantee full sequential consistency across threads. In
particular, when compared with TSO, it allows the re-ordering of Store actions with other Store actions and with atomic instructions;

- **SPARC-RMO**
  
  A fully relaxed memory model that, again, does not guarantee sequential consistency across threads.

The precise details of the memory re-ordering allowed by each of these models are given in Figure 8. Other architectures provided descriptions of the required behaviour of the hardware, but is only more recently that memory models have been developed for the popular x86 and ARM architectures that provide semantics for the effects of the hardware behaviour on programs.

The x86 architecture (Sewell, Sarkar et al. 2010) generally provides sequentially consistent behaviour. Only one fence type is required. This full fence ensures that Store actions (the ultimate result of Write events) are not re-ordered after subsequent Load actions (corresponding to Read events). There is a *mfence* instruction that explicitly provides this effect. The same effect may also be achieved by adding the *lock* prefix to certain instructions. These instructions make explicit the fact that the *lock* prefix makes the change to the memory location visible simultaneously to all processing threads as well as enforcing sequential consistency within the thread that issues the instruction. Accordingly, a *lock:add* that adds zero to the memory address is commonly used as a full memory fence.

Although there are various classic algorithms that organise mutual exclusion using distinct Load and Store operations, it is much simpler to use a CompareAndSet instruction that atomically tests the memory location against a parametrically supplied value and replaces it with a different parametrically supplied value if the test succeeds. A similar instruction has been part of the IBM 370 architecture (IBM 1983) since 1970. The x86 architecture has a *lock:cmpxchg* instruction provided for this purpose. It performs the actions atomically and has the effect of a full memory fence. Sewell, Sarkar et al. (2010) provide a description of the programming model for the x86 architecture.
The ARM architecture (ARM_Holdings 2014) has a relaxed memory model. All the combinations of Load and Store actions may be re-ordered with respect to each other. The provided dmb direct memory barrier instruction has a number of different modes of operation, but, as McKenney (2010) explains, only the "global scope" mode completely satisfies the conditions for restoring sequential consistency. Naively, and expensively, sequential consistency might be restored by planting a dmb instruction in every edge between memory access events. However, the ARM architecture generally respects address dependencies so that Load and Store actions on the same memory address are not re-ordered. This knowledge may be used to reduce the number of instructions needed to achieve the required effect.

Rather than CompareAndSet, the ARM architecture provides the LoadLinked/StoreConditional (LL/SC) paradigm. It has been formally proven (Herlihy 1991) that CompareAndSet can be constructed out of LoadLinked/StoreConditional and vice-versa. The LoadLinked instruction retrieves the value from a memory location and sets a flag. The StoreConditional instruction tests the flag and, if it is still set, unsets it and stores a new value in the memory location. If the flag is not set, the Store action is not performed. If other threads perform store operations adjacent to the memory location, the flag is unset. The interpretation of "adjacent" depends on the particular hardware implementation. In early implementations, the flag covered a complete memory module. Contemporary implementations are more specific, such as "the memory locations that would share the same cache line". However, the precise details of different implementations tend to be retained as commercial secrets. The LoadLinked/StoreConditional pair is not atomic and a limited number of other instructions may be executed between them by the issuing thread. The ARM architecture's LoadLinked and StoreConditional instructions are never re-ordered with respect to each other, though other Load and Store instructions may be re-ordered with respect to them. This means that a complete equivalent to the x86 lock:cmpxchg instruction requires the combination of ldaxr; ...; strlxr; followed by a dmb instruction to provide a full fence. The full fence ensures that the effect of the StoreConditional is made
visible to all other threads and that prior Store actions complete before Loads that follow the LL/SC pair.

There is a long-standing group of researchers who have developed memory models for the various hardware architectures that are useful for programmers (Sewell, Sarkar et al. 2010, Alglave, Maranget et al. 2011, Sarkar, Memarian et al. 2012, Alglave, Maranget et al. 2014, Maranget, Sarkar et al. 2015). By collaborating with the designers and implementers of hardware architectures they have been able to devise models of the behaviour of these processors that are more useful in aiding the design of programs that are sequentially consistent. Maranget, Sarkar, et al. (2015) provides a programmer’s introduction to the memory model for processors with relaxed memory models. Some researchers have also undertaken the difficult task of validating these models experimentally against the behaviour of real hardware implementations. To perform these experiments, they have devised a series of very short and simple programs that are known as the Litmus tests (Alglave, Maranget et al. 2011). Each test delivers a small set of observed values. The memory model for the architecture is used to predict these values. Where weak memory is involved in a multi-threaded circumstance, these values will include, not only the values that would be expected in a uniprocessor case, but also the values predicted by an exhaustive evaluation of all the values that might occur when the full inter-thread interactions of all the possibilities allowed by the architecture are considered. Experimentally, a Litmus test is run repeatedly against a representative hardware configuration for a particular architecture. The values observed as the result of each trial are recorded. As expected, the majority of these results will reflect the uniprocessor case. However, in a small minority of cases, other values are observed. If these accord with the values predicted by the model, this is taken as a validation that the model is correct. If unpredicted values are observed, this is taken as an indication that the model must be revised or enhanced. In other tests, results, appropriate to weak memory behaviour, predicted by the model are never observed in a large number of trials. This is interpreted to imply that the hardware implementation is less relaxed than the definition of the architecture that it
implements. These Litmus tests have been given a taxonomy so that each test can be uniquely identified by a short alphanumeric sequence that succinctly describes its characteristics. Figure 9 shows the well-known pattern for message passing between two threads (MP).

![Figure 9 - Message passing pseudo-code (Maranget, Sarkar et al. 2015)](image)

As a Litmus test this reduces to the code shown in Figure 10.

![Figure 10 - MP Litmus test (Maranget, Sarkar et al. 2015)](image)

Given the extreme simplicity of the code, it is practical to enumerate the weak memory inter-leavings that respect the program order in each thread as illustrated in Figure 11.

![Figure 11 - Valid inter-leavings for MP Litmus test (Maranget, Sarkar et al. 2015)](image)

Other inter-leavings, such as $r_1=1 \land r_2=0$ are forbidden by sequential consistency and by the x86/AMD64 architectures, but are allowed by the ARM (ARM_Holdings 2014) and POWER (May, Silha et al. 1994) architectures. Figure 12 shows the results obtained by running this test on a variety of different hardware implementations of the ARM and POWER architectures. The ratios displayed show the number of times the forbidden result was observed and the total number of trials.
The definitions of the Litmus tests shown in Figure 9 and Figure 10 and the results shown in Figure 11 and Figure 12 are quoted directly from Maranget, Sarkar et al. (2015). They are typical of a very large body of experimental work that covers the spectrum of weak memory behaviour across a wide range of different architectures and implementations. They show that even though the Litmus tests are very compact and specifically designed to expose a lack of sequential consistency, the fraction of trials where this behaviour is observed is very small. This provides empirical support for the view that it is futile to use conventional testing techniques to try and find errors caused by the execution of code in a weak memory environment. The group has also researched the ways in which the features of various architectures may be used to restore sequential consistency where it is required. They have used their memory models to predict the optimal selection and placement of fence instructions and then experimentally verified that these placements do, indeed, restore sequential consistency. It is this body of work that informs the advice regarding the implementation of Java Virtual Machines that we discuss in section II.1.4 of this chapter.

Some authors use the terms Store and Load. Others use the terms Write and Read. We have chosen not to attempt to impose either convention on the other, but to use the words of the authors in the belief that this will, ultimately, cause less confusion.

II.1.3. Java memory model

Saraswat, Jagadeesan et al. (2007) proposed a theory of memory models, but the C and C++ languages have only recently been provided with a memory model that allows for the effects of weak memory (Batty, Owens et al. 2011). Conversely, the original Java Language Specification (Gosling, Joy et al. 1996) supported multi-threading. The `synchronized` keyword ensures that blocks of code are executed by only one thread at a time. Other threads are blocked until the monitor for the section of code is released. The `volatile` keyword

<table>
<thead>
<tr>
<th>Kind</th>
<th>POWER</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PowerG5</td>
<td>Power6</td>
</tr>
<tr>
<td>MP</td>
<td>10M/4.9G</td>
<td>6.5M/29G</td>
</tr>
</tbody>
</table>

Figure 12 - Weak memory effects for POWER and ARM (Maranget, Sarkar et al. 2015)
was intended to provide finer-grained coordination on single atomic variables. Maessen and Shen (2000) described an improved synchronisation technique. But the major change was initiated by Pugh (1999) who identified a number of test cases where the specification provided an inadequate definition of the results that might be observed if the programs were to be executed as genuinely concurrent threads in which full rein was given to the known hardware and software optimisations. He proposed a significant revision of the specification to limit the optimisations that might be employed by the Java Virtual Machine (JVM), compilers and the CPU hardware. A committee that included programmers, compiler writers and designers of CPU architectures extended this work. Manson, Pugh et al. (2005) formally documented these deliberations. The consequential changes were specified in (Pugh 2004) and published in Chapter 17 of the specification that was released with Java 5 (Gosling, Joy et al. 2005). This specification states axiomatically the behaviour that must be observed as the result of the execution of multi-threaded programs. It does not specify how this is to be achieved, thus allowing considerable flexibility in the design of hardware and in the compiler optimisations that may be used.

II.1.3.1.Memory model guarantees

When Manson, Pugh et al. (2005) set out the rationale for the revised Java Memory Model (JMM), they acted from the premise that, wherever possible, sequential consistency should be guaranteed. Where this requirement conflicted too violently with the results of compiler and hardware optimisations, they adopted the standard that the results should be "reasonable", while accepting that this might be a subjective compromise. For the first time, they defined not only the semantics that "compilers", in the broadest sense of that term, must deliver so that properly synchronised programs can assume sequential consistency, but also the semantics for incorrectly synchronised programs.

To develop the specification, Manson, Pugh et al. considered a variety of test cases and solved the problems they presented by defining an
appropriate behaviour model. The test cases shown in Figure 13, Figure 14 and Figure 15 below are quoted directly from Manson, Pugh et al.

**Behaviour of correctly synchronised programs**

Consider the execution of the program shown in Figure 13. Each thread perceives its execution order to conform to the *program order*. However, as viewed by the other thread, it may be very different. There is a *data race* on the variable $z$ and the final value in $r_1$ may or may not be 1.

Initially, $x = 0$ and $z = \text{false}$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x = 1$;</td>
<td>3: if ($z$) $r_1 = x$;</td>
</tr>
<tr>
<td>2: $z = \text{true}$;</td>
<td>|</td>
</tr>
</tbody>
</table>

If $r_1 = x$; executes, it will read 1.

**Figure 13 - Data race (Manson, Pugh et al. 2005)**

By declaring a variable to be *volatile*, as shown in Figure 14, a reasonable programmer seeks an assurance that if thread 2 reads $v$ as *true*, then it will read $x$ as 1. To provide this assurance, Manson, Pugh et al. proposed the *happens-before* model. This defines *volatile reads* and *volatile writes* as synchronisation actions so that, in the case of Figure 14, the use of a *volatile* variable $v$ makes 2: a *volatile write* and 3: a *volatile read*.

$v$ is a volatile variable

Initially, $x = 0$ and $v = \text{false}$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x = 1$;</td>
<td>3: if ($v$) $r_1 = x$;</td>
</tr>
<tr>
<td>2: $v = \text{true}$;</td>
<td>|</td>
</tr>
</tbody>
</table>

If $r_1 = x$; executes, it will read 1.

**Figure 14 - Happens-Before with volatiles (Manson, Pugh et al. 2005)**

The happens-before order ensures that any volatile read will synchronise with the last volatile write to the same variable and that subsequent reads of other shared variables, such as $x$, will read sequentially consistent values as of the time of the volatile write. Correct sequential consistency is assured
because 1: precedes 2: in program order and 2: precedes 3: because of the happens-before relationship.

The Java Memory Model specifies other synchronisation edges to ensure that, for example: other threads cannot prematurely observe final variables; and read actions subsequent to thread-join actions correctly observe the last write actions of concluding threads.

**Behaviour of incorrectly synchronised programs**

Manson, Pugh et al. particularly addressed the undesirable behaviour that is allowed by the simple happens-before model when the program is not correctly synchronised.

Initially, \( x == y == 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r_1 = x );</td>
<td>3: ( r_2 = y );</td>
</tr>
<tr>
<td>2: ( y = r_1 );</td>
<td>4: ( x = r_2 );</td>
</tr>
</tbody>
</table>

Incorrectly synchronised, but we want to disallow \( r_1 == r_2 == 42 \).

**Figure 15 - Out-of-thin-air result (Manson, Pugh et al. 2005)**

Figure 15 describes a program that has data races on both \( x \) and \( y \). It is, therefore, incorrectly synchronised. The surprising result illustrated in Figure 15 is possible under certain optimisations. Suppose the compiler decides to speculatively execute 2: \( y = r_1 \); first. The result would depend on the stale value held in \( r_1 \), which we might suppose to be 42. This is referred to as the "out-of-thin-air" value. Once this has occurred, then the sequence \( (3, 4, 1) \) yields the result:

\[ r_1 == r_2 == 42. \]

This raises concerns where, for example, the out-of-thin-air value happens to be a reference to an object to which the code should not have access. Such a value would offer a security loophole similar to out-of-bounds array access.

Earlier language memory models, such as that for the Ada language (Barnes 1995), used terms like "erroneous" and "undefined" to dismiss the effects of weak memory models on data race conditions. In the interests of
security, Manson, Pugh et al. insisted that program semantics must be fully defined, even for incorrectly formed programs. The proposed solution to this need was the notion of causality. The problem in defining this notion is to prohibit the behaviour described by Figure 15, by restricting the permitted optimisations, without losing all their benefits. Causality "justifies" reordered statements by finding a sequentially consistent execution in which that statement is executed. This is referred to as a well-behaved execution. The whole execution of a program is well-behaved if it can be validated iteratively by committing actions from a well-behaved execution and then repeating the process until all the remaining actions have been committed.

Outcomes

For the first time, the Java 5 version of the Java Memory Model (Gosling, Joy et al. 2005) provided assured semantics for multi-threaded programs. It rationalised the semantics of correctly synchronised programs and, again uniquely, introduced a model, that of causality, which set bounds on the behaviour of ill-synchronised programs so that safety and security were preserved. This was a very significant improvement.

II.1.3.2.Subsequent work

Cenciarelli, Knapp et al. (2007) criticised the Java Memory Model for its lack of formal definition and informal reliance on specific examples. They proposed a formal denotation and proved that the various properties specified axiomatically by Manson, Pugh et al. could be derived from the axioms of their denotation. Their work considered only finite executions that exclude dynamic allocation. Botincan, Glavan et al. (2009) considered whether the process of iteration described in the causality model used by Manson, Pugh et al. can establish whether any particular execution is well-behaved and concluded that, in general, this question is undecidable.

Initially, x==0, y==0

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = x;</td>
<td>r2 = y;</td>
</tr>
<tr>
<td>y = r1;</td>
<td>x = (r2==1)?y:1;</td>
</tr>
<tr>
<td></td>
<td>print(r2);</td>
</tr>
</tbody>
</table>

Figure 16 - Original code
The code presented in Figure 16 poses another problem. Ševčík and Aspinall (2008) suggested that some common compiler optimisations in use, such as the re-ordering of memory access to non-volatile locations, might violate the Java Memory Model. This original code cannot print 1.

Initially, x==0, y==0
Thread 1
r1 = x;
y = r1;
print(r2);

Thread 2
r2 = y;
x = 1;
print(r2);

Figure 17 - After first HotSpot VM transformation

Initially, x==0, y==0
Thread 1
r1 = x;
y = r1;
print(r2);

Thread 2
x = 1;
r2 = y;
print(r2);

Figure 18 - After second HotSpot VM transformation

However, Figure 17 and Figure 18 show the results of applying legitimate compiler transformations. The code shown in Figure 18 can print 1 when interleaving semantics are applied.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>SC</th>
<th>JMM</th>
<th>AltJMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace-preserving</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Memory access re-order</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Remove read after read</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Remove read after write</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Remove irrelevant read</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Introduce irrelevant read</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Remove write after write</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Remove read after write</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Roach-motel re-order</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>External action re-order</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Figure 19 - Validity of transformations (Sevcik and Aspinall 2008)

These authors proposed a weaker legality test that would allow this transformation. The characteristics of this legality test are summarised and compared with sequential consistency and the Java 5 memory model in the column headed AltJMM in Figure 19. The only difference is in the third row, where memory access re-ordering is permitted.
A number of much-used Java classes, notably the String class, use the technique of lazy evaluation of constants. There is a data race between threads that use instances of the String class to be the first to create the reference to a constant. Narayanasamy, Wang et al. (2007) argue that this is harmless. Boehm (2011) argues that even though the optimizations that might expose inconsistent behaviour are not presently implemented, they are not excluded from future development. So, data races are never "benign". This argues that reasoning about programs with data races should be avoided except where it is needed to detect them with the goal of removing them.

More recently, Lochbihler (2012) provided a novel unified formalism linking Java source code and its bytecode that built on the weak legality proposed by Ševčík and Aspinall (2008), but allowed for dynamic allocation. He provided an important series of formal proofs of theorems regarding multi-threaded Java programs, including:

- Data-race-free programs always provide sequentially consistent behaviour;
- Interleaving semantics are valid for all programs even those with data races.

This second result is important because it confirms the existence of consistent bounds on observed behaviour. These results are of great significance because the formal proofs were checked using the Isabelle/HOL proof assistant (Nipkow, Paulson et al. 2002) thus avoiding the errors and inconsistencies of previous work. With over 200 000 lines of Isabelle/HOL code, this is the largest known work in the registry of formal proofs. Lochbihler's work finally established a solid theoretical foundation for the Java Language Specification. We rely on his proofs to sustain the premise that if a Java Virtual Machine correctly implements the Java Language Specification and respects the access protocol of the acquire/release paradigm, then it will be free from data races.

The Java language has no features for associating shared variables with critical section guard conditions so that the integrity of the acquire/release paradigm rests with the correctness of the code. In this thesis, we deal with
this problem specifically through our innovative proposal for a thread-safe data store class, which we describe in Chapter VII. This class relies on the knowledge we acquired through the study of efficient de-limiter patterns, which we document in that chapter. Another approach with the same intention is described in (Demange, Laporte et al. 2013).

II.1.4. Java implementations

The changes to the Java specification to define the improved memory model identified by Manson, Pugh et al. (2005) were documented in Java Specification Request (JSR) 133 (Pugh 2004). The specified memory model defines how it should behave but leaves open how the Java Virtual Machine should achieve that behaviour on particular target CPU architectures. To guide the developers of Java Virtual Machines, Lea published the JSR 133 CookBook (Lea 2008). The CookBook explains how memory fences should be used to implement the Java volatile construct. The discussion explains the need for fences in Store-Load, Store-Store, Load-Store and Load-Load edges between shared memory events, but dismisses as too difficult the possibility of identifying the prior or next event for any given memory event. It recommends instead that accesses to volatile variables should be implemented using the fences appropriate to a conservative "worst-case" assumption.

When Java 5 was released, most Java applications used x86 or SPARC as their target architectures. Accordingly, the CookBook correctly stated that:

"Many of these barriers usually reduce to no-ops. In fact, most of them reduce to no-ops, but in different ways under different processors and locking schemes. For the simplest examples, basic conformance to JSR-133 on x86 or sparc-TSO using CAS for locking amounts only to placing a Store-Load barrier after volatile stores."

The Web page that publishes the Cookbook was last updated in 2011 to include a Preface that makes reference to the work on hardware memory models. We have taken the view that the wider acceptance of the ARM and POWER architectures requires that the simple recipe described in the Cookbook should be re-considered.
Prior to Java 9, there was no mechanism available to a developer through which he/she could explicitly control the placement of memory fences. Java 5 provided a package of classes, such as AtomicInteger, that supported methods, like `compareAndSet()`, that provided atomic access to primitive variables. These methods mapped on to the UnSafe class that provided direct raw access to memory. Originally, the UnSafe class was hidden from developers, but these restrictions were soon circumvented. This represented a significant potential security loophole, so Java 9 introduced the VarHandle class designed to provide equivalent, but secure, features. As a by-product of this development, the VarHandle class included static methods for acquireFence, releaseFence, etc. whose semantics were intended to be similar to the fence methods defined in the C++11 specification (ISO 2014). Java Enhancement Proposal (JEP) 193 (Lea and Sandoz 2015) defined these new features, but left open to the developers of the Java Virtual Machine the way in which the fences would be implemented. In our investigations, we have considered the ways in which the VarHandle class methods might be used to implement alternative and perhaps more efficient critical section de-limiter patterns. We have then further considered how the implementation of these methods might be optimised within the Java Virtual Machine.

**II.1.5. Java Virtual Machine - components**

The standard Java Virtual Machine (JVM) is implemented as an interpreter and two HotSpot Just-in-Time (JIT) compilers, C1 and C2. The C1 compiler performs a translation of the bytecode of a method into machine code. It also plants profiling code within the generated instruction stream so that, after a sufficient number of repetitions, a "normal" execution path can be distinguished from the "abnormal" cases. The C2 compiler uses this information to guide the generation of a linear sequence of machine code instructions where the expectation is that, in a "normal" path, all branch instructions will "fall through" so that the processor executes the instructions sequentially. This provides the optimal execution pattern for
cached and pipe-lined processors. We show the relationship between these components diagrammatically as a schematic in Figure 20.

![Diagram of Java Virtual Machine components](image)

**Figure 20 - Java Virtual Machine components**

The C2 compiler is implemented in C++, but Java 9 includes the Java Virtual Machine Compiler Interface (JVMCI) (Rose 2016). This provides a standard interface between the JVM’s internal components and external compilers written in Java so that the latter could provide and extend the features of the C2 compiler. The JVMCI is not specific to any particular external compiler so that, at least theoretically, a variety of different research and production implementations might be made available as part of the Java Development Kit, or otherwise. The vendors of Java have, for some time, been developing such a compiler under the project name Graal, in collaboration with an academic institution. The project and its artefacts are described in a number of published articles (Duboscq, Stadler et al. 2013, Simon, Wimmer et al. 2015, Wimmer 2015, Eisl, Grimmer et al. 2016). When Java 9 was delayed, it was decided to incorporate Graal alongside the C2 compiler as part of the JVM issued with the Open Java Development Kit.

The Graal compiler begins by transforming the Java bytecode into an Internal Representation (IR). This IR takes the form of a graph with nodes and edges. This graph is optimised in various ways by Graal’s process *phases*. A phase acts on the nodes of the graph by invoking a method specific to the phase, which must be implemented by every node that participates in the phase. This is controlled by the use of Java interface definitions. When the high-level optimisations are complete, the graph is transformed into a Low-level Internal Representation (LIR). The LIR is still recognisably Java,
but many of the more complex constructs are mapped into patterns of more primitive level operations. These patterns are referred to as *snippets*. Finally, there is a low level transformation of the LIR into the machine code of the target architecture.

This structured internal architecture eases the task of adding new optimisation features to the compiler. It is merely necessary to modify a few existing nodes and add others to achieve the desired effects. The activation of this new code is achieved by invoking it within the phase-specific method associated with the phase in which the action should take place.

The Graal project includes a visualisation tool, *igv*, which presents an IR Graph as a diagram. The compiler intermittently dumps a copy of the current state of the IR so that the progress of the compilation can be followed as a sequence of diagrams. There is a second visualisation tool, *c1v*, available for the x86 architecture that presents an assembly-like listing of the generated machine code. These tools facilitate high-level de-bugging of extensions to the compiler.

The work of the Graal project is available open-source from (https://github.com/graalvm/). From our viewpoint, the most significant publications are a tutorial presentation (Wimmer 2015) and its associated video presentations, and the description of the Internal Representation (IR) (Duboscq, Stadler et al. 2013).

**II.1.6. Abstract Event Graph**

In her doctoral thesis, Alglave (2010) proposed a generic framework for reasoning about weak memory model executions. It used a global time axiomatic style inspired by the models for the Alpha and SPARC processors (Shasha and Snir 1988). However, it differed from that previous work in allowing relaxed rather than atomic memory accesses. This framework was subsequently referred to as an Abstract Event Graph (AEG). Batty, Memarian et al. (2015) observe that the AEG does not deal adequately with "out of thin air" values. However, this criticism does not affect its use in our work because the Java environment (Manson, Pugh et al. 2005) explicitly excludes the appearance of such values.
An AEG describes a program in terms of memory events. Each memory event is characterised by:

- direction - Write or Read
- location - the memory address being accessed
- processor - the processor that causes the event
- label - a label unique across all processors

So that, a write action to a variable \( \nu \) might be written as an event \((a)W\nu\) where \(a\) is the label of the event.

The total order relation, \( \rightarrow \), represents the written order of instructions for a particular processor. This relation is the order in which events would occur in a sequentially consistent execution.

The abstract event recognises the particular characteristic of memory events that the existence of the event is sufficient. The absolute value that is read or written is irrelevant. The only necessary characteristic of the value is its equality with another value, so that it is meaningful to say that a value read is the same as one that was written. Alglave showed that a set of such events together with the program order over those events soundly and completely represents the original program with respect to its effects on shared variables. We rely on this theorem as the basis for our analysis of critical sections and data races.

In a weak memory model, the program order does not, in general, reflect the actual execution order because of the permitted re-ordering of instructions. The program order relation cannot link events on different processors. To represent completely the inter-thread effects of an execution, Alglave defines other relations that can link events on different processors. The read-from relation, \( \rightarrow_{rf} \), links a read event and its preceding write event with the implication that the value read is that written by the preceding write event. The write-serialisation, \( \rightarrow_{ws} \), relation imposes a total order over write events. The from-read relation, \( \rightarrow_{fr} \), indicates that a read event receives the value of a memory location that existed prior to the execution of a particular write event.
Figure 21 and Figure 22 are quoted from Alglave's paper. Figure 21 shows the AEG that Alglave derives for a simple Litmus test program. Figure 22 shows one of the executions that may be derived from the AEG shown in Figure 21.

In examining thread interactions for sequential consistency, we have placed particular reliance on one of Alglave's proofs. She showed that, in searching for a lack of sequential consistency, it is not necessary to perform an exhaustive examination of all execution paths. It suffices to follow the style of Shasha and Snir (1988) and search for critical cycles. The structure of events is reduced to a graph whose links are the program-order relationships and competing-pair relationships so that the structure shown in part (b) of Figure 21 is reduced to the abstract event graph shown in Figure 23 where the mono-directional \( \rightarrow \), \( \rightarrow \), and \( \rightarrow \) relations are individually replaced by bi-directional \( \text{cmp} \) relations. This graph shows a number of cycles such as, (a), (b), (c), (a). Clearly, in a more extensive graph, cycles like that will participate in a number of other cycles. The term critical cycles describes the minimum set of cycles.
Alglave showed that if an AEG, such as that shown in Figure 23, is acyclic, then it is sequentially consistent. Conversely, breaking the cycles with memory fences restores sequential consistency.

Several writers (Nimal 2014, Shipilëv 2016) agree that the selection and placement of memory fences to restore sequential consistency is difficult and error-prone. Xiong, Park et al. (2010) estimate that 80% of all attempts at informal synchronisation are flawed. Nimal, whose work (Nimal 2014) is discussed here, agreed with this assertion and proposed the use of an algorithm to automate this process.

In his doctoral thesis (Nimal 2014), Nimal built on Alglave's work (Alglave 2010, Alglave, Maranget et al. 2014) by describing an algorithm for the automated selection and placement of memory fences to ensure the sequential consistency of multi-threaded programs. The selected fence instructions depend on the target architecture for the execution of the program. He demonstrated that, in the C/C++ environment, these instructions could be planted into the source code and re-compiled into executable code with guaranteed sequential consistency. He showed empirically that the degradation of the overall performance of a number of sample programs when processed in this way was acceptably small.

In his work, he documents a number of conditions that must be satisfied for the valid use of Abstract Event Graphs. They are:

a) The processor neither omits instructions nor inserts other instructions;

b) The control-flow-graph should be statically resolved;

c) The functions called should be statically resolved;

d) The threads running should be statically determined;

e) All addresses should be resolved.

Condition (a) implies that the AEG analysis must be performed after all other compiler optimisation phases. Condition (b) is a natural consequence of the
use of abstract events, but has scalability implications because of the potential path explosion. In-lining the code of invoked functions (methods) is the easiest way to satisfy condition (c). In practical use, condition (d) is satisfied by assuming that any code being investigated might be simultaneously executed against itself in multiple threads. Condition (e) presents difficulties for the Java environment, which we discuss more fully in section V.3 of Chapter V.

Nimal describes the automated selection and placement of memory fences as a sequence of processes:

1. **Derive a composite AEG from the source code that incorporates the AEG’s of all its active threads.**
   This uses tools that are peculiar to the C/C++ environment, so that our use of his work must provide a substitute for this step;

2. **Search for critical cycles.**
   Nimal uses his own implementation of Tarjan's algorithm (Tarjan 1972) that incorporates heuristics identified by Alglave (2010). We have provided a novel implementation of Alglave's heuristics by using the Java stream concept to facilitate a multi-threaded algorithm;

3. **Break the cycles in each thread.**
   Potentially, any given program-order edge between two events may participate in a number of cycles. For each such participation, the algorithm constructs an inequality that expresses the various ways in which the placement of a fence might break that cycle. Each fence type has an arbitrarily assigned cost that expresses the relative costs of the different fence types. All these inequalities, together with a constraint that the solution should have a minimum cost, are formatted and submitted for solution to an Integer Linear Programming (ILP) solver. This returns a result from which the types and placement of fences can be extracted. In our implementation we have used the Java ILP interface program (javailp.sourceforge.net) to invoke the SAT4J solver (sat4j.org).
Commenting on the scalability of his "musketeer" implementation of these algorithms, Nimal notes that the performance is limited not by the number of lines of code to be processed but by the number of successive branch instructions. An AEG with many branches produces a proliferation of cycles that must be searched.

We have made a substantial adaptation of Nimal's method to implement it within the Java environment, relying heavily on his theoretical proofs that his extensions to Alglave's work remain sound.

II.1.7. Escape analysis

Both the analysis for data races and the analysis for the selection and placement of fences require the full resolution of references to shared variables. Within the Java environment, it is normally assumed that the memory required to implement arrays and objects will be taken from heap storage. However, if it can be shown that such an object never escapes from the method in which it is declared, then it is possible to allocate space on the stack and to release this space when the method exits. This reduces the load on the garbage collector. Choi, Gupta et al. (1999) published an algorithm that detects the variables that have not escaped from a Java method. Their work was part of a larger effort to build a native Java compiler and so focussed on the identification of variables that cannot be in scope beyond the scope of the method. The documentation of the more recent Java releases states that this algorithm has been incorporated into the optimisations performed by the Java Virtual Machine's Just-in-Time (JIT) compilers. The challenge of following chains of alias references is hard, so Choi, Gupta et al. adopted the engineering compromise of arbitrarily curtailing the search of alias chains. It is possible for an optimising compiler to take this information and re-write the bytecode. Within the Java bytecode, it is easy to recognise access to heap variables because there are specific instructions types used for that purpose: GETFIELD, PUTFIELD, GETSTATIC and PUTSTATIC. However, from the viewpoint of our research, we cannot be certain whether any GETFIELD, PUTFIELD, GETSTATIC or PUTSTATIC instruction that remains after this escape analysis refers to a genuinely shared variable.
Accordingly, we have chosen to save the time that would be taken to execute the escape algorithm statically and we treat all access to heap variables as potential data races. This does not affect the soundness of our work though it does increase the likelihood of false positive reports.

### II.1.8. Analysis of parallel loops

Radoi and Dig (2015) specifically address the problem of detecting data races in programs that process Collections in a Single-Program-Multiple-Data (SPMD) manner using parallel loops.

The algorithm is described in seven steps:

1. **Pointer analysis using the WALA tool (WALA 2015).**
   This delivers an invocation graph, a control-flow graph for each method and a heap graph;
2. **Find potential data races.**
   Traverse the program representation to match the operands of instructions with heap objects;
3. **Find locksets that guard instructions.**
   Using Interprocedural, Finite, Distributive, Subset (IFDS) analysis (Reps, Horwitz et al. 1995);
4. **Filtering.**
   Eliminate invocations of thread-safe methods and identify invocations of unsafe methods. This step is based on an a priori classification of classes and their methods;
5. **Deep synchronising.**
   Eliminate data races between correctly guarded accesses;
6. **Bubble-up.**
   Propagate data races in library code so that the fault is reported in the invoking application code;
7. **Synchronising.**
   Finally eliminate any remaining data races that relate to correctly guarded accesses.

This research does not address the general problem of the detection of data races. It specifically focuses on the problem of data races that are caused
when a Collection is processed in iterative code that is executed in concurrent threads. It leverages the features of analysis code-sets, such as WALA and IFDS analysis, which are not dedicated to the detection of data races. This leaves open the opportunity for more precise analysis when improved versions of these code-sets are developed. However, as we discuss in Chapter IV, section IV.2.5, while that promise remains unfulfilled, there is little practical loss in adopting some simplifying approximations. By applying these approximations from the outset, we were able to merge many of the steps listed in this section into a single pass over the class files of a program. This had the expected benefit in improved scalability. The summarised abstract event graph (SAEG) notation, which we describe in section IV.3 of Chapter IV, reduces our analysis of the handling of a Collection to the analysis of a simple single conditional statement. This statement can then be analysed in a non-specific manner using our general-purpose race detection algorithm. Because we do not repeat the analysis of the lambda expressions for different elements of the Collection, we avoid the possibility of large numbers of false positive reports without the costs of a more precise analysis.

We acknowledge that our algorithm would be improved by the incorporation of the Filtering and Bubble-up techniques. However, Radoi and Dig published subsequent to the completion of our work in this area.

II.1.9. Thread-safe objects
Recently, Daloze, Marr et al. (2016) described a software architecture for the support of thread-safe objects for dynamically typed languages such as JRuby (Nutter, Enebo et al. 2011). Although the context and implementation is quite different, the arguments presented for their design of thread-safe mechanisms have great similarity with those we use in the design of our DataStore class. They argue that efficiency must be achieved by ensuring that read actions do not require synchronization.

II.2. Other related work
This second section deals with important work that is less directly related to our investigations. We begin by noting that our work has focussed on Java
and not specifically on other languages that use the Java Virtual Machine. The on-going research into reasoning about weak memory model execution is introduced in Section II.2.2. Finally, there is a brief discussion of the development process needed to extend the Java Virtual Machine for releases prior to Java 9.

II.2.1. Languages

We rely heavily on work that relates to the hardware memory models. However, although this work generally uses C or C++ for experimentation, our work has been focussed on the impact of their results on the execution of Java programs under the Java Memory Model, as implemented by the Java Virtual Machine. We note that there are languages, such as Scala (Odersky, Altherr et al. 2007), that compile to Java bytecode and are, therefore, executable through the Java Virtual Machine. We have excluded from our research the validation that our algorithms are compatible with any peculiarities in the bytecode generated by Scala compilers.

II.2.2. Reasoning about sequential consistency

Alglave (Bornat, Alglave et al. 2015, Alglave, Cousot et al. 2016) and Vafeiadis (Vafeiadis and Parkinson 2007, Dodds, Feng et al. 2009, Doko and Vafeiadis 2016) are working on independent approaches to reasoning about weak memory execution. As Alglave sadly observes, both deliver the ability to reason about weak memory execution, though it seems that there is only a minimal overlap between the two approaches. What they have in common is a strong mathematical foundation. This is one of the characteristics identified by Woodcock, Larsen et al. (2009) as an obstacle to the general adoption of formal methods for the specification of systems. Such methods included Semantic Denotation (Stoy 1977), the Z language (Spivey 1992) and VDM (Bjørner and Jones 1978). Recent research (Matichuk, Murray et al. 2015) has shown that the cost of using formal methods in the design of systems tends to rise quadratically with the size of the proof. In this thesis we have concentrated on a pragmatic approach to the detection of data race errors in typical Java programs.
We note that in his most recent paper (Doko and Vafeiadis 2016), Vafeiadis and his collaborator used as a test case a simple SpinLock based on CompareAndSet logic. They comment that in such a coding pattern every effort should be made to keep memory fence instructions outside the loop. This reduces the cost of unsuccessful contention. We address this matter specifically in Chapter VI.

II.2.3. Compiler intrinsics

In Java 8 and earlier versions, the direct memory access features of the Unsafe class were implemented through Java Virtual Machine compiler intrinsics rather than through the use of the Java Native Interface (JNI). This avoids the known costs of using the JNI. In a compiler intrinsic, the compiler specifically recognises a high-level construct, such as

    Unsafe.compareAndSet(param1, param2, param3, ...)

and directly replaces it with appropriate machine code instructions from the target hardware architecture. The OpenJDK Cookbook (Kasko, Kobylyanskiy et al. 2015) provides detailed instructions on how to take the Open Java Development Kit source code and modify it to introduce new compiler intrinsics or to extend existing intrinsics. In this way it is possible to build an experimental version of Java that incorporates new features. Prior to Java 9 this was the known mechanism for implementing memory fences.

The Graal project, described in section II.1.5, provides an alternative and Java-based way of achieving a similar result. The Graal compiler is officially supported for experimental purposes in releases from Java 9 onwards. We describe our use of the Graal project artefacts in detail in Chapter VI.
II.3. Summary

Our review of prior work revealed several avenues for research:

• A systematic investigation of the costs of different synchronisation techniques to validate the commonly held view that the synchronized construct is costly;
• An improved algorithm for finding data races in multi-threaded Java programs;
• A search for and evaluation of improved synchronisation techniques;
• A more efficient implementation of the Java Memory Model for weak memory model architectures.

In the next chapter we provide a description of our experimental work to evaluate the relative cost of different de-limiter patterns when used to synchronise the methods of standard Java Collections classes.
Chapter III Cost of synchronisation

"No-one drives to a different city just to buy a cup of coffee"
Anonymous.

More generally, we might say that the cost of regularly invoking a function should be commensurate with the cost of the work done within that function.

Suppose that the cost of invoking a function is $c$, and the cost of the useful work done, $w$. Then the total cost of achieving a task, $t$, is given by

$$ t = c + w \quad (1) $$

We might try to qualify the term "commensurate" by saying that the cost of invoking the function must be less than some reasonable fraction, $r$, of the total cost, where $0 < r < 1$.

$$ c \leq r \times t \quad (2) $$

Replacing $t$ gives us

$$ c \leq r(c + w) \quad (3) $$

which we can re-arrange as

$$ c \leq \frac{r}{(1-r)} w \quad (4) $$

As a concrete example, consider two processes $P$ and $Q$ running on separate processors linked by a communications link. If $P$ invokes a function of $Q$, then the corresponding cost $c$ will be of the order of tens of milliseconds. If, for example, $Q$ is a database server whose response time for SQL queries is of the order of hundreds of milliseconds, then the situation would be acceptable. The reasonable fraction is of the order of 11%.

Now consider the case where $P$ and $Q$ are two agents within a multi-agent system that is hosted on the same processor. As, for example, Enterprise Java Beans (EJBs), they would communicate over an Internet Protocol (IP) software stack using a Java 2 Enterprise Edition (J2EE) transaction processor, such as WebSphere, WebLogic or JBoss. The overhead of the inter-process communication, $c$, is about ten milliseconds. In this case, the number of instructions executed within $Q$ might easily be as low as 100 000 instructions, so that $w$ is of the order of 0.1 milliseconds. This is
tolerable only if we accept a design where about 90% of the processing power is consumed in overheads.

Finally, we consider the case where \( P \) and \( Q \) are implemented as threads within the same process that communicate through a shared object. The execution of the de-limiter patterns represents the overhead, \( c \). Examination of the code of the java/util/Vector class shows that its methods execute hundreds of instructions only where the method invoked involves an iteration over the elements of a large collection. Most of the methods of the Vector class execute only tens of instructions.

Let us assume that the work executed in a method, \( w \), is \( \approx 100 \) instructions and that the reasonable fraction, \( r \), is 20%. Substituting these values into the previous expression (4) gives us

\[
c \leq \frac{20}{(100 - 20)} 100
\]

This suggests that in this case \( c \leq 25 \) instruction equivalents. Where \( w \) is 10, a reasonable allowance for \( c \) reduces to 2.5 instruction equivalents. This is barely enough for a simple read or write instruction. Accordingly, we must conclude that the Vector class, as implemented, must spend the greater part of its time in synchronisation overheads.

The detrimental consequences of this and similar circumstances are:

- Even if the work performed within a critical section is small, the lock is held for a significant time. This increases the chance that other threads will contend for the lock;
- Increased contention strengthens the demand for "fairness". It becomes increasingly important that waiting threads are delayed only as long as it is "fair". In this context, we define "fairness" to mean that, if a thread is suspended awaiting the release of a lock, it will be given a "fair" chance of acquiring the lock when it is released. "Fairness" is often taken to mean that a thread cannot be locked out indefinitely by the arbitrary intervention of other threads;
- Extending the lock implementation to improve "fairness" increases its overhead.
This is a vicious cycle. Historically, many designers have tried, unsuccessfully, to devise informal synchronisation techniques to avoid these overheads.

We have the belief that there is a better solution based on the use of lower-overhead critical section de-limiters. This encourages the use of small critical sections that give low times for the holding of locks. This, in turn, reduces contention, which again reduces the overheads of the lock mechanism. This is a virtuous cycle.

In Chapter VII we describe the logical extension of this principle, the use of a lock-free DataStore class for sharing data between threads.

In the next section, we present evidence that the cost of synchronisation constructs is significant and that the direct invocation of different de-limiter patterns can yield significant performance benefits.

III.1. Measuring relative costs on x86 host

Here we investigate the relative cost of the synchronized construct when compared to other synchronization techniques, such as CompareAndSet. We report the results achieved by repeating previous experiments and explain how we have extended an open source benchmark framework to support a more extensive empirical examination of the relative costs of different synchronisation techniques.

III.1.1. Cost of synchronized

There is extensive anecdotal and experimental evidence that using the synchronized construct incurs significant overheads. The experiment reported by Thompson, Farley et al. (2011) used an earlier version of Java, so we repeated it with Java 8 and with an early release version of Java 9-ea. We confirm that the cost of a synchronized block is significantly greater than those of a corresponding un-synchronized block. We note here that our objective was to obtain relative values from a simple, robust experiment rather than expend greater effort in eliminating sources of perturbations.

Having successfully implemented a high-performance online betting system, Thompson, Farley et al. (2011) built a prototype transaction processing system written in Java that was intended to support real-time
financial transactions. They reported that the system, which used conventional Java Enqueue and Dequeue facilities, spent the majority of its elapsed time manipulating message queues. They subsequently developed the Disruptor class, which provides efficient message passing based on the use of a ring buffer. Through careful design they were able to organise multi-threaded access to this ring buffer that required write contention on only one shared variable. By using the Disruptor class they were able to implement a transaction processing system that achieved a throughput of one million transactions per second on commodity Intel hardware.

As a justification for their decision to develop the Disruptor class, they published the results of a simple experiment to measure the costs of the synchronized construct compared with a functionally similar use of CompareAndSet. We have repeated a similar experiment using contemporary versions of Java and find that, where there is no contention between threads, there is now no significant difference in overheads between the synchronisation techniques. Our simple Java test program starts a number of threads. Each thread then invokes a method a million times. The invoked method may be un-synchronised, synchronised using the synchronized construct, or synchronised using CompareAndSet. By using different combinations of these methods, we were able to compare the efficiency of the different synchronisation methods when there is no contention for the lock and again when there is contention.

![Figure 24 - Comparison of synchronisation techniques](image-url)
Figure 24 shows the elapsed times for fifty test runs. Each test run invokes the method one million times. The tests were conducted on a platform whose CPU chip has support for the simultaneous execution of four threads.

For three or more threads, the CPU utilisation of the test program did not rise above 250-260%, while the activity of the operating system rose to about 35%.

The results show significant scatter, but the important result is that, contrary to the original findings of Thompson, Farley et al. (2011), where there is no contention there is no significant difference between the overheads of CompareAndSet and synchronized. We attribute this improvement to the use of biased locking (Vasudevan and Salapura 2010). This technique involves observing where the program is being executed in a single thread and avoiding the unnecessary synchronisation overheads in that circumstance. The version of Java that we used in our experiments incorporates this technique (Dice 2001).

We conducted a further series of tests where two, three, four, five and six threads were in contention. As an example, Figure 25 shows the results from the test with three contending threads. Once again the graph represents the elapsed times in milliseconds for fifty runs of one million method invocations.

These results suggest that the overheads are significantly reduced by the use of CompareAndSet rather than synchronized. We tried to measure the overheads attributable to un-synchronised code but we were unable to
separate values for read and write operations from the background "noise". These results were obtained using a simple synthetic program. To answer the possible criticism that they do not represent what might occur in a more realistic circumstance, we repeated the comparison using the more sophisticated environment of the SynchroBench framework. In the next section, we describe the SynchroBench framework and then present the results we obtained by comparing the execution of synchronised and unsynchronised members of the Collection package.

**III.1.2. Using SynchroBench - our adaptations**

**III.1.2.1. SynchroBench**

To support our extensive evaluation of the relative performance of different de-limiter patterns, we have made use of a standard benchmark framework. Gramoli (2015) published a comprehensive comparison of the impact of different synchronisation techniques on concurrent algorithms. This paper also described Synchrobench, an open source micro-benchmark suite written in Java and C/C++ for multi-core machines. We have extensively adapted the Java variant of this benchmark framework to focus specifically on the impact of the use of different synchronisation techniques in implementations of the Lock interface when applied to the various standard Java Collections classes. The SynchroBench micro-benchmark (Gramoli 2015) was developed to support the comparative evaluation of different synchronisation techniques across a variety of data structures. Here we describe how we have adapted it to focus on the specific needs of our research.

The SynchroBench framework exercises different classes by demanding that they implement a particular interface specification. The framework then exercises each class in the same way by invoking the methods of this interface. To benchmark a particular algorithm implemented by a Java class, the researcher must modify the source code of the class so that it implements the required interface. Inspection of the code of the framework revealed that the methods it uses are the same as those specified by the standard `java/util/List` interface. Accordingly, we adapted the framework to rely on the `java/util/List` interface and showed that, with this adaptation, we could
exercise un-modified instances of the ArrayList class loaded from the Java 8 reference library and from the corresponding Java 9 early access library. More generally, we extended the features of the benchmark to allow the parametric specification of the use of the java/util/Collection interface with the generation of a mix of method invocations drawn from this more basic interface. This allowed us to test a greater variety of classes from the Collection package by including classes that did not support the java/util/List interface, such as TreeSet and HashSet.

The List parameter to the benchmark causes the execution of a mix of size, get and put methods. The Collection parameter causes the execution of a mix of add and remove methods.

The design of SynchroBench uses an iteration to repeatedly invoke a framework method that exercises the class under test by invoking its methods. The standard framework method provides a mix of operations by randomly choosing operations according to a parametrically specified proportion. For example, a particular test run may use an 80/20 ratio of adds to removes. Investigation of anomalies in the observed early results revealed that the Java Random class itself uses synchronisation. To eliminate this perturbation, we devised a less-accurate, though effective way of implementing a defined pseudo-random mix of functions and adapted our variant of SynchroBench accordingly.

Most of the classes in the Collections package of the Java Development Kit (JDK) are not thread-safe. Developers are advised that where thread-safe operation is required, they should provide a wrapper class that wraps each method in a synchronized block. For example, the add(E e) method of the ArrayList class might be wrapped in a method of a ThreadSafeArrayList class as shown in Figure 26.
class ThreadSafeArrayList extends ArrayList {
    ...
    @Override
    public boolean add(E e) {
        boolean result = false;
        synchronized(this){
            result = super.add(e);
        }
        return result;
    }
}

Figure 26 - Wrapper class example

As we wished to investigate the relative performance of different critical section de-limiter patterns against the same standard classes from the Collections package, we defined a generic wrapper class that invoked the framework method, which in turn invoked the methods of the List interface according to the defined mix of functions.

class Repeater implements Runnable {
    PayLoad payload;
    // payload is set in the constructor
    // stop is a static variable
    public static boolean stop = false;
    public void run(){
        while (!stop) {
            payload.payload();
        }
    }
}

abstract class PayLoad {
    void payload();
}

class RepeatNoSync extends PayLoad {
    void payload() {
        ballast();
    }
}

class RepeatSync extends PayLoad {
    void payload() {
        synchronized(this) {
            ballast();
        }
    }
}

Figure 27 - Sub-classes for de-limiter patterns
We then provided sub-classes, each of which implemented a different de-limiter pattern. The classes shown in Figure 27 show an example of the technique. The ballast method is the static framework method that implements the mix of method invocations. The original Synchrobench framework simply choke-feeds the class under test with the workmix of actions. This is unrealistic. It forces the situation where using multiple threads is detrimental to throughput.

![Figure 28 - No benefit from multi-threading](image)

The graph in Figure 28 shows the count of operations per millisecond for different synchronisation patterns as the number of threads is increased. In all cases, the number of operations per millisecond is lower than that achieved with a single thread. In such circumstances, it is pointless to use multi-threading. We have enhanced the framework to provide a dummy instruction load so that there is a realistic, and parametrically adjustable, balance between the synchronised and un-synchronised use of the CPU. By using this feature we were able to choose our test circumstances so that we have focussed our attention on those parts of the performance envelope where the use of multi-threading is beneficial.

Although it re-uses much of the design of SynchroBench, our framework incorporates different facilities. In particular, we have used the features of the Java language to provide a flexible and extensible package that benchmarks the performance of different de-limiter patterns against the same collections using the same work mix of functions.

In the next section we describe our use of this adapted framework to compare the cost of executing methods of the Vector class, with the corresponding cost of executing a similar mix of equivalent method
invocations against the unsynchronised, but functionally equivalent ArrayList class. The use of different but functionally equivalent classes avoided the need to make any alterations to the source code of these classes. The measurements were taken in a uniprocessor mode with a single thread executing each of the classes. From these measurements, we obtain an estimate of the relative cost of the `synchronized` construct when used in circumstances that more closely resemble those in real programs.

### III.1.3. Uniprocessor performance

Measuring the relative performance of the two classes in a uniprocessor circumstance provides a comparison of the overheads incurred by using synchronisation where none is required.

We measured the performance of the Vector class and that of the ArrayList class using the Nosync option of our adapted benchmark framework to leave synchronisation as an action to be taken within the methods of the class. This meant that the Vector class invoked synchronisation while the ArrayList class did not. We specified the Collection interface and the maximum number of threads as one. We performed a single test with 200 elements in the test collection. These results are summarised in the graph shown in Figure 29. This shows that for a functionally equivalent set of operations, the synchronised class incurs an overhead even though there is no possibility of contention.

![Figure 29 - Vector versus un-sync ArrayList](image)

In the next section we describe the results of our measurement of the relative performance of a number of different de-limiter patterns when compared to that provided by the `synchronized` construct.
III.1.4. Multi-threaded performance

In our experiments we used our adapted benchmark on an x86 host to examine the performance of the following de-limiter patterns:

- Sync - **synchronized** block;
- RLock - ReentrantLock methods;
- Flags - semaphore using **volatile** variables;
- CAS - CompareAndSet using an AtomicInteger instance;
- ANR - CAS with no re-entrancy provision;
- ANY - ANR with no Thread.yield();

The Sync pattern uses the **synchronized** construct to designate a block as a critical section. The RLock pattern replaces the use of a syntactic block with a region of code de-limited by invocations of the lock and unlock methods from the Lock interface. The Lock interface implementation used is the standard java/util/ReentrantLock.

The Flags pattern is a multi-thread synchronisation pattern based loosely on Peterson's mutual exclusion algorithm (Peterson 1981) with the shared variables declared using the **volatile** construct.

```java
public class CASLock implements Lock{
    private int nCount = 0;
    private final AtomicInteger ar = new AtomicInteger(0);
    @Override
    public void lock(){
        int self = (int)Thread.currentThread().getId();
        if (flag == self){}
        else{
            while(flag != 0){Thread.yield();}
            while(!ar.compareAndSet(0, self)){Thread.yield();}
            nCount++;
        }
    @Override
    public void unlock(){
        if (nCount > 0) {nCount--} else {nCount = 0}
        if (nCount == 0){
            ar.set(0);
        }
    }
}
```

Figure 30 - Java code for CAS pattern
It is included primarily as an illustration that this algorithm, and other similar algorithms that were devised before the invention of CompareAndSet instructions, cannot compete with algorithms that exploit the benefits of atomic access to variables. The CAS pattern implements the Lock interface using the methods of the AtomicInteger class so that the implementation is similar to that provided by the RLock pattern. This is shown in Figure 30. We provide this variant to investigate whether there are any performance differences between the implementation within the ReentrantLock class and a simple implementation that uses an AtomicInteger variable.

All these patterns make provision for handling re-entrant attempts to acquire a lock that is already held by the thread. This means that a class with synchronized public methods can freely invoke those methods internally without causing a deadlock. The Vector class makes extensive use of this feature. This technique is illustrated by the code shown in Figure 31. To allow for re-entrancy, the synchronisation code must keep a note of the identity of the thread holding the lock and a count of the number of times the lock has been acquired in a re-entrant manner.

```java
class LazyClass {
    int [] a;
    ...
    public synchronized int getSize(){
        return a.length;
    }
    public synchronized int getFirst(){
        if (getSize() < 1) return -1;
        return a[1];
    }
}
```

Figure 31 - Reentrant use of synchronized

This, in turn, means that the synchronisation code must establish the identity of the thread for every synchronisation action. We have established empirically that querying the thread identity incurs the overheads of synchronisation through a lock:add instruction (as described earlier in Chapter II, section II.1.2). These costs can be avoided by providing both public and private methods for code, such as getSize(), which is used
internally while being visible publically. The changes needed to achieve this are demonstrated by the code shown in Figure 32.

```java
class ImprovedClass {
    int [] a;
    ...
    public synchronized int getSize(){
        return getSizeHelper();
    }
    private int getSizeHelper(){
        return a.length;
    }
    public synchronized int getFirst(){
        if (getSizeHelper() < 1) return -1;
        return a[1];
    }
}
```

Figure 32 - Improved use of synchronized

This obviates the need for re-entrancy support with the potential for a more efficient implementation of synchronisation for that class.

In Figure 32, we showed that it is not difficult to re-code the methods to avoid the use of re-entrancy, though there may be some increase in the complication of the code. Such re-coding need not incur a loss of efficiency. Accordingly, we measured the performance of patterns that do not support re-entrant attempts to acquire a lock. The ANR pattern is a modification of the CAS pattern to remove the provision for handling re-entrancy. Our results show that this modification provides an improvement in performance.

In section III.1.3 we reported our measurement, in the Java 8 environment, of the relative cost of using the `synchronized` construct compared with that of using AtomicInteger compareAndSet(). The initial results from the full benchmark indicated that one of the causes of this high overhead is the attempt to ensure "fairness". Within operating system code, suspended processes are often enqueued on a First-In-First-Out (FIFO) queue. When the resource they need becomes available, only the process at the head of the queue is re-scheduled and the operating system often connects it immediately to the resource so that the possibility of conflict in accessing the resource is minimised. Where the expected waiting time is
long compared to the time taken by the en-queue and de-queue actions and the consequences of deadlock are serious (e.g. the machine locks up), this design is justified. However, this is not the case for many application systems particularly where there has been significant design effort to ensure that the critical sections are short. Under these circumstances, gains in performance are available through the use of other simpler de-limiter patterns.

The ANY pattern builds on the ANR pattern. It removes not only the provision for re-entrancy, but also the yielding of the thread where there is contention for the lock. This completely eliminates any code that tries to ensure that competing threads are treated "fairly" in obtaining access to the lock. In Figure 33 we show the Java code for the ANY lock implementation. The corresponding payload method replaces the normal lock.lock() by a spin-lock on lock.trylock().

```java
public class ANYLock implements Lock{
    private final AtomicInteger ar = new AtomicInteger(0);
    @Override
    public boolean tryLock() {
        return (ar.get() == 1 ? false : ar.compareAndSet(0,1));
    }
}
```

Figure 33 - Java code for ANY lock

The ANY pattern is effective because it reduces to a minimum the code executed within the lock implementation.

**III.1.4.1. Collection interface**

Using our extended framework, we conducted an extensive sequence of test runs of the Collection interface against different classes. In generating a workload for the Collection interface, the framework uses a parametrically specified mix of add and remove methods. This simulates the actions of a program that is continually changing the number of elements in a Collection. We used the ArrayList, TreeSet and HashSet classes and varied the Dummy and Ballast limit values over a wide range. The Dummy value specifies the size of the un-synchronised workload. The Ballast limit specifies the maximum number of elements in the collection. We plotted aggregate
throughput across all threads against the number of threads invoked. Where the throughput with a single thread was superior to the multi-threaded performance we noted that in that part of the performance envelope the overheads of synchronisation outweigh the useful work.

Figure 34 - Multi-threaded execution with dummy load

First we present the results achieved by exercising the ArrayList class. We set Dummy at the relatively high value of 7000 and the Ballast limit at the relative low value of 600. This high un-synchronised workload relative to the synchronised workload ensured that we were operating in the region where there is a benefit from multi-threaded working. The results for this sample test are shown graphically in Figure 34.

We then reduced both the Dummy and the Ballast limit values to 4000 and 500 respectively and obtained a similar pattern of performance across the increasing number of threads, but at a generally higher level of operations per millisecond. Figure 35 shows the results for this second sample test.
We performed a series of tests that varied both the Dummy and Ballast limit values and obtained graphs with a similar pattern. We found that varying the mix of add and remove methods had no significant effect on the results.

We observed that the ANY pattern shows the best performance. The custom patterns, Flags, CAS and ANR, come in the middle with Sync and, particularly RLock, providing the worst performance.

We then repeated the experiment using the TreeSet class with the same Dummy and Ballast conditions as used for the test shown in Figure 35. The ArrayList class incurs a complexity of $O(n)$ for its add and remove actions, where $n$ is the number of elements in the collection. The TreeSet class, which uses a binary tree representation, incurs a corresponding complexity of $O(\log n)$. This means that, for a TreeSet, the synchronised workload, represented by the add and remove actions, is significantly reduced, particularly for larger collections, as shown in Figure 36.

![Figure 35 - Multi-threaded execution, lower workload per operation](image1)

Under the test conditions, the amount of synchronised work is relatively small, so that the level of contention is low. The graph shows that the Sync implementation is well suited to these conditions, though the beneficial performance margin does not appear until the system is close to saturation. With five scheduled threads there was less than 1% idle time. A series of tests with increasing values for Dummy showed that as the relative fraction of synchronised work diminished, the margin between different
synchronisation patterns also diminished. This concurs with the obvious intuition that, if there is little synchronisation, its efficiency is immaterial.

When we reduced the Dummy load to 500, so that the probability of contention increased, we obtained the results shown in Figure 37.

![Figure 37 - TreeSet with increased contention](image)

Similar tests with the HashSet class did not reveal any additional facts, so we have not included samples of those results here.

These results show that, where there is any significant amount of contention, there is potential for increased throughput by using our ANY synchronisation pattern.

In summary, we found that over a wide range of realistic test conditions, including instruction mix ratios, number of elements in the collection, number of threads, and type of collection:

- The ANY pattern provides the best performance particularly where there is high contention;
- The Sync pattern is superior only where the contention is minimal.

We note that under conditions where there is little contention, the contribution of the locking cost to the total operational cost is small so that any variations are of little significance.

### III.1.4.2. List interface

We then performed a similar set of experiments using the List interface on the ArrayList and CopyOnWriteArrayList classes. As well as changing the number of elements in the collection and the dummy load, we varied our
percentW parameter to achieve a varied mix of read and write actions. We present here a few samples of output from the tests that illustrate our particular observations.

![Graphs showing operations per millisecond for increasing number of threads with different dummy loads.](image)

**Figure 38 - ArrayList, List interface, Collection of 500 elements**

Figure 38 shows the graphs obtained from running the ArrayList class with a collection of 500 elements. The instruction mix was 5% contains actions with 10% of the remainder being write actions. The graphs show the number of operations per millisecond for an increasing number of threads. Each data series line represents a particular style of synchronisation with the different graphs corresponding to the use of differently sized dummy loads. The graph for Dummy = 100 is typical of a situation where there is no benefit
in running more than one thread. The graphs for Dummy = 200 and 300 show the tipping point where the increasing non-synchronised load provides the opportunity for effective multi-threaded working. We note the superior performance provided by our ANY implementation. As the size of the dummy load is further increased, the degree of contention diminishes and, generally, the performance differences between the different synchronisation patterns become less marked. This confirms the overall performance effects observed for the test conditions described previously in section III.1.3.

For completeness, we repeated the sequence of tests that varied the size of the collection and the instruction mix as well as the size of the non-synchronised load against the CopyOnWriteArrayList class. However, although the absolute performance values were different, the results showed the same pattern of behaviour. Accordingly we have not included further samples of these results here.

**III.1.4.3. Commentary on observed scatter in results**

The graphs presented in the previous section are the aggregate of a large number of individual measurements. These individual results exhibit a substantial amount of scatter. This subsection examines the causes of this scatter and the steps taken within our experimental technique to manage the resultant influence on our observations. We describe the hardware and software on the two platforms that we used for experimental work during the course of our research. We then consider the characteristics of those platforms that may affect measured values and how the effect of these perturbations may be managed.

**Equipment**

Over the course of our work we used two platforms. Both of these were Apple MacBook laptops. A small amount of initial work was done using Java 7, but the bulk of our experimentation was conducted on platform 1 as specified below. Our experimentation with the Graal compiler, which we describe in Chapter VI, was done using platform 2.
**Platform 1**

Apple MacBook Air  
Intel Core i5 1.7GHz  
4GB main memory  
OS X 10.11.5  
Java 1.8.0_05

**Platform 2**

Apple MacBook Pro  
Intel Core i5 2.9GHz  
8GB main memory  
macOS 10.12.3  
Java 1.8.0_05 and 9-ea+157, 168

The Core i5 processor has a single chip with two cores each of which supports two threads. The operating system recognises this as four processes.

**Sources of perturbation**

Here, we consider the various characteristics of our test environment that tend to perturb our observations:

- Chip TurboBoost and Temperature control.  
  We found no convenient way to control this feature and accepted its operation as a contribution to the general level of "noise;"

- Page swaps.  
  We limited the effect of this activity by ensuring that the real memory was undersubscribed;

- Scheduled background processes.  
  The MacOS process for disabling daemons, including the mail daemon is manual and error-prone so we elected to accept the effect of this background activity as a contribution to the general level of "noise". Where we observed an obvious outlier result we attributed it to this effect and excluded it;

- Java garbage collection.  
  We managed this by providing an adequate warm-up time for each test run;
• Just-in-Time (JIT) compilation overheads.
  We managed this overhead through the use of a warm-up period.

• Stack and heap memory.
  We avoided dynamic requests for additional memory by using the command line parameters to set the maximum and minimum memory sizes to the same value, 1GB.

• Framework overhead.
  We note that the benchmark framework itself consumes resources that reduce the number of available processing cycles.

Managing scatter

The SynchroBench framework is set up to run the test circumstance for a warm-up period, which defaults to 5 seconds. Without re-loading the program it then repeats and times the run.

In our adapted variant, we set up a test circumstance, and then ran each of a number of different synchronisation implementations for a warm-up period and a test run that recorded the performance. In our early experiments, we observed that successive tests of the same implementation in the same circumstances showed significant scatter. Accordingly, we made a further adaptation to run a selected set of synchronisation implementations successively in a test run. These test runs were then repeated across a variety of different operating circumstances, recording the performance of each implementation in each circumstance. This meant that there was an improved chance that, in a given run, each synchronisation implementation would be subjected to the same degree of perturbation. We then repeated this pattern over a total of many hours.

If we were interested in extracting accurate values from this data, we would have subjected it to a Fourier analysis, filtered out the "noise" and so obtained accurate absolute values. This technique was used as early as the late 1960s (Harrison, Sandars et al. 1969). However, as our interest is only in the relative performance figures, we chose instead to adopt a less sophisticated approach. We inspected the results, deleted obvious outliers
and applied a commercially available implementation of a "least squares" algorithm to the rest.

**Conclusion regarding accuracy of results**

We note that there are a number of characteristics of our experimental platform that cause our results to show some scatter. We have reduced this by addressing some of the causes and have handled the remaining level of scatter by smoothing the results with a least-squares algorithm. We believe that this combination yields results whose accuracy is sufficient to justify the inferences that we draw from them.

**III.1.5. Examination of C2-generated machine code**

To investigate the causes of the reported behaviour, we used the features of the Java Virtual Machine (JVM) provided with Java 9-ea, the early access release of Java 9. We used the JVM command-line features to force the C1 and C2 Just-in-Time (JIT) compilers, described previously in section II.1.5 of Chapter II, to provide readable dumps of the x86 machine code generated during benchmark runs in an x86 environment.

-XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,*.payload.

We inspected the machine code generated by the de-limiter patterns and also the code generated for synthetic test methods containing arbitrary sequences of VarHandle fence method invocations.

We observed that:

- The optimiser had performed various pieces of instruction re-ordering;
- The C1 and C2 compilations were completed before the end of the benchmark warm-up phase so that the measured results represent the effect of executing the optimised code generated by the C2 compiler;
- The C2 compiler limits its in-lining of invoked methods so that many examples of optimised code comprised successive invocations of code fragments that had been transformed into methods. Very few examples had the memory fence instructions in-lined within the main
execution path, which made it difficult to re-construct an execution path for a critical section from the machine code;

- CompareAndSet operations are implemented using the \textit{lock:cmpxchg} instruction.
- Similarly, full fence operations are implemented using \textit{lock:add};
- No other fence instructions, such as \textit{mfence}, were observed;
- The C2 compiler optimises a succession of fences by eliminating fences that are not required within the x86 environment and by re-ordering full fences to optimise their placement in the generated instruction stream.

Inspection of the generated machine code shows that the underlying memory fences used by Sync and RLock are the same as those used by CAS. This suggests that the observed differences in performance should be attributed to the attempts to provide "fairness" and to cater for re-entrancy rather than to any relative difference in the cost of the fence instructions.

III.2. Summary

In this section we described our experimental investigation of the relative performance provided by different synchronisation techniques. Our results confirm the intuition that significant performance improvements may be achieved through the use of alternative de-limiter patterns. These benefits are most significant around the tipping point where the balance of synchronized and un-synchronized work causes the use of multi-threaded working to become beneficial.

Regardless of which de-limiters are used, it is still possible for code to include data races. In the next chapter, we describe our algorithm for statically detecting those data races.
Chapter IV Statically detecting data races

"I'm still going to spell it out because I'm thick.
Watch my arithmetic.
It's nowhere near as good as yours."

"Smiley's People"
John Le Carré

We begin by discussing how data races occur and define them formally in terms of memory events. We recapitulate the acquire/release paradigm and explain how data races can occur because of failures to respect the implicit protocol associated with its use. Then, we present a review of our reasons, introduced in Chapter II, for selecting the Abstract Event Graph (AEG) as the most appropriate abstraction for our work.

In section IV.2.1, we present the notion that different implementations of the acquire/release paradigm may use different patterns of instructions to de-limit the protected passages of code. We consider the commonly used delimiter patterns, weighing their relative advantages and disadvantages. This section also defines the implicit and explicit conditions that must be satisfied for a successful search for data races caused by errors in the use of the acquire/release paradigm. We then expose the problems posed by the attempt to satisfy those conditions in a program implemented in Java that performs a static analysis of Java bytecode. We discuss the use of techniques to mitigate those difficulties. We conclude this section by describing the detail of our algorithm for detecting data race errors of this type and explain the approximations we use to achieve adequate scalability. Our static approximations affect the soundness and completeness of the detection of data races. In section IV.2.6 we discuss these effects and consider their impact on the practical use of our prototype.

The AEG abstraction does not handle well, programs that access individual elements of Collections. As it eliminates all numeric values, accesses to individual elements, such as coll.get(0) and coll.get(1), cannot be distinguished so that all such accesses must be treated as equivalent. Section IV.3 provides a proof that our innovative summarisation notation can
be used to represent the actions of Java *streams* within an AEG. We explain how this allows the scope of our data detection algorithm to be extended to a significant sub-set of the programs that access Collections.

In section IV.4 we provide a description of our prototype implementation of the algorithm with an indication of the classes that we re-use in the work described in Chapter V and Chapter VI. Our prototype omits support for certain Java constructs that do not impact the overall validity of our algorithm. We discuss these limitations in section IV.5.

Finally, section IV.7 presents an evaluation of our prototype and a comparison of its performance against the well-known tools, JavaRaceFinder (Kim, Yavuz-Kahveci et al. 2012) and Chord (Naik, Aiken et al. 2006).

**IV.1. Causes of data races**

Data races occur where a number of concurrently executing threads make un-regulated access to the same shared memory location and at least one of these accesses is a write action. This is illustrated by the schematic displayed in Figure 39.

![Figure 39 - Processes with a data race](image)

We express this more formally with a number of expressions. To simplify these expressions, we employ the convention that, unless otherwise stated, the denotation given in an expression applies to all subsequent expressions.

Let $A$ denote the set of actions that can be performed on shared variables. We denote the set of shared variables by $V$ and an individual variable by $v$. Actions may be either *read* or *write*.

$$A \equiv \{\text{read}, \text{write}\} \quad (5)$$

$$a \in A \quad (6)$$
We refer to an access to a shared memory location as an event, \(e\), which we denote by a tuple comprising a unique identifier, an action and a variable. Events are derived from instructions. Let \(I\) denote the set of instructions in a class. Let \(C\) denote the set of classes used within a program. We denote an instruction, \(i\), by a tuple comprising: line number, class, opcode and operand. The line number within a class uniquely identifies an instruction.

Let \(\text{class}\) denote a class name that uniquely identifies a class so that

\[
\text{class} \in C
\]  

Let \(ln\) denote a line number so that

\[
ln \in \mathbb{N}
\]  

Let \(\text{ID}\) denote the set of all unique identifiers for instructions, so that

\[
\text{ID} \equiv \mathbb{N} \times \mathcal{C}
\]  

\[
id \equiv \langle ln, \text{class} \rangle
\]  

\[
id \in \text{ID}
\]

Let \(\text{OPC}\) denote the set of all opcodes. We are only interested in the restricted sub-set of these opcodes that refer to heap memory addresses, so that

\[
\text{OPC} \equiv \{\text{GETFIELD, GETSTATIC, PUTFIELD, PUTSTATIC, \ldots}\}
\]

and we define subsets of these

\[
\text{GET} \equiv \{\text{GETFIELD, GETSTATIC}\}
\]  

\[
\text{PUT} \equiv \{\text{PUTFIELD, PUTSTATIC}\}
\]

Let \(opc\) denote an instance of one of these opcodes, so that

\[
opc \in \text{OPC}
\]

For these opcodes, the operand identifies a variable, so that we denote an operand by

\[
\text{opd} \in V
\]

We denote an instruction by \(i\) as a tuple,

\[
i \equiv \langle id, opc, opd \rangle
\]

and provide the projection functions

\[
id_i ((id, opc, opd)) \equiv id
\]  

\[
opc_i ((id, opc, opd)) \equiv opc
\]  

\[
opd_i ((id, opc, opd)) \equiv opd
\]
Instruction identifiers are unique, thus we exclude duplicates from the set of instructions, so that
\[ I \equiv \left\{ i \in ID \times OPC \times OPD \mid \exists i_1 \in ID \times OPC \times OPD \land id_i(i) = id_i(i_1) \right\} \] (22)

We denote a memory event \( e \) by a tuple comprising a unique identifier, an action and the variable that is being accessed.
\[ e \equiv \langle id, a, v \rangle \] (23)

Let \( E \) denote the set of all such events that occur within a program, so that
\[ e \in E \] (24)

We define projection functions for the event so that
\[ id_e((id, a, v)) \equiv id \] (25)
\[ a_e((id, a, v)) \equiv a \] (26)
\[ v_e((id, a, v)) \equiv v \] (27)

We map instructions into events with a function, which we denote by
\[ e_i(i) : I \rightarrow ID \times A \times V \]
\[ \langle id_i(i), read, opd_i(i) \rangle \text{ when } opc_i(i) \in GET \]
\[ \langle id_i(i), write, opd_i(i) \rangle \text{ when } opc_i(i) \in PUT \] (28)

A data race occurs where there are unsynchronised events, with a write event and a read event accessing the same variable.

Let \( E_r \) denote the set of read events and \( E_w \), the set of write events.
\[ E_r \equiv \{ \langle id, a, v \rangle \in ID \times A \times V \mid a = read \} \] (29)
\[ E_w \equiv \{ \langle id, a, v \rangle \in ID \times A \times V \mid a = write \} \] (30)

Let \( E_v \) denote the set of events that access the variable \( v \)
\[ E_v \equiv \{ e \in E \mid v_e(e) = v \} \] (31)

We include only sets that include a read and a write. Let \( E_{vr} \) denote the set of read events that access a variable and \( E_{vw} \), the corresponding set of write events.
\[ E_{vr} \equiv E_v \cap E_r \] (32)
\[ E_{vw} \equiv E_v \cap E_w \] (33)

Let \( EDR_v \) denote the set of sets of events that access variable \( v \) and have at least one read event and one write event.
\[ EDR_v \equiv \{ e \in E_v \mid (\exists e_r \in E_{vr}, \exists e_w \in E_{vw}) \} \] (34)
So that we denote the set of events that cause data races by

\[ E_{dr} \equiv \bigcup_{v \in V} EDR_v \]  

(35)

and the corresponding set of variables on which data races exist by

\[ V_{dr} \equiv \{ v \in V \mid \exists i \in ID \wedge \exists a \in A \wedge \langle i, a, v \rangle \in E_{dr} \} \]  

(36)

The most common technique used to regulate access events is the acquire/release paradigm. A thread that wishes to access a shared memory location first acquires a lock. If the lock is unavailable, the thread either blocks waiting for the lock or abandons its request for the lock. When the thread has completed its access to the shared memory location, it releases the lock. The Lock mechanism ensures that only one thread at a time can acquire the lock. However, it cannot ensure that other threads respect that lock. All the threads for which the shared memory location is in scope can access it freely at any time, whether they have acquired the lock or not.

Accordingly, there are two classes of data race associated with the acquire/release paradigm:

- Accesses to shared memory locations by concurrent threads that have not acquired any lock;
- Accesses to shared memory locations by concurrent threads that have not acquired the particular lock that guards the memory location that they access.

We refer to code bounded by matching acquire and release actions as a critical section. We say that the critical section is de-limited by these actions.

Lochbihler (2012) provided a proof that, for Java programs running under the Java Memory Model, if a program is free from data races, it is also sequentially consistent. It is trivially easy to devise a program that demonstrates the counter-example that the converse is not true. Programs that use the acquire/release paradigm are free from data races only where they conform to the implicit access protocol of that paradigm. The Java language supports but does not explicitly recognise that protocol.
IV.2. Finding data races

Our algorithm for finding data races makes the assumption that the delimiter patterns used achieve the result of ensuring that uniprocessor conditions exist within a critical section. The investigation of whether particular patterns achieve this condition in the most efficient way is deferred to Chapter V.

IV.2.1. De-limiter patterns

This sub-section discusses the known and popular de-limiter patterns with their relative advantages and dis-advantages. These patterns, which are described in detail in the Java Language Specification (Gosling, Joy et al. 2005), include:

- synchronized blocks;
- implementations of the Lock interface;
- code that is functionally equivalent to the methods of the Lock interface, but which uses different patterns.

We discuss each of these in turn in the following sub-sections.

IV.2.1.1. synchronized blocks

The synchronized construct is implemented within the Java Virtual Machine (JVM) and completely automates the acquisition and release of a hidden lock associated with a particular object. It ensures that a thread can only execute the block of code if it successfully acquires that lock. Unsuccessful threads are suspended awaiting the release of the lock. The Java code to invoke this feature is shown in Figure 40.

```java
Integer a = new Integer(0);
synchronized(a){
    // code that does something
}
```

Figure 40 - Use of synchronized construct

This pattern has many advantages:

- The critical section is a syntactic block so that its de-limiters naturally conform to the lexical scope;
The Java Virtual Machine (JVM) takes responsibility for ensuring that the lock is always released irrespective of the way in which the code exits from the block e.g. by throwing an Exception;

The JVM organises the suspension and re-scheduling of threads waiting for the lock.

However, these advantages must be weighed against the significant difference between the cost incurred by the use of this construct and that incurred by the use of other synchronisation techniques. These comparative costs were investigated experimentally in Chapter III.

**IV.2.1.2. Lock interface**

This interface is formally defined by `java/util/concurrent/locks/Lock`. It defines two primary methods: `lock();` and `unlock()`. On return from the `lock()` call, the calling code may assume that it has acquired control of the lock and the resources that it guards. It is the responsibility of the calling code to release the lock by invoking `unlock()` when it has completed the execution of the code in its critical section. In addition to these two fundamental methods, the Lock interface defines other methods that give the calling code greater control over the actions taken when the attempt to acquire the lock is unsuccessful. The normal action of an unsuccessful `lock()` is to suspend the thread awaiting the release of the lock. The `tryLock()` method always returns immediately with a boolean return value. The `true` response indicates that the lock has been successfully acquired, the `false` response, the contrary.

The greater control provided by the Lock interface brings some disadvantages:

a) The invocations of `lock()` and `unlock()` are quite independent of the lexical structure. It is possible to build valid Java where there are execution paths in which the `lock()` and `unlock()` invocations do not form matched pairs. Unmatched pairs cause deadlocks. (They may also cause data races, though this depends on the implementation. Many developers adopt a defensive approach that simply ignores unnecessary `unlock()` invocations.)
b) The developer must take responsibility for considering all the ways in which code might leave a critical section.

As a simplified example of the case shown in a) above, we have Figure 41.

```java
Lock aLock;
int flag = 0;
int x;
...
 aLock.lock();
if (flag == 42) {
    x = 42;
} else {
    x = flag;
    aLock.unlock();
}
```

*Figure 41 - Invalid use of Lock interface*

To accommodate the case shown in b) above, the description of the Lock interface recommends that it should generally be used with the pattern shown in Figure 42.

```java
Lock aLock;
aLock.lock();
try {
    // critical section actions
} finally {
    aLock.unlock();
}
```

*Figure 42 - Capturing exceptions with Lock interface*

This complicated pattern must be used for every critical section. It provides an increased opportunity for coding errors.

**IV.2.1.3. Other de-limiter patterns**

It is possible to implement the acquire/release paradigm by using de-limiter patterns that provide the functionality of the Lock interface with directly implanted patterns of instructions, such as those provided by the Java 9 VarHandle methods. We consider this option in Chapter V. For the purposes of this chapter, it suffices that these de-limiter patterns provide sequential consistency in and around critical sections, and this chapter relies on that assumption.
IV.2.2. Pre-requisites for finding data races

In section II.1.6 of Chapter II, we introduced the concept of an Abstract Event Graph (AEG) that soundly reduces the execution of a program to a succession of memory events. In this chapter we rely only on this property as we make explicit the assumption that the de-limiter patterns are effective in providing sequentially consistent execution.

We find data races by searching for accesses to memory locations within code that must satisfy some explicit and implicit conditions that we summarise here:

- **Sequentially consistent execution** - within a critical section the code must be guarded by de-limiters that enforce mutual exclusion so that the reasoning can rely on the sequential execution within a single thread that is guaranteed by all contemporary processor architectures;

- **The instructions** - the code being analysed must be the code that is executed. The executed code must not contain additional instructions and must not omit any instructions;

- **Alias-analysis.** The identity of shared memory locations must be resolved. It must be possible to recognise whether two memory access events refer to the same location;

In the following sub-sections we consider these requirements more deeply.

**IV.2.2.1. Sequential consistency**

If we are to reason about the code contained within critical sections, we require the assurance that each of the critical section bodies is executing in a sequentially consistent manner.

Consider the simple message-passing (MP) pattern that might reasonably be used in a simple implementation of the Lock interface. We show an example of this pattern in the Java fragment shown in Figure 43. In this example we deliberately omit the **volatile** declaration of the variable \( v \) to demonstrate the effects of legitimate optimisations.
1: while (!stop) {
2:     while (v != 0) {Thread.yield();}
3:     // critical section
4:     r1 = x;
5:     v = 1;
6: }  

Figure 43 - Simple message passing (MP) fragment

An optimising compiler or processor that assumes a uniprocessor circumstance and does not guarantee sequential consistency might reason that there is no dependency between \( v \) and \( x \) and \( x \) is not changed by the process. Accordingly, it might choose to re-order line 4 before line 2.  
This would re-order the code shown in Figure 43 so that it appears as shown in Figure 44.

1a: r1 = x;
1: while (!stop) {
2:     while (v != 0) {Thread.yield();}
3:     // critical section
4:     // empty
5:     v = 1;
6: }  

Figure 44 - Re-ordered fragment

This transformation invalidates the intention of line 2, which is that the variable \( x \) is not read until it is in a consistent state as indicated by the flag variable \( v \). This is one of many counter-examples that demonstrate that without the fence actions provided by critical section de-limiters it is not possible to analyse for data races.

If we consider the code fragment shown in Figure 43, with the flag variable correctly declared as \texttt{volatile}, the corresponding AEG is shown in Figure 45. This notation was introduced in section II.1.6 of Chapter II.

\[RV\]
\[RX\]
\[WV\]

Figure 45 - AEG for simple MP fragment

In our analysis of this AEG we recognise \( RV \) and \( WV \) as de-limiters so that the AEG becomes transformed into a critical section with enclosed events.
IV.2.2.2. The instructions

Accessing memory locations is more expensive than accessing the registers of a processor. Accordingly, an optimising compiler or processor, again assuming a uniprocessor circumstance, might choose to cache the value of the variable \( v \) in a local register and transform the code. This transformation is shown in Figure 47.

```java
1: while (!stop) {
1a:   r2 = v;
2:     while (r2 != 0) {Thread.yield();}
3:   // critical section
4:     r1 = x;
5:     r2 = 1;
6: }
```

This has a number of potentially detrimental effects:

- Any external changes to the value of \( v \) that occur after line 1a will not be perceived by this process;
- Because the write action in line 5 is no longer propagated to variable \( v \) other processes cannot perceive the release of the critical section that was programmed in line 5.

These effects invalidate the logic of the lock round the critical section.

IV.2.2.3. Alias resolution

Suppose that we have two concurrently executing threads. One executes the method \( \text{foo()} \), the other the method \( \text{bar()} \) as shown in Figure 48.

```java
class c;
//Thread 1               //Thread 2
void foo(){             void bar(){
    c.x = 42;          c.x = 24;
}                     }

foo();                 bar();
```

Figure 48 - Alias fragment
Whether \textit{foo} and \textit{bar} access the same variable when using the reference \texttt{c.x} depends on what happens to \texttt{c} during the execution path that leads to the execution of the two threads.

\begin{verbatim}
Aclass c;
c = new Aclass();
//Thread 1
void foo(){
c.x = 42;
}

foo();

//Thread 2
void bar(){
c.x = 24;
}

bar();
\end{verbatim}

\textbf{Figure 49 - Independent objects}

If, for example, the execution paths are as shown in Figure 49, then the two references to \texttt{c} refer to different objects and there are two distinct variables called \texttt{x}.

Conversely, if the variable \texttt{c} has the same value in both threads, as illustrated in Figure 50, then there is a potential data race on its variable \texttt{x}.

\begin{verbatim}
Aclass c = new Aclass();
//Thread 1
void foo(){
c.x = 42;
}

foo();

//Thread 2
void bar(){
c.x = 24;
}

bar();
\end{verbatim}

\textbf{Figure 50 - Single shared object}

The problem is compounded where the object reference is passed through a number of alias variables in different execution paths.

\textbf{IV.2.2.4. Summary of necessary pre-conditions}

The necessary pre-conditions are:

- Sequential consistency
- The instructions
- Alias-resolution

It is well-established that, in general, the static analysis of a program to enumerate its execution paths is NP-hard (Horwitz 1997), and that, even for programs of finite size and limited complication, it is often computationally impractical. Accordingly, we must satisfy these conditions without an exhaustive exploration of the possible execution paths.
In the next section we explain our data race detection algorithm by contrasting it with that used in the Chord tool.

**IV.2.3. Comparison with Chord**

The Chord tool (Naik, Aiken et al. 2006) is one of the more significant examples of the application of static analysis techniques to the problem of finding data races in Java programs. We provide here a description of its algorithms and explain ours by highlighting the similarities and differences between the two systems.

**IV.2.3.1. Basic description**

The Chord tool distinguishes between *open* and *closed* programs. A closed program is one where there is a public `main` method from which all parts of the program may be reached. An open program is one where there are many public methods and, in general, to ensure that all parts of the program are reached it is necessary to invoke all the methods. The authors describe a harness synthesis algorithm to convert open programs into closed programs. We attempted a re-implementation of this algorithm that revealed that, as described, the algorithm has some deficiencies:

- It does not cope with method parameters that are Interface objects or abstract classes;
- It does not handle the initialisation of parameter objects that have parameterised initialisation methods. This problem is exacerbated where the parameter objects are Interface objects or abstract classes. Our "no-values" abstraction is not affected by the choice of concrete class to represent an Interface or abstract class, so we chose to rely on the manual construction of harness code to convert open to closed programs.

The Chord tool uses the SOOT framework (Vallee-Rai 2000) to transform Java bytecode into suitable internal data structures. Unfortunately, SOOT has not been maintained to support the Stack Frame Maps that were made mandatory in Java 8. We wished to exploit the Java 8 language features within our prototype and to analyse programs that used the Java 8 *stream* feature and the Java 9 VarHandle methods. So, we chose to
use the ASM bytecode handling framework (Bruneton, Lenglet et al. 2002), which is actively maintained by its authors.

Using the ASM framework, we transform bytecode files into Class objects. Each Class object has a collection of Method objects. The ASM bytecode handler and our code together deliver methods with ordered lists of instructions. The instructions are classified as tokens so that a single pass over the list of instructions in a method can parse the list for loops and resolve each loop into a control-flow decision with two branches: one that represents the omission of the loop body and another that represents the execution of the loop body. This leaves each method with an Abstract Event Graph where the loops have been resolved. This simplification is valid because of our adopted "no-values" abstraction.

At the same time, we eliminate instructions that do not have an effect in our "no-values" abstraction, so that all that remains is conditionals, method invocations and accesses to heap variables. In section IV.2.5, we analyse the consequences of the loss of the modifying object reference for GETFIELD and PUTFIELD instructions.

The description of the Chord tool refers to memory events, but does not explicitly invoke the abstract event concept. We have chosen to identify our internal data structure as an AEG because of the formal proof that it soundly and completely represents the effect of the program with respect to its weak memory interactions (Alglave 2010).

Both the Chord tool and our system evaluate the call graph starting from a main method, whether pre-existing or constructed. Because we do not need to determine the actual values corresponding to the formal method parameters, we can construct a method's invocations by scanning its instruction list. This is in contrast with Chord that incurs the expense of a more accurate construction of the subject program's call-graph. Our call graph is effectively instantiated as a recursive stack of invocations of our method processing code. Where a method invokes many other methods we establish a separate thread to process each branch of the call-graph. Our algorithm carefully avoids the overhead of repeatedly processing classes and methods in these separate threads. The call graph ends when the method
being processed makes no invocations of other methods. We return back up
the recursion stack processing each method in turn for critical sections.

Within the Chord algorithm, the SOOT code delivers a list of pairs of
events that may refer to the same memory location. It then identifies data
races by successively refining its list of pairs of events. There is a rich body
of work concerned with the static analysis of C programs and, in particular,
Linux device-drivers. (Engler and Ashcraft 2003, Kahlon, Yang et al. 2007,
Young, Jhala et al. 2007, Kahlon, Sinha et al. 2009, Seidl and Vojdani 2009,
Pratikakis, Foster et al. 2011, Kahon, Sankaranarayanan et al. 2013, Vojdani,
Apinis et al. 2016). These works generally use a flow-sensitive analysis that
is computationally expensive. As Naik, Aitken et al. (2006) note, the
existence of the **synchronized** construct trains programmers to use critical
section de-limiters that are aligned with the lexical structure. The absence of
this construct from the C language makes flow-sensitive analysis of greater
importance. We experimented with a context-sensitive call-graph generation
approach similar to that described in (Grove, DeFouw et al. 1997) and
observed that it did not scale well. From this we take the lesson, that flow-
sensitive analysis must be avoided if our algorithm is to be efficient. The
work on points-to analysis (Steensgaard 1996, Milanova, Rountev et al. 2005,
Lhoták and Hendren 2006, Naik and Aiken 2007) shows that this analysis is
expensive with an expense that increases with the degree of precision. We
have deliberately chosen to adopt a radically simple approach to aliasing that
emphasises speed over precision. By making this decision early in the design
of our algorithm we were able to eliminate a great deal of computationally
expensive tasks whose output is subsequently considered superfluous. Our
goal was to detect data races errors in otherwise correct and conventional
multi-threaded Java code. In section IV.2.6, we examine the extent to which
commonly taught and used coding patterns are compatible with this
approach. We note that the current standard configuration of Chord uses an
unsound k=0 alias analysis that must suffer from many of the criticisms that
we discuss in that section.

In our search for efficiency, we adopt a different approach which
incorporates an effective k=0 alias analysis at the earliest stages of
processing. Starting at the bottom of the call graph we summarise each method as a set of critical sections each of which contains memory events. To reflect the control-flow graph within the method we categorise critical sections and their events using "must" and "may" criteria as explained in detail in section IV.2.5.2. The method summary also includes a "non-critical" section whose body is the events that occur outside of de-limited critical sections. The de-limiter pattern for a critical section defines the variables or objects that constitute the value of the guard for that critical section. For methods that make invocations, the summarised effect of those invoked methods is added to the method summary of the invoking method. The set of critical sections for any method is directly added to the method summary of the `main` method. The "non-critical" sections in the invoked methods are treated as follows. If the invocation of the method lies within a critical section then the events of the "non-critical" section are handled by forming the union of the set of events in that "non-critical" section with the set of events in the enclosing critical section. If the invocation of the method does not lie within a critical section then the "non-critical" section of the invoking method is re-formed by taking the union of the "non-critical" events in the invoking method with the events in the "non-critical" section of the invoked method. We treat all method invocations as if they are executing in a multi-threaded environment, until we encounter a method whose class implements the `Runnable` interface. This indicates the limit of the multi-threaded context. We add the "non-critical" section to the list of critical sections instead of propagating its instructions upwards. This implements the rule that, in a multi-threaded execution context, a data race can exist between synchronised and un-synchronised instructions. We also add the instructions in the "non-critical" section into a program-level "non-critical" section. This is used to implement the rule that, in a multi-threaded execution context, there may be data races between instances of the same un-synchronised instruction that are executed in different threads.

The ultimate result of this process is the production of a method summary for the `main` method that includes a single "non-critical" section and a set of critical sections. We then form the set of variables guarded by
each guard value and thence find data races as those variables that occur
under more than one guard. We consider the cases of nested critical sections
and define the way in which they are handled by our algorithm in section
IV.2.5.3.

One of the problems that our static analysis must solve is the
recognition of critical section de-limiter patterns. We have chosen to achieve
this by parsing the instruction stream for known de-limiter patterns. This is,
obviously, limited to knowledge of the patterns that are used. It cannot
recognise unknown patterns. There is a body of work that relies on the use of
annotations to identify critical section de-limiters (Flanagan and Freund
et al. 2006, Flanagan and Freund 2007), and Chord uses annotations to avoid
false positive reports associated with harness code. We have assumed that,
as is generally true in industrial circumstances, the source code is not
available and the starting point must be the bytecode contained in the .class
files. Accordingly, we have chosen to avoid the use of annotations in our
prototype. We do not believe that the more extensive approach presented in
(Chen, Lu et al. 2013) offers benefits commensurate with its costs.

We hold intuitively that the approach chosen by Naik, Aitken et al.
icurs unnecessary costs and that our approach yields a more efficient
solution. In the next section we discuss the characteristics of the two
algorithms that will affect the respective analysis times.

IV.2.3.2. Complexity analysis

Our process for traversing the call-graph involves a single pass over all
the instructions in a program. The complexity is $O(n)$ where $n$ is the size of
the program as the number of its instructions. The process of successively
merging lower-level method summaries into higher-level summaries and
ultimately into the method summary for the main method involves union
operations on sets. In the case of List collections, this simply involves adding
copies of the pointers to objects to the end of the List. The complete effect of
this process is that the "non-critical" sections of every level except the
highest are copied into the summary of the immediately higher level. If the
call-graph is m levels deep, then, in general, all "non-critical" sections are copied m/2 times. Depending on the design of the program, m is some fraction of n so that our process for constructing the top-level method summary is of time complexity, O(n^2). The number of critical sections depends on the design of the program but will be some small fraction of n. Our experiments confirm this assertion. The average number of events in a critical section, will depend on the design of the program, but in programs constructed in accordance with the best principles will be small. This means that the space-complexity of the top-level method summary is O(n) with an absolute value that is a small fraction of n.

If s is the size of the top-level method summary as the aggregate of all the events in its critical sections, together with the number of events in the "non-critical" section, the transformation of these sections into the set of guards with their list of guarded variables has time-complexity O(s). In general, the set of guards will be smaller than the set of critical sections.

Finding data races from the set of guards requires taking the guards in pairs and finding the intersection of their sets of variables. The selection of pairs from the set of guards has time-complexity O(g^2) where g is the number of guards and, depending on the algorithm used, the intersect function has time-complexity O(v^2) or O(v×log v), where v is the number of variables guarded by a guard. This depends on the implementation of these sets as lists, tree-sets or hash-sets. These operations do not scale well. However, we contend that these high-complexity operations are applied to only a small number of guards and variables. The process of deriving these guards and their variable sets scales adequately with program size.

In contrast, the Chord tool begins by using the SOOT framework to deliver pairs of memory events that appear to refer to the same variable. This has a space-complexity of O(e^2) and a time-complexity of O(e^2) or, at best, O(e×log e), where e is the number of events in the program. Once again the cost of finding a variable access in an existing set depends on the implementation of the set. As in our algorithm, e is a fraction of n. The results reported for the tool (Naik, Aiken et al. 2006) show that the numbers of Original Pairs delivered by the SOOT framework do indeed rise as some
increasing function of the number of LOC in a program. The tool then refines the list of pairs by applying call-graph construction, alias analysis and thread-escape analysis. Finally, lock analysis is applied to reduce the pairs of events to those that actually cause data races. Conversely, our algorithm immediately divides the code of the program amongst a set of critical sections so that each event is assigned to a single critical section. As an example, the Vector class would reduce to fifty-eight critical sections. All these sections would have the same guard. Although not all instructions are memory events, some LOC may generate more than one event. Accordingly we assume that the total number of events is well approximated by the number of LOC in the program. In the Chord tool, reachability and alias analysis are applied together during the call-graph construction. These analyses together dramatically reduce the number of pairs. However, we contend that our analysis offers superior efficiency because it records only the events themselves rather than their interactions with other events. Indeed, our algorithm searches for circumstances where a variable is accessed in a controlled fashion. Only when we have found such circumstances do we impose the criterion that requires a write event. This defers the processes with high complexity to a stage where the number of objects to be processed has been reduced to a minimum. In the next section we examine how well our algorithm copes with commonly taught coding patterns.

IV.2.4. Commonly taught patterns
The popular textbook (Magee and Kramer 1999) provides sample programs that demonstrate different coding patterns for achieving concurrency in Java programs. All these examples are coded as Applets so that the student may be given a graphic demonstration of the execution of the code. We have selected four programs: Carpark, Garden, Bridge and SpaceInvaders. These programs use either synchronized methods or blocks. We tested our prototype in two different ways. First we edited the Garden program to remove the graphics. This gave a very short program that just exposed the
concurrency logic. As expected we received false positive reports from un-synchronised access to a counter implemented as an instance variable.

In Figure 51, we show the code of the Counter class. This class is separately instantiated in two different threads so that in fact, there is no data race. However, when the code is changed to make the two threads use the same counter instance, the false positive becomes real.

```java
class Counter {
    int value=0;
    Counter(){
    }
    void increment() {
        int temp = value;   //read[v]
        value=temp+1;       //write[v+1]
    }
}
```

**Figure 51 - Counter class**

This possibility is recognised in the example with the presence of a second class as shown in Figure 52.

```java
class SynchronizedCounter extends Counter {
    SynchronizedCounter(){
        super();
    }

    synchronized void increment() {
        super.increment();
    }
}
```

**Figure 52 - SynchronizedCounter class**

We then ran our prototype against the un-modified Garden, Bridge and SpaceInvaders classes that included all the graphics handling. In the case of SpaceInvaders this involved the processing of 12047 SLOC that was achieved in 1.8 seconds on our Platform 2 using Java 8. As we report in section IV.7.3, the processing of the same program by Chord took just over 1 minute. A more extensive treatment of our experimental results is contained within section IV.7.

Our search for efficiency has a cost in a lack of soundness and completeness. In the next section we examine which styles of program our algorithm can handle effectively and where the approximations that we use impair the soundness and completeness of our approach.
IV.2.5. Approximations for scalability

To avoid the path explosion implied by the full expansion of all execution traces in a program, our implementation uses three approximations:

- Treating class instance variables as if they are **static** variables;
- Classification of critical sections and events as "must" or "may";
- Simplified resolution of assignment of events to critical sections;

In this section we discuss the beneficial effects of these approximations on the scalability of our prototype implementation. In section IV.2.5.1 we discuss our "all static" approach to the resolution of the addresses of variables. We enforce an alignment of critical sections with the syntactic block structure. This is explained in section IV.2.5.2 where we define our "must" and "may" classifications and explain how they mitigate the loss of information associated with our handling of conditionals. In the final section, IV.2.5.3, we deal with our approach to the handling of nested and overlapping critical sections. We discuss the effect of all these approximations on the soundness and completeness of the detection of data races in section IV.2.6.

IV.2.5.1. "All static" approach to variables

We use an abstraction on the operands of bytecode instructions that access heap memory variables. This allows us to improve scalability by computing the effect of these instructions without reference to the preceding execution path.

As discussed in section IV.2.2.3, the correct resolution of the operands of GETFIELD and PUTFIELD instructions requires the expansion of the execution traces and an evaluation of the object reference that exists on the top-of-stack when the instruction is executed. Conversely, by treating these instructions as if they were instances of GETSTATIC and PUTSTATIC respectively, the implementation can derive the identity of the accessed variable statically from the operand of the instruction. This, in turn, means that the variable accessed by an event can be directly derived from the instruction. It does not require knowledge of the trace that leads to the execution of the instruction. This is particularly important, because it means
that, in the abstraction created by this approximation, the identity of the variable does not depend on the resolution of the potential alias chain through the actual parameters of the method invocation chain back to the creation of the variable. The identity of the variable is established solely on the basis of the instruction itself. The method invocation chain from the starting method identifies the set of methods and their invocations that form the program. Conversely, the events within each method are resolved by a static and independent analysis of each method.

**IV.2.5.2. Classification of critical sections and events as "must" or "may"**

We enforce the alignment of critical sections with the syntactic block structure. This allows us to summarise the effects of each block independently and thence, to merge these effects into a single method-level summarisation. This makes a significant contribution towards scalability. We classify critical sections and events as "must" or "may" to mitigate the loss of information that would otherwise occur.

Although de-limiter patterns are not constrained to implement the Lock interface, they must provide equivalent functionality. Accordingly, in discussing the interaction between critical sections and syntactic block structuring we shall use the symbols \( L \) and \( U \), standing for `lock()` and `unlock()` respectively, to denote the start and end de-limiters of a critical section.

![Figure 53 - "May" event](image)

The **synchronised** construct is syntactically tied to the block structure and the JVM takes responsibility for correctly releasing the implicit lock if
exceptions are thrown. Conversely, the $L$ and $U$ de-limiters may be placed anywhere.

Within a method, the control-flow is a directed graph. The possible execution traces corresponding to this graph form a hierarchy. We cater for cases similar to that shown in Figure 53 by categorising $Wx$ as a *may* write action.

Events that *must* occur are those that occur in every trace. Events that *may* occur are those that do not occur in every trace. Formally, let $TR$ denote the set of all traces, $E_{tr}$ the set of events that occur within an trace, $tr$, $ETR$ the set of such sets, $E_{must}$, the events that *must* occur, and $E_{may}$, the events that *may* occur.

$$ETR \equiv \bigcup_{tr \in TR} E_{tr}$$  \hspace{1cm} (37)$$

$$E_{must} \equiv \bigcap ETR$$  \hspace{1cm} (38)$$

$$E_{may} \equiv E \setminus E_{must}$$  \hspace{1cm} (39)$$

Similarly, we cater for cases like that shown in Figure 54 by categorising the critical section as *may*.

![Figure 54 - "May" critical section](image)

Let $CS_{tr}$ denote the set of critical sections that occur within a trace, $tr$, and $CSTR$, the set of such sets.

$$CSTR \equiv \bigcup_{tr \in TR} CS_{tr}$$
Let $CS_{\text{must}}$ denote the critical sections that must occur, and $CS_{\text{may}}$ denote those that may occur, then

$$CS_{\text{must}} \equiv \bigcap \ CSTR$$

(40)

$$CS_{\text{may}} \equiv CS \setminus CS_{\text{must}}$$

(41)

(42)

However, the graph shown in Figure 55 presents more of a challenge. If the left-hand conditional branch is followed, then $Wx$ clearly belongs within the critical section. However, if the right-hand branch is followed, the critical section is closed within the branch, so that $Wx$ is no longer guarded. Depending on the implementation of $U$, the attempt to close an already closed critical section may be handled defensively, or may cause a runtime exception.

If a particular memory event may be guarded or not, depending on the actual execution path, this is clearly a data race circumstance that we could detect by an exhaustive search of the execution paths. We have chosen to adopt the pragmatic engineering approach, which is to enforce the rule that $L$ and $U$ must conform to the rules of syntactic block structuring.

![Figure 55 - Invalid de-limiter usage](image)

The prototype detects this situation and immediately reports it as an error. This means that the example shown in Figure 55 is reported as invalid because the $U$ within the conditional branch has no preceding $L$ within the
same block. This has two advantages for the implementation of our static analysis:

- In this respect, each conditional has a contained effect. Critical sections are either wholly internal or wholly external to any path. This avoids the usual exponential processing costs associated with successive conditionals;
- The rule follows the pattern for well-formed language grammars and is, therefore, compatible with the use of a grammar-directed parser.

With the exception of methods that form part of the Lock interface, we apply the same principle to whole methods. A critical section is either wholly internal or wholly external to a method.

The use of "must" and "may" classification preserves some of the information that would otherwise be lost as the result of our handling of control-flow graphs. In qualitative sense, critical sections and events that "must" occur are more likely to occur than those that "may" occur.

**IV.2.5.3. Nested and over-lapping critical sections**

To facilitate the discussion of these cases, we extend the notation for delimiters. Let $L_{gi}$ denote the start de-limiter for the $i$th instance of a critical section with the guard condition $g$ and $U_{gi}$ denote the corresponding end delimiter.

When all the method invocations and control flow branches have been handled, the remaining instruction sequence might contain the sequence of delimiters $L_{11} L_{21} U_{21} U_{11}$. However, our grammar-driven parser cannot distinguish this sequence from the sequence $L_{11} L_{21} U_{11} U_{21}$. It always interprets the sequence as nested critical sections.

Now let us consider the sequence

$L_{11} Rx L_{21} Ry U_{21} U_{11} \ldots L_{12} Wx Wy U_{12} \ldots L_{22} Wz U_{22}$.

It is intuitively clear that the intention is that $Ry$ belongs within $g_1$ and that $L_{21} U_{21}$ is redundant.

But what if the sequence were

$L_{11} Rx L_{21} Rz U_{21} U_{11} \ldots L_{12} Wx Wy U_{12} \ldots L_{22} Wz U_{22}$.

Here it is intuitively clear that $Rz$ belongs within $g_2$. 
It is possible to devise an attribution algorithm that will correctly separate these cases. However, we have, instead, adopted the pragmatic engineering compromise that simply attributes memory events to the immediate critical section within which they occur. We make no attempt to "promote" these events to any enclosing critical sections. This follows our conviction that critical sections should be kept as short as possible. Nested critical sections must tend to increase the occupancy of the lock associated with the outer critical section with the consequential increase in contention. It also provides an opportunity for deadlocks to occur where different parts of the same program do not acquire the locks in the same order. It is good coding practice to avoid these pitfalls and we have adopted a simple approach rather than catering fully for an approach that is error-prone.

**IV.2.5.4. Summary of approximations for scalability**

We use three approximations to improve the scalability of our prototype:

- "All static" approach to alias resolution;
- Classification as "must" or "may";
- Simplified handling of nested critical sections.

These three measures, applied together eliminate the interactions between methods and between successive conditional statements within the same method. This enables an efficient implementation. Once the set of methods used in a program has been identified, the effect of the whole program can been found by merging the effects of the methods. The effect of each method can be determined individually and, if desired, in parallel with the analysis of the other methods.

These approximations reduce the complexity of our algorithm to $O(n)$ where $n$ is the number of lines of code in the program. We discuss the effect of these measures on the soundness and completeness of our implementation in section IV.2.6.

**IV.2.5.5. Handling arrays and Collections**

Using abstract events, we can distinguish neither different elements of the same array nor different elements of the same collection. This yields the sound but worthless result that if multiple threads are accessing the same
shared array or collection, then all events that reference the array or
collection are potential data races. Fortunately, the Java 8 streams feature
provides an avenue out of this difficulty, which we discuss in section IV.3.

IV.2.6. Soundness and completeness of data race detection
To reduce the processing time required by our prototype implementation we
have adopted several palliative measures:

• Treating class instances as static variables;
• Method-level must and may summaries;
• Block-structuring of critical sections;
• Simplified handling of nested critical sections;

Taken together these measures mitigate the effects of path explosion. This
has the expected beneficial effect on scalability.

In this section we examine the extent to which these measures affect
the soundness and completeness of the analysis. We note that any assertion
of soundness and completeness relies on Alglave’s proof that an AEG soundly
and completely represents the effect of executing a program (Alglave 2010).

IV.2.6.1. Effect of static approximation of variables
In this sub-section, we consider the effect of conflating instance variables
with a corresponding static variable.

For the purpose of this analysis, we make the following definitions. Let
name denote the declared name of a variable within a class. Let class denote
the name of the class. Let ci denote the unique identifier of an instance of the
class. Let var denote the combination of class and name. Let C denote the set
of all classes in a program, and N the set of all names of variables.

\[
name \in N \tag{43} \\
\text{class} \in C \tag{44} \\
\text{var} \equiv \langle \text{class}, \text{name} \rangle \tag{45}
\]

Let V denote the name combinations that represent variable names so that

\[
V \subseteq C \times N \tag{46} \\
\text{var} \in V \tag{47}
\]

Let vs denote a statically declared variable, and VS the set of such variables.

\[
\text{vs} \in VS \tag{48}
\]
\( VS \subseteq C \times N \) \hspace{1cm} (49)

Let \( v_{ci} \) denote a class instance variable as a tuple comprising class identifier, \( ci \), and variable name, \( var \).

\[ vci \equiv (ci, var) \] \hspace{1cm} (50)

Let \( VCI \) denote the set of class instance variables in a program. By the rules of the Java language, there cannot be a class instance variable and a static variable in the same class that share the same name.

\[ vci \in VCI \] \hspace{1cm} (51)

\[ VCI \equiv \{(ci, var) \in \mathbb{N} \times V \mid \nexists var \in VS\} \] \hspace{1cm} (52)

Let \( gv \) denote the relationship between the flag variable of a guard and a guarded variable and \( GV \) the set of such relationships

\[ gv \equiv (vg, v) \in V \times V \] \hspace{1cm} (53)

\[ gv \in GV \] \hspace{1cm} (54)

Let \( agv \) denote our approximated equivalent of \( gv \), with mapping functions

\[ agv \equiv (avg, av) \] \hspace{1cm} (55)

\[ absv : V \rightarrow V \]

\[ \langle v \rangle \text{ when } v \in VS \]

\[ \langle var \rangle \text{ when } \langle ci, var \rangle \in VCI \] \hspace{1cm} (56)

\[ abs : GV \rightarrow AGV \]

\[ \langle gv, v \rangle \rightarrow \langle absv(vg), absv(v) \rangle \] \hspace{1cm} (57)

Let \( AGV \) be the set of approximated equivalents, so that

\[ AGV \equiv \{avg \in VS \times VS \mid avg = abs(gv) \land gv \in GV\} \] \hspace{1cm} (58)

**Case 1**

If both \( vg \) and \( var \) are static, then there is no approximation and the identification of events is sound and complete.

**Case 2**

If \( vg \) is static and \( var \) is the name of a class instance variable, then, with regard to completeness, we have a counter-example. Let \( var_0, var_1, \text{ etc.} \) denote different instances of the variable \( var \) so that

\[ var \in C \times N \] \hspace{1cm} (59)
\[ \text{var}_0, \text{var}_1 \in \{(ci, \text{var}) \in VCI\} \]  
(60)

Let \( u, v \) denote different guard variables, so that two distinct relationships may be denoted by

\[ gv_0 \equiv \langle u, \text{var}_0 \rangle \]  
(61)
\[ gv_1 \equiv \langle v, \text{var}_1 \rangle \]  
(62)

and there is no data race.

Our abstraction reduces the two relationships to abstract relationships \( agv_0 \) and \( agv_1 \), so that

\[ agv_0 = \langle u, \text{var} \rangle \]  
(63)
\[ agv_1 = \langle v, \text{var} \rangle \]  
(64)

There are two different guards with the same variable. This will give a false positive report of a data race.

**Case 3**

If \( vg \) is a class instance variable and \( \text{var} \) is static, then, with regard to soundness, we have a different counter example. Let \( vg_0, vg_1 \) denote different instances of guard variables with the same name \( vg \). Then, the list of relationships can include

\[ gv_0 \equiv \langle vg_0, \text{var} \rangle \]  
(65)
\[ gv_1 \equiv \langle vg_1, \text{var} \rangle \]  
(66)

This has two different guards with the same variable that ought to be reported as a data race. However, our abstraction will reduce this to

\[ avg_0 \equiv \langle vg, \text{var} \rangle \]  
(67)
\[ avg_1 \equiv \langle vg, \text{var} \rangle \]  
(68)

This will deliver a false negative that fails to report the data race on \( \text{var} \).

**Case 4**

If both \( vg \) and \( \text{var} \) are class instance variables, then, by the argument presented for Case 2, the abstraction will cause false positive reports and, by the argument presented for Case 3, the abstraction will cause false negative reports. This means that, in general, occurrences of this case must be identified and specifically excluded to avoid presenting false negative reports. However, we now examine the particular case where both \( vg \) and
var belong to the same class instance. This is important because, as we show in later in this section, this is an obviously useful coding pattern.

Formally, let \( vg \) and \( var \) be denoted by

\[
vg \equiv (ci_{vg}, class_{vg}, vg) \tag{69}
\]
\[
var \equiv (ci_{var}, class_{var}, var) \tag{70}
\]

then a guard relationship conforming to this specific circumstance is denoted by

\[
gv4 \in \left\{ (ci_{vg}, class_{vg}, vg), (ci_{var}, class_{var}, var) \in VCI \times VCI \mid ct_{vg} = ci_{var} \land class_{vg} = class_{var} \right\} \tag{71}
\]

Let \( GV4 \) denote the set of such relationships

\[
gv4 \in GV4 \tag{72}
\]

We denote some guards and variables as follows using subscript notation that conforms to the rules for the set \( GV4 \)

\[
vg_0 \equiv (ci_0, class_0, vg) \tag{73}
\]
\[
var_0 \equiv (ci_0, class_0, var) \tag{74}
\]
\[
vg_1 \equiv (ci_1, class_1, vg) \tag{75}
\]
\[
var_1 \equiv (ci_1, class_1, var) \tag{76}
\]

Then we denote two guard relationships, \( gv_0, gv_1 \) by

\[
gv_0 \equiv (vg_0, var_0) \tag{77}
\]
\[
gv_1 \equiv (vg_1, var_1) \tag{78}
\]

There are no data races as each variable instance has its own guard instance. Our abstraction reduces this to

\[
avg_0 \equiv (vg, var) \tag{80}
\]
\[
avg_1 \equiv (vg, var) \tag{81}
\]

Each of these abstract relationships correctly summarises all the relationships from which it has been abstracted.

In the section that follows we extend this analysis to consider the case where the guarded variable is not primitive.

\textit{Resolution of Cases}

In all cases, if the guard variable is primitive then it must be declared \textbf{volatile} so that changes to its value are correctly propagated across threads.
Conversely, if it is not a primitive class (which includes String), then the field must be declared **final**. This ensures that the guard object cannot be changed in one thread while it is being used in another.

Next, we consider what resolution actions are needed to deal adequately with the four different cases.

Case 1 requires no resolution action.

Case 2 is typical of the code fragment shown in Figure 56.

```java
class Case2 {
    // var is guarded by lock
    private int var;
    private final static Lock lock = new ReentrantLock();
    ...
}
```

*Figure 56 - Case 2 example code*

The probable intention is either that both `var` and `lock` are `static` or they are both not `static`. In either case, we take the view that the false positive report acts as a useful alert of a probable error.

As case 3 reports false negatives, we propose that a production implementation would include a specific check to report occurrences of this circumstance. We have been unable to envisage practical counter-examples that require the use of this pattern. A code fragment, typical of this circumstance, is shown in Figure 57.

```java
class Case3 {
    // var is guarded by lock
    private static int var;
    private Lock lock = new ReentrantLock();
    ...
}
```

*Figure 57 - Case 3 example code*

A typical coding pattern that would give rise to Case 4 is shown in Figure 58.

```java
class Case4 {
    // var is guarded by lock
    private int var;
    private final Lock lock = new ReentrantLock();
    ...
}
```

*Figure 58 - Case 4 example code*
class SynchronizedArrayList {
    private final ArrayList arr = new ArrayList();
    ...
    public synchronized boolean add(E e) {
        return arr.add(e)
    }
    ...
}

Figure 59 - SynchronizedArrayList

The commonly occurring pattern corresponding to Case 4 is the use of the Adapter pattern to provide thread-safe variants of classes that implement the Collections interface. Figure 59 shows this pattern applied to the ArrayList class.

Code that invoked the add(E e) method would be transformed into the critical section shown in Figure 60.

To recognise this situation, we extend our algorithm to apply the following check:

- If the guard is local, then the variable must belong to the same class or must have the same type as a variable of that class.

The corresponding pattern using explicit lock invocations as delimiters is shown in Figure 61.

class LockedArrayList {
    private final ArrayList arr = new ArrayList()
    private final Lock lock = new ReentrantLock();
    ...
    public boolean add(E e) {
        boolean response;
        lock.lock();
        response = arr.add(e);
        lock.unlock();
        return response;
    }
    ...
}

Figure 61 - LockedArrayList
The critical section corresponding to an invocation of `LockedArrayList.add(E e)` is shown in Figure 62.

```plaintext
guard LockedArrayList.lock
variables LockedArrayList.arr
      ArrayList.elementdata
```

Figure 62 - LockedArrayList CS

The element e does not appear either of the critical sections because it is handled as a parameter and, thus, as a value local to the method invocation. It is not accessed using GETFIELD or GETSTATIC.

Applying the same check as that applied to SynchronizedArrayList correctly ensures that the guard and the guarded variables belong to a common instance so that the approximation to equivalent static variables yields a correct result.

Given the actions recommended in this sub-section and subject to the stated assumption, our analysis shows that, though incomplete, our analysis is sound for many commonly used synchronisation patterns.

Consider the example in Figure 63 below.

```plaintext
class Triangle {
    protected Point[] points;
    static volatile int modcount = 0;
    public Triangle() {}
    public Triangle(Point[] points) throws Exception {
        // This clones the references
        this.points = points.clone();
        if (!isOK()) {
            throw new Exception("Triangle must have three points");
        }
    }
    public boolean isOK() {
        return points.length == 3;
    }
    public void modifyPoint(int index, float x, float y) {
        synchronized(points) {
            points[index].x = x;
            points[index].y = y;
        }
    }
}
```
public class RightAngled extends Triangle {
    public RightAngled(Point[] points) throws Exception{
        super(points);
        if (!isOk()) {
            throw new Exception("Not a triangle");
        }
    }
    @Override
    public boolean isOK() {
        float hsq;
        float asq;
        float osq;
        synchronized(points) {
            // must satisfy Pythagoras's relation
            // detail of this code omitted
        }
        return hsq == asq + osq;
    }
}

class Checker implements Runnable {
    boolean stop = false;
    Triangle.RightAngled rt;

class Checker(Triangle.RightAngled t) {
    rt = t;
}

class Modifier implements Runnable {
    boolean stop = false;
    Triangle tr;

class Modifier(Triangle t) {
    tr = t;
}

    public void run() {
        while (!stop) {
            try {
                Thread.sleep(1000);
                tr.modifyPoint(0, 0, -1);  
                stop = true;
            } catch(Exception e) {
                stop = true;
            }
        }
    }
}

Figure 63 - Triangle class example

In this example, a Triangle is described by three Points. A Point holds the Cartesian coordinates as primitive values. A RightAngled class extends
Triangle and imposes the Pythagorian criterion. It is intended that these classes should be thread-safe, so the public methods synchronize on the points array. However, because the points array is not initialised with a deep copy, if the Triangle and RightAngled classes are initialised with the same set of points there is a data race. The guards invoked are local to the class instances though the points themselves are shared. This is demonstrated by the Checker and Modifier classes. If this error is corrected, by initialising the points array with a deep-copy the data race is eliminated.

However, our analysis continues to report a potential data race error as it cannot appreciate that the sets of points in the two class instances are now distinct. We observe that the better coding practice would be to make each class instance of Point immutable and change the modifyPoint() method as shown in Figure 64. This technique confines write actions within the class initialisation and would, therefore, not be faulted by our analysis.

```java
public void modifyPoint(int index, float x, float y){
    synchronized(points) {
        points[index] = new Point(x, y);
    }
}
```

Figure 64 - Improved modifyPoint()

We have been unable to find counter examples that would not be trapped by our explicit additional checks, though, as shown by this example, these may produce some false alarms.

**IV.2.6.2. Effect of method summaries**

Because of the conflation of variables discussed in section IV.2.6.1, our abstraction of events does not rely on the actual values passed as parameters to method invocations. We do not classify method invocations as must or may. We treat all invocations as must occur and all invocations of the same method as being identical in their effect within our abstraction. These decisions significantly simplify the graph representing the chains of method invocations. These approximations allow us to reduce the effect of a method invocation to the method summary of the invoked method. The adopted technique for handling nested critical sections as described in section IV.2.5.3 simplifies the handling of invocations because it suffices to add the
critical sections to the summary of the main method and merge the invoked method's non-critical section with the current section of the invoking method, critical or non-critical. This has no detrimental effects beyond those already discussed in section IV.2.5.3.

**IV.2.6.3. Analysis within a method**

We rely on Alglave's proofs that the abstract event graph concept soundly and completely represents the effect of a program (Alglave 2010). We assume that the proof would apply mutatis mutandis to a program described by its Java bytecode. Based on this assumption, our analysis is generally sound. However, even "must" events within "must" critical sections may not occur. Consider the sample code fragment shown in Figure 65.

```java
if (a) { b(); } else { c(); }
...
if (a) { d(); } else { e(); }
```

**Figure 65 - Common conditionals**

If the evaluation of a yields the same result in both conditional statements, the execution sequences: b(); e(); and c(); d(); are impossible. However, the abstract event analysis includes these possibilities as a safe over-approximation. If they occur, they will cause a false positive report.

**IV.2.6.4. Summary of soundness and completeness**

In this section, we have provided an analysis of the extent to which our algorithm and its prototype implementation are capable of detecting data race errors. We believe that the analysis is generally sound and we have shown where it is un-sound and incomplete. We believe that our approximations deliver a useful measure of efficient scalability.

**IV.2.7. Other related work**

In the section we discuss other work in the same field that has had less direct influence on our thinking. There are sub-sections that deal with:

- Dynamic analysis;
- Formal verification;
**IV.2.7.1. Dynamic analysis**

This is still an active area of research (Flanagan and Freund 2010, Flanagan and Freund 2013, Wilcox, Finch et al. 2015). Recent work has incorporated an initial static analysis in an attempt to reduce the degree of instrumentation required (Rhodes, Flanagan et al. 2017). Implementations of cloud computing rely on large farms of processors. On such a platform, a parallel algorithm may be implemented across many processors using a software platform such as OpenMP (Maruyama, De Supinski et al. 2016). In that circumstance, the overheads of instrumentation are small when compared to the cost of transporting messages between processors. However, as we argued in Chapter III, this biases the design in favour of large long-running tasks. In this thesis, we have been concerned to address the problems associated with the implementation of parallelism using a multi-threaded approach. We have taken the view that this cannot accommodate the Observer effect distortion caused by instrumentation.

**IV.2.7.2. Formal verification**

There is a corpus of work that applies the techniques of formal verification to the task of proving that a program is free of data races. The techniques include memory-model sensitive approaches (Roychoudhury 2002, Yang, Gopalakrishnan et al. 2004, Burckhardt, Alur et al. 2006) and theorem proving (Ábrahám, de Boer et al. 2005). We feel that the caveats set out in section II.2.2 of Chapter II apply with equal force to these techniques, making them unsuitable for general industrial use.

**IV.3. Handling access to elements of a Collection**

Because of our "no-values" approximation, we cannot distinguish which elements of a Collection are being individually accessed. As a result, we cannot deal with the coding pattern where different threads access dis-joint ranges of indexes to a Collection. We deliver the sound but incomplete result that all such accessed are conflicting and thence potential data races.

This section describes the extension of our algorithm to handle those programs where access to elements of a Collection is achieved by the use of the Java streams feature. We introduce the concept of Java streams and show
how the Abstract Event Graph notation introduced in section II.1.6 of Chapter II may be extended to handle stream actions. We assume that the lambda functions provided to the stream actions act only on the element of the Collection with which they have been provided. We provide a proof that this extended notation succinctly and correctly encodes the effects of the stream actions as applied to each element of the Collection. We use the summarisation notation to extend our search for data races to programs that use the Java streams feature.

Because the abstract event abstraction makes it impossible to distinguish between the different elements of arrays or Collections, our static analysis can only say that all accesses to individual elements are potential data races. This is not a very useful result. The Java streams feature avoids this difficulty by encapsulating the access to individual elements within the stream-support package. In particular, this package makes the definition of stream actions agnostic to the mode of processing of the stream.

We begin this section by explaining how the concept provides a different way of ensuring that a thread has uniprocessor access to elements of arrays and Collections. Then we present our proposed extension to the notation that describes memory events. This extended notation summarises the effects of stream actions by implicitly encoding the effect of applying the stream action and its lambda expression to every element of the Collection. We present a proof that the resultant Summarised Abstract Event Graph (SAEG) correctly encodes the information from the corresponding AEG. Hence, we claim that the technique of representing stream actions as summarised events is equally valid within our algorithm for detecting data races.

IV.3.1. Background to Java streams

Section IV.2.3 considered the problem of finding data races caused by errors in implementing the implicit access protocol associated with the use of the acquire/release paradigm.

As noted by Radoi and Dig (2015), most data race detection algorithms have difficulty in handling programs that process Collections. The Java
streams feature appeared at Java 8 (Oracle Corp. 2014). It was intended to change the paradigm for processing the elements of Collections classes so that, where appropriate, the change from serial processing to parallel processing might be encapsulated within the Collections framework. Where parallel processing is specified, the framework assumes responsibility for organising the division of the processing into tasks that act on dis-joint sub-sets of the Collection. The framework assigns these tasks to threads drawn from a thread-pool and, where required, organises the merging of results from individual tasks. The framework encapsulates the process of iterating over the elements of a Collection, presenting them as a stream. There are stream methods that provide filters, mapping of the elements into elements of a different type, and aggregations. The developer can inject specific code into the operations of this framework by supplying it parametrically to stream methods as lambda functions.

We use as an example a collection whose elements are instances of a class T, which is a data-only class with a single attribute a as in Figure 66, and show a minimalist stream that increments the attribute for every element of the collection.

```java
public class StreamSimple {
    class T { int a = 0; }
    void body() {
        ArrayList<T> arr = new ArrayList<T>();
        // code to initialise the elements of arr
        arr = (ArrayList<T>)arr.stream()
            .map(p -> {p.a++; return p;})
            .collect(Collectors.toList());
    }
}
```

Figure 66 - Minimalist stream example

Figure 67 shows the corresponding bytecode representation.
The lambda function is mapped into a synthetic private method, `lambda$0`. The INVOKEDYNAMIC invocation obtains a handle to this method, which is then passed as a parameter to the `map` method of the stream. This makes it relatively easy to:

- Identify the start of a stream invocation and extract identity of the relevant collection;
- Extract the identity of the method that represents the lambda function from the operands of the INVOKEDYNAMIC instruction;
• Identify the end of the stream actions by the occurrence of a stream method that is one of the defined terminal methods that close a stream.

The default mode of operation is serial so that the stream methods and their lambda functions are applied successively to each element of the Collection in turn. However, the constraints imposed by the stream methods and the compilation of the lambda functions make it possible to change the mode of operation so that the framework partitions the Collection into dis-joint sub-sets and processes these sub-sets in concurrent threads. The framework completely encapsulates all this work so that the division of the Collection, the management of the threads and any necessary merging of results are all performed without intervention by the developer.

**IV.3.2. Our representation of a stream as an Abstract Event Graph**

In performing this transformation we assume that the Java stream implementation correctly encapsulates the management of threads and the splitting of the collection so that the stream actions are correctly applied regardless of whether the mode of operation selected is serial or parallel.

We describe how we can represent the actions of a stream by extending the notation used in an Abstract Event Graph to include actions that refer to elements of a Collection. We refer to this as a Summarised Abstract Event Graph (SAEG). For the purposes of this chapter, it is sufficient to prove that a summarised AEG soundly and completely represents the actions of the stream. We assume that the semantics of streams are proven and correctly implemented. We provide worked examples to show how this notation deals with particular ways in which streams can be used in handling the elements of Collections.

Provided that the lambda functions passed to stream actions are correctly written, there will be no data races. However, if the lambda functions refer to shared variables other than the elements of the stream then it is possible to cause data races.

We use as an example the class shown in Figure 66 above. For the purposes of discussion we denote the elements of the collection by
\[ arr \equiv \{x_0, x_1, \ldots x_n \} \] (82)

Figure 68 shows the corresponding AEG. The essential characteristic of a stream is that the same actions are applied to every element of the stream. Consequently, the AEG, as shown, consists of a set of event groups where each group is applied to a particular element.

We propose a more compact notation that reduces the size of the graph by summarising the groups as illustrated in Figure 69.

The notation \( S(arr) \) implies "every element of the collection". This notation achieves a compact representation that can handle unbounded collections without sacrificing the precision of the operational semantics.

**IV.3.3. Proving SAEG correctly encodes the AEG**

In this sub-section, we show proof that the notation for the summarisation of stream operations correctly encodes the events of the Alglave AEG of which
it is a transformation. This establishes that our approach is at least as sound and complete as the original AEG. This is sufficient to justify the use of the summarisation within the data detection algorithm that forms the subject of this chapter.

**IV.3.3.1. Preliminaries**

*Alglave's notation*

- $Rx$ read action on variable $x$
- $Wx$ write action on variable $x$

*SAEG notation*

- $RS(coll)$ read action on collection $coll$
- $WS(coll)$ write action on collection $coll$
- $RS(coll).a$ read action on the variable $a$ within every element of collection $coll$
- $WS(coll).a$ write action on the variable $a$ within every element of collection $coll$

**IV.3.3.2. Summary events**

We refer to the events within the SAEG as summary events.

**Theorem 1**

A summary event soundly and completely represents the set of events for a stream invocation.

We prove this theorem by separately proving it to be true for events that refer to elements of a collection and for events that refer to shared variables.

**Lemma 2**

Theorem 1 is true for events that refer to elements of the collection.

**Proof**

Consider a stream definition whose lambda expressions make no reference to anything other than the elements of the stream collection. Suppose that this stream definition refers to a collection $coll \equiv \{x_0, x_1, x_2, \ldots \}$. If we consider the sequential implementation of the invocation of this stream definition, then, in turn, each element of the collection will be processed by a group of operations such as map, flatmap, etc. Each operation, together with its associated lambda expression will generate a number of events so that the group generates a set of events. Each event has an action, $a$, and the variable
on which it acts, $x$. We denote these events by tuples $\langle a, x \rangle$. There will be a set of events that represent the effect of the action of the stream invocation on a particular element, $x_0$, of the collection.

\[
\langle a_0, x_0 \rangle \langle a_1, x_0 \rangle \langle a_2, x_0 \rangle \text{ etc.} \quad (83)
\]

with similar sets of events for each element of the collection

\[
\langle a_0, x_1 \rangle \langle a_1, x_1 \rangle \langle a_2, x_1 \rangle \text{ etc.} \quad (84)
\]

\[
\langle a_0, x_2 \rangle \langle a_1, x_2 \rangle \langle a_2, x_2 \rangle \text{ etc.} \quad (85)
\]

etc.

However, streams may be executed in parallel, so that the constraints imposed by the $stream$ paradigm must include the possibility that there are concurrently executing threads each of which processes a single element of the collection. The only constraint is the program order, $po$, so that, for any given element of the collection,

\[
a_0 \rightarrow_{po} a_1 \rightarrow_{po} a_2 \rightarrow_{po} \text{ etc.} \quad (86)
\]

This means that only some sequences of events are valid, but we can, legitimately, take a grouping where each action is applied to the elements of the collection before the next action is executed. This transforms the set formed by the union of the sets shown in (104), (105) and (106) into the sets represented in (108), (109) and (110).

\[
\langle a_0, x_0 \rangle \langle a_0, x_1 \rangle \langle a_0, x_2 \rangle \text{ etc.} \quad (87)
\]

\[
\langle a_1, x_0 \rangle \langle a_1, x_1 \rangle \langle a_1, x_2 \rangle \text{ etc.} \quad (88)
\]

\[
\langle a_2, x_0 \rangle \langle a_2, x_1 \rangle \langle a_2, x_2 \rangle \text{ etc.} \quad (89)
\]

etc.

This re-grouping neither adds to nor subtracts from the total set of elements representing the effect of the stream invocation and the transformation is, therefore, sound and complete with respect to the definition of a data race over the original AEG. In particular, any data race on a variable such as $x_1$, in the original AEG, as formally defined in section IV.1, will exist on the variable $x_1$ in the SAEG because a critical section imposes no ordering on its guarded events.
We can represent this re-grouping as

\[
\{a_0\} \times \text{coll} \tag{90}
\]
\[
\{a_1\} \times \text{coll} \tag{91}
\]
\[
\{a_2\} \times \text{coll} \tag{92}
\]
\[
\text{etc.} \tag{93}
\]

and note that the program order constraint is still effective.

\[
\{a_0\} \times \text{coll} \xrightarrow{po} \{a_1\} \times \text{coll} \xrightarrow{po} \{a_2\} \times \text{coll} \xrightarrow{po} \text{etc.} \tag{94}
\]

Using our notation, we represent this statement of the effect of the stream invocation as, for example,

\[
\text{RS} (\text{coll}) \xrightarrow{po} \text{WS} (\text{coll}) \xrightarrow{po} \text{RS} (\text{coll}) \tag{95}
\]

The same arguments apply mutatis mutandis for references to the fields within an element of a collection.

Lemma 3

*Theorem 1 is true for events that refer to shared variables.*

Proof

Consider a stream definition in which a lambda expression refers to a variable \(y\). If we form sets, as before, then every set will contain the same event \(\langle a, y \rangle\). Hence, the effect of all the sets is soundly and completely represented by a single summary copy of that event.

Proof of Theorem 1

Events either refer to an element of the collection or to a shared variable. There are no other cases. Because the events implement the no-values abstraction they are truly independent. If a stream includes both types of event it is legitimate to consider each type in isolation. Lemma 2 proves that theorem 1 holds for elements of a collection. Lemma 3 proves that theorem 1 holds for shared variables. This proves theorem 1.

IV.3.4. Applicability of SAEG to standard iteration

In this section we discuss how the approach used to handle stream actions might be extended to handle the style of iteration that avoids the explicit enumeration of the elements of a collection.

Suppose that \text{coll} is defined as \text{Collection}<\text{T}> and \text{T} is a class with a single \text{int} variable \(a\). The collection might be processed by code similar to that shown in Figure 70.
class T {
    int a;
}
...
Collection<T> coll = new ArrayList<T>();
...
for (T t : coll) {
    t.a++;
}

Figure 70 - Sample iteration code

The corresponding bytecode is shown in Figure 71.

ALOAD 0
GETFIELD au/com/wcc/drd/samples/TryIteration.coll : Ljava/util/Collection;
INVOKEINTERFACE java/util/Collection.iterator ()Ljava/util/Iterator;
ASTORE 2
GOTO L0
L1
FRAME FULL [au/com/wcc/drd/samples/TryIteration T java/util/Iterator] []
ALOAD 2
INVOKEINTERFACE java/util/Iterator.next ()Ljava/lang/Object;
CHECKCAST au/com/wcc/drd/samples/TryIteration$T
ASTORE 1
ALOAD 1
DUP
GETFIELD au/com/wcc/drd/samples/TryIteration$T.a : I
ICONST_1
IADD
PUTFIELD au/com/wcc/drd/samples/TryIteration$T.a : I
L0
FRAME SAME
ALOAD 2
INVOKEINTERFACE java/util/Iterator.hasNext ()Z
IFNE L1
RETURN
MAXSTACK = 3
MAXLOCALS = 3

Figure 71 - Bytecode for iteration example

Our code for transforming bytecodes into tokens already generates specific tokens for the labels L0 and L1 so that the subsequent analysis can make specific provision for the handling of loops. It would be possible to extend the set of tokens used to include tokens specific to this style of iterated loop. If this were done, then the loop body, between L0 and L1, could be recognised and treated similarly to a lambda expression supplied as the parameter to a stream method.

The references to elements of the collection would be recognised as distinct variables and excluded from the lists of references to shared variables.

IV.3.5. Summary of benefits provided by SAEG
Although we still have no solution for the problem of analysing programs that explicitly make reference to individual elements of a Collection, we have
shown that, for those who embrace the Java *streams* feature, it is possible to apply Abstract Event Graph analysis to their programs. We have provided a proof that the lambda functions passed to stream actions can be handled within an Abstract Event Graph by using our novel summarisation notation. This approach is no less sound and complete than the analysis of the AEG itself. We have further shown that a similar approach could be used to handle the iteration pattern that avoids explicit enumeration of elements of the collection. We have provided a description of the implementation that would be needed to extend our existing data race detection algorithm to handle *stream* actions and this style of iteration. This should encourage the use of the *streams* feature rather than explicitly enumerated iteration as our analysis can detect errors that might otherwise pass unnoticed.

**IV.4. Implementing our algorithm**

This section describes the detail of our data race detection implementation. The overall processing is controlled by the Invocation Hierarchy Explorer (IHE), which is described in section. Each method is processed with the algorithm described in the following sections to deliver *Annotated sections and events*, as shown in Figure 72. These are then in-lined into the invoking method as the IHE retreats back up the invocation hierarchy. The search for data races then follows the flow from *Form guards* onwards.

![Data race detection schematic](Figure 72)

137
In the following sections, we describe each of the steps in the schematic in detail.

**IV.4.1. Identifying events and their operands**

In this section we describe how we form the Abstract Event Graph from the instructions in a method. For the purposes of the abstraction used in our algorithm, a method has a set of instructions and each instruction is denoted by a tuple comprising *linenumber, class, opcode* and *operand*. The denotations for these and the definition of events were provided in section IV.1.

The Chord tool uses the SOOT framework (Vallee-Rai 2000) to transform Java bytecode into suitable internal data structures. There are three open-source libraries for the manipulation of Java bytecode:

- SOOT originally developed by Vallee-Rai (2000);
- BCEL - maintained by the Apache Foundation (Apache 2010);
- ASM - written by Eric Bruneton, (Bruneton, Lenglet et al. 2002), maintained by him and his colleagues, and distributed through ObjectWeb.

Unfortunately, neither SOOT nor BCEL has been maintained to support the Stack Frame Maps that were made mandatory in Java 8. We wished to exploit the Java 8 language features within our prototype and to analyse programs that used the Java 8 *stream* feature and the Java 9 VarHandle methods. So, we chose to use the ASM bytecode handling framework (Bruneton, Lenglet et al. 2002), which is actively maintained by its authors.

This framework provides two different but complementary models for accessing Java bytecode:

- Event-driven
- Tree-structured

The ASM manual advises that the Tree-structured model offers a slightly more flexible interface, but at the price of higher CPU resource consumption
and a larger memory footprint. As we were concerned to increase the efficiency of our process and to minimise its memory footprint, we chose to use the event-driven view.

The event-driven view is based on the Visitor pattern. The ClassReader and ClassWriter classes within the package mechanise all the details of reading and writing bytecode files. As the input file is traversed, the ClassReader invokes the appropriate methods of a user-written sub-class of the provided abstract ClassVisitor class. These methods can take any desired action including invoking the ClassWriter to generate a modified copy of the bytecode. We have used these features to generate modified bytecode for classes that, for example, includes planted invocations of fence methods.

Using the ASM framework, we transform bytecode files into Class objects. Each Class object has a collection of Method objects. Each method has a list of Instruction objects. As the ASM framework passes over a method, we classify each instruction as a type of token so that a subsequent pass over the Instruction list of a method can parse the stream for the patterns of conditional instructions that represent loops.

Javacc is an open source parser generator from javacc.org. It is written in Java to take a grammar specification and generate the Java source of a corresponding recursive descent parser. It also provides a Visitor pattern so that, for example, a control-flow graph can be constructed as the input is analysed. We have used this tool to provide a flexible way of identifying patterns of instructions and again to identify the limits of critical sections by parsing the de-limiter patterns from the bytecode of methods.

The output from this parser is a simplified list of instructions where each loop is reduced to a simple conditional where one branch represent the execution of the loop body and another represents its omission. This simplification is valid because of our adopted "no-values" abstraction. Thus, the output from this parse phase is a method-level abstract event graph.

The variable referenced by GETSTATIC and PUTSTATIC instructions may be statically derived from the operand of the instructions. For GETFIELD and PUTFIELD instructions, this operand gives only the address within an object. To resolve the address, this must be combined with the
object reference from the top-of-stack. An accurate determination of the top-of-stack value can only be obtained from the execution path leading to the instruction.

We avoid the exhaustive enumeration of execution paths by considering the summarisation of the effect of the initialisation that is executed before the threads are started in the following way. Consider the problem posed by the code shown in Figure 49 and Figure 50. By analysing the initialisation code, we could enumerate all the possible objects to which $c$ may be a reference. We then assume that both $foo$ and $bar$ may use any of these objects. All these objects must be instances of the same class, $Aclass$. If they are different instances, there is no data race. However, in the cases where they are the same instance, there is a data race possibility. Accordingly, we might soundly approximate the finding of data races by concentrating on those cases where the objects are the same. We can achieve the same effect as this approximation by treating all GETFIELD and PUTFIELD instructions as if they are GETSTATIC and PUTSTATIC instructions. This approximation is sound but incomplete. It erroneously ignores the cases where the initialisation code deliberately and carefully segregates the objects prior to starting the threads so that each thread has its own set of objects. We analysed the effects of this approximation on the soundness and completeness of our approach in section IV.2.6.

An Abstract Event Graph (AEG), as shown in Figure 45, represents the effect of a program as a set of memory events that access variables. In our algorithm for finding data races we assume that critical section de-limiters function correctly so that the program is sequentially consistent. Accordingly, the only characteristic of the AEG on which we rely in this chapter is that it soundly and completely represents the execution of the program with respect to variables.

In our algorithm we assume, generally, that critical sections and events encountered in an execution trace will be encountered in many threads in the program, though we have made a static and approximate analysis of the methods that may be invoked in a multi-threaded context. This is the only available approximation that can be applied to an open program and is a safe
over-approximation for closed programs. For closed programs, it may report false positives for those programs that carefully segregate the use of synchronisation methods into separate threads.

IV.4.2. Finding critical section de-limiters

In this section and the next we describe the process for finding critical sections and for associating events with them and for classifying those events under must and may categories. We find critical sections (and report invalid program structures) by using a recursive descent parser. Rather than designing and implementing a specific parser, we chose to build in flexibility by using an open source parser generator driven by a grammar definition. This approach was able to accommodate both the parsing of a variety of delimiter patterns and the handling of lock() and unlock() invocations.

We divide these de-limiters into three classes:

- **synchronized** construct;
- Lock interface invocations;
- Other patterns.

**Synchronized methods and blocks**

Where the synchronized construct is applied, the critical section is the whole body of the method. We handle this case with specific coding for synchronized methods. The corresponding lock is identified by the class name and method name. A synchronized block is de-limited by matching pairs of MONITORENTER and MONITOREXIT instructions. The reference on the top-of-stack provides the object instance that is the lock. In both these cases, the critical section is a block that conforms to the lexical structure. There are no problems with the critical section having its de-limiters in different blocks.

Figure 73 shows an example of the use of these patterns,
class Example01 {
    Object lock = new Object();
synchronized void aMethod() {
        // Useful work
    }
    void bMethod() {
        ...
        synchronized(lock) {
            // Useful work
            ...
        }
        ...
    }
}

Figure 73 - Synchronized methods and blocks

**Lock interface invocations**

The Lock interface invocations are not constrained to conform to the lexical block structure. However, a failure to do so makes it difficult to prove that there are no orphan de-limiters. Because implementations of the Lock interface encapsulate the mechanisms used, we chose to recognise invocations of its methods explicitly as de-limiters. This caters for the most common cases where critical section de-limiters are isolated within methods. We deliberately chose to otherwise prohibit critical sections that span inter-method boundaries to enable the summarisation of critical sections and their events on a per-method basis. This is crucial to a reduction of the path explosion that would otherwise occur.

Figure 74 shows a correct use of the use of explicit lock invocations as de-limiters. Essentially, the synchronized block syntax is replaced by the use of lock.lock() and lock.unlock() invocations.
class Example02 {
    Lock lock = new ReentrantLock()
    void aMethod(){
        ...
        lock.lock()
        //Useful work
        ...
        lock.unlock()
        ...
    }
}

Figure 74 - Explicit locks

A similar effect can be achieved by the use of AtomicInteger instances.
final int FREE = 0;
final int TAKEN = 1;
AtomicInteger flag = new AtomicInteger(FREE);
void aMethod(){
    ...
    while flag.compareAndSet(FREE, TAKEN))
        {Thread.yield();}
    //Useful work
    ...
    flag.compareAndSet(TAKEN, FREE));
    ...
}

Figure 75 - AtomicInteger de-limiters

Other patterns
In-lining the code associated with de-limiter patterns rather than encapsulating them within a class may achieve a small, but measurable, gain in performance. It may be argued that this in-lining process should be left to an optimising Just-in-Time compiler, however, many practitioners would opt for annotations as a way to provide the necessary syntactic sugar while ensuring the absence of method invocation overheads. Accordingly, we have accepted the need to parse an instruction stream for the other patterns that represent known implementations of de-limiters.

Figure 76 shows the use of VarHandle features as de-limiters. The use of a separate DSHelper class forces the static resolution of variable handles.
class Example04 {
    final int FREE = 0;
    final int TAKEN = 1;
    int flag = FREE;
    void aMethod(){
        
        while (DSHelper.flag.compareAndSet(this, FREE, TAKEN)){
            Thread.yield();
            
            //Useful work
            
            
        }
        
        DSHelper.flag.compareAndSet(this, TAKEN, FREE);
        
    } 
}

Figure 76 - Explicit use of VarHandle features

As noted previously in section IV.2.1.2, the code shown in Figure 74, Figure 75 and Figure 76 is logically correct, but requires further embellishment to implement the proper handling of exceptions.

Sample grammar

Figure 77 shows a sample grammar that covers synchronised blocks and critical sections de-limited by AtomicInteger invocations.

The de-limiter patterns are recognised as the bytecode is transformed into tokens that are then analysed by the parser.

\[
\begin{align*}
section & \equiv (subsection) + \\
subsection & \equiv \text{atomic} | \text{sync} | \text{instrn} \\
sync & \equiv \text{synccstart section synccend} \\
synccstart & \equiv \langle \text{START} \rangle \\
synccend & \equiv \langle \text{SEND} \rangle \\
\text{atomic} & \equiv \text{atomicstart section atomicend} \\
\text{atomicstart} & \equiv \langle \text{GET} \rangle (\text{INSTRN}) (\text{INSTRN}) (\text{ASTART}) \\
\text{atomicend} & \equiv \langle \text{AEND} \rangle \\
\text{instrn} & \equiv (\text{INSTRN}) (\text{GET})
\end{align*}
\]

Figure 77 - Sample grammar

Other patterns would be accommodated by extending the definition of subsection and by adding additional definitions similar to atomic, atomicstart and atomicend. We developed grammars to recognise:

- message-passing with two variables, similar to Dekker's algorithm;
- message-passing with a single variable, similar to Peterson's algorithm.
In all cases, the implementations were modified so that the various flag variables were declared to be volatile. This use of the volatile construct forces the JVM to treat accesses to the variable as if they were atomic and to propagate changes to the variable to all threads. Otherwise, compiler optimisations and weak memory effects may prevent read actions from observing write actions in other threads that change the value of the shared variable. Here we show examples of Java programs that simply use volatile variables as flags.

**Message-passing with two variables**

```java
public class MP2 {
    private static volatile int u = 0;
    private static volatile int v = 0;
    public static volatile boolean stop = false;

    public send() {
        int ucurrent = 0;
        while (!stop) {
            while (u == ucurrent) {Thread.yield();}
            v = ucurrent;
            ucurrent++;
        }
    }

    public receive() {
        int vcurrent = 0;
        while (!stop) {
            while (v == vcurrent) {Thread.yield();}
            u = vcurrent++;
        }
    }
}
```

**Figure 78 - Message-passing two variables**

Lines 9 and 19 in Figure 78 are recognised as acquire de-limiters. Lines 11 and 21 are recognised as release de-limiters.

**Message-passing with one variable**

```java
public class MP1 {
    private static volatile int u = 0;
    public static volatile boolean stop = false;
```
In Figure 79, lines 8 and 16 are recognised as acquire de-limiters. Lines 10 and 18 are recognised as release de-limiters.

IV.4.3. Associating events with critical sections

In section IV.1 we gave a general definition of data races where the accessing events are not synchronised. Here we consider the cases where the synchronisation is faulty. We detect such data races by finding variables that are accessed under different guards. First we find the critical sections and associate with each the guard and the set of variables that the critical section guards.

We expand the control-flow graph of a program into a set of execution traces, which we examine for critical sections.

Let \( G \) denote the set of guards used in a program, where a guard, \( g \), protects a set of variables. Each guard is a tuple that includes the type of guard and the identity of the variables or objects that implement the lock. For the purposes of this analysis, this level of detail is not relevant. All that matters is that the guards are distinct.

\[
g \in G \tag{96}
\]

A critical section, \( cs \), is denoted by a tuple comprising a guard, \( g_{cs} \), and a set of events, \( E_{cs} \), that are guarded.
\[ \text{cs} \equiv \langle \text{g}_{\text{cs}}, \text{E}_{\text{cs}} \rangle \]  
(97)

with the projection functions
\[ \text{g}_{\text{cs}}(\langle \text{g}_{\text{cs}}, \text{E}_{\text{cs}} \rangle) \equiv \text{g}_{\text{cs}} \]  
(98)
\[ \text{E}_{\text{cs}}(\langle \text{g}_{\text{cs}}, \text{E}_{\text{cs}} \rangle) \equiv \text{E}_{\text{cs}} \]  
(99)

Let \( \text{CS} \) denote the set of critical sections so that
\[ \text{cs} \in \text{CS} \]  
(100)

Let \( \text{ECS} \) denote the set of sets of events guarded within critical sections so that
\[ \text{ECS} \equiv \{ \text{E}_{\text{cs}} \mid \exists \text{cs} \in \text{CS} \land \text{E}_{\text{cs}} = \text{E}_{\text{cs}}(\text{cs}) \} \]  
(101)

We treat events that do not occur within a designated critical section by placing them in a critical section with a unique, conventionally identified guard. Let \( ncs \) denote this section so that
\[ ncs \equiv \langle \text{g}_{\text{ncs}}, \text{ecs} \rangle \in \text{CS} \mid \exists \text{g}_{\text{ncs}} \in \text{G} \land \text{g}_{\text{ncs}} = \text{NONCRITICAL} \]  
(102)

In this sub-section we have formally defined critical sections and provided a denotation for the events that occur within them. Next we show how we group together critical sections that share the same guard and thence, how we derive the sets of events that are guarded by the same guard.

**IV.4.4. Grouping critical sections, guards and guarded variables**

Let \( \text{CS}_g \) denote the set of critical sections that share the same guard, \( g \).
\[ \text{CS}_g \equiv \{ \text{cs} \in \text{CS} \mid \exists g \in \text{G} \land g = \text{g}_{\text{cs}}(\text{cs}) \} \]  
(103)

Let \( \text{E}_g \) denote the set of events that share the same guard, \( g \), which we derive from the critical sections by
\[ \text{E}_g \equiv \{ e \in \text{E}_{\text{cs}} \mid \exists \text{cs} \in \text{CS}_g \land \text{E}_{\text{cs}} = \text{E}_{\text{cs}}(\text{cs}) \} \]  
(104)

We derive the set of variables that share the same guard, \( \text{V}_g \), from these events.
\[ \text{V}_g \equiv \{ v \in \text{V} \mid \exists \text{id} \in \text{ID} \land \exists \text{a} \in \text{A} \land (\text{id}, \text{a}, v) \in \text{E}_g \} \]  
(105)

In this sub-section we showed how we derive the sets of variables that are guarded by specific distinct guards. Next we show how we use these to find potential data races and how we impose the full criteria for the existence of data races.
IV.4.5. Finding data races across critical sections

This section describes the remaining steps in the process. A potential data race exists if a variable is a member of more than one set of guarded variables. Hence, we find the set of variables that are involved in data races, DR, by taking the intersections of the sets of variables from pairs of guards, members of $G$.

Let $GPAIR$ denote the set of pairs of non-equal guards and GPAIR the set of all such pairs

$$GPAIR \equiv \{ (g_0, g_1) \mid g_0, g_1 \in G \land g_0 \neq g_1 \} \quad (106)$$

Let $V_{g_0}, V_{g_1}$ denote the sets of variables guarded by two different guards, $g_0, g_1$, so that

$$V_{g_0} \equiv \{ v \in V \mid g = g_0 \} \quad (107)$$

$$V_{g_1} \equiv \{ v \in V \mid g = g_1 \} \quad (108)$$

The set of potential data races, PDR, is given by the union of the pair-wise intersections of guarded variables with different guards.

$$PDR \equiv \bigcup_{(g_0, g_1) \in GPAIR} \{ V_{g_0} \cap V_{g_1} \} \quad (109)$$

Let $E_{pdr}$ denote the set of events that access a potential data race, $pdr$, and $EPDR$ denote the set of such sets

$$E_{pdr} \equiv \{ e \in E \mid \exists pdr \in PDR \land pdr = v_e(e) \} \quad (110)$$

$$EPDR \equiv \bigcup_{pdr \in EPDR} E_{pdr} \quad (111)$$

Because we assume that all critical sections exist in all concurrent threads, we need only check that the events have at least one each of read and write events. Accordingly, we re-define the expressions presented in section IV.1 more simply. Let $E_r$ denote the set of events whose action is read and $E_w$, the set of events whose action is write

$$E_r \equiv \{ (id, a, v) \in E \mid a = read \} \quad (112)$$
\[ E_w \equiv \{(id, a, v) \in E \mid a = \text{write}\} \quad (113) \]

so that the set of events for actual data races is given by the potential data race events filtered by the criterion.

\[ E_{dr} \equiv \left\{ e \in E_{pdr} \mid \exists e_{pdr} \in EPDR \land \exists e_r \in E_{r} \land \exists \exists e_w \in E_{w} \right\} \quad (114) \]

Now we derive the set of actual data race variables from the set of events that cause data races and thence we derive the list of instructions that access those variables. The corresponding list of data variables is given by

\[ DR \equiv \{v \in V \mid \exists id \in ID \land \exists \exists a \in A \land \exists (id, a, v) \in E_{dr}\} \quad (115) \]

We can derive the set of instructions that cause data races, \( I_{dr} \), from the events

\[ I_{dr} \equiv \{ i \in I \mid \exists e \in E_{dr} \land id_e(e) = id_i(i)\} \quad (116) \]

**IV.4.6. Summary of data race detection algorithm**

In this section we formally defined the processing steps by which we extract the instructions that cause data races from their corresponding list of events and the way in which the events are grouped within critical sections. We associate events with critical sections. Based on the guards used by each critical section we form the sets of events guarded by distinct guards. From this we derive the sets of variables that are guarded by particular guards.

Potential data races exist where the same variable is accessed under different guards. As discussed in section IV.2.5.3, we allow nested critical sections, but assign events to the nested section. This may cause some false positives where variables guarded by an outer section are accessed within a nested section. Actual data races require a number of read and write events, so we find the events that access potential data race variables and impose the criterion to deliver actual data race events. We derive the variables corresponding to these events and thence the set of instructions that access these variables.

**IV.5. Reusable Framework**

This section provides some explanatory detail on the way in which the algorithm described in this chapter has been implemented as Java classes. These algorithmic classes are integrated within a package of Java classes that
form a supporting infrastructure. In particular we identify the infrastructure code that we re-use in the work described in Chapter V.

The Invocation Hierarchy Explorer (IHE) provides the essential execution framework that relies on the Class and Method cache to insulate it from the details of handling bytecode files. The Data Race Detection and Abstract Event Graph analysis classes are invoked from within the IHE.

In this section we deal only with the use of this infrastructure to support the finding of data races. In Chapter V, we describe its re-use to support the selection and placement of fences through an Abstract Event Graph (AEG) analysis.

**IV.5.1. Method invocation hierarchy**

The Invocation Hierarchy Explorer encapsulates the ability to investigate a hierarchy of method invocations. It relies on the Class and Method Cache to provide it with information about classes and methods while encapsulating the details of the interface with the ASM bytecode manipulation library. The explorer has a generic mechanism for executing specific code as the hierarchy is traversed. The critical section analysis and AEG analysis classes are invoked through this mechanism. The process is shown in Figure 80.

![Diagram](source)

**Figure 80 - Implementation schematic**

Our implementation assumes that the user can identify an entry point method for the program. For closed programs this is simple, but open programs require the synthetic construction of a suitable harness class.

The system examines the method and prepares a list of the classes named in its method invocations. It builds those classes and then recursively repeats the process for each of the invocations. The system creates a separate processing thread for each invocation and then waits for these threads to complete. The recursion ceases when the invoked method has no
invocations of its own or when the invocation refers to a method that has already been encountered so that we specifically trap re-entrant invocation graphs. This mechanism is encapsulated within a specific class that provides a generic mechanism for handling the corresponding invocation graph. The variations associated with the different uses of the class, such as the critical section analysis and the AEG analysis, are accommodated by standard inheritance techniques.

IV.5.2. Class and method cache

The system maintains lists of classes and their methods for the program that is being analysed. As the usage of new classes is discovered they are added to the lists. When the system needs to obtain the information about a method, it retrieves it from the cache. If it can be found, then the method and all the other methods of the same class have already been built. If not, the retrieval triggers the building of classes and their methods.

When this process is complete, the method will be present in the method cache. This part of the design is illustrated in Figure 81.

The building of classes uses a multi-thread technique. If a class is not yet built, a thread is dispatched to build the class and all its methods. More threads are dispatched until all the classes in the list have been built or are being built. The system then waits until the building of these classes is complete. This part of the design is shown in Figure 82.
We conducted a simple test to determine the benefit achieved by this multi-threaded design. Figure 83 shows the times in milliseconds taken to process a set of eight classes one million times. In the serial test we processed each of the eight classes individually while in the parallel test we set up all the classes and processed them together. The results shown in Figure 83 were obtained on Platform 1 as described in section III.1.4.3 of Chapter III.

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Parallel</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1950</td>
<td>1918</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1887</td>
<td>348</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1891</td>
<td>364</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1921</td>
<td>364</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1930</td>
<td>350</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1951</td>
<td>337</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1899</td>
<td>360</td>
<td>5</td>
</tr>
</tbody>
</table>

Building a class involves the reading of the bytecode contained in the Java class file and its transformation into our ClassExtract and MethodInfo classes. The MethodInfo class holds a list of Tokens. A Token encapsulates an instruction. The different sub-classes of Token are used:

- to identify and fix up references and labels;
- to identify memory events;
- to identify invocations.

The Java class files are read using the ASM bytecode manipulation library (Bruneton, Lenglet et al. 2002). The ClassExtract and MethodInfo classes are built by the Visitor-pattern methods of an Adapter class following the
example in the ASM documentation. This part of the design is shown in Figure 84.

![Figure 84 - Build class from bytecode]

We have chosen to build all the methods for a class at once to avoid the cost of repeatedly opening the class file and initialising the ASM code.

**IV.5.3. Finding critical sections**

This processing is specific to the finding of data races.

When the system reaches a "leaf" method, that is a method that requires no further recursion, it processes the method according to the following sequence of steps to generate a method summary:

- The Tokens for the method are processed sequentially;
- A grammar-driven recursive descent parser is used to identify the critical sections from their de-limiter patterns;
- The non-critical section and the critical sections are built, together with their memory events.

The same process is applied at every level as the recursion stack unwinds with the addition that the effect of the invoked methods is merged with the current method. This in-lines the effect of the invoked methods.

Our grammar specifically supports method invocations from the Lock interface as de-limiter patterns. This caters for much common usage, while allowing the constraint that the start and end of a critical section must occur within the same method. As explained in section IV.2.5.3 above, we do not support over-lapped critical sections and we adopt a pragmatically simple approach to the handling of nested critical sections.
IV.5.4. Extracting data races from a method summary

The exploration of the method invocation hierarchy yields a single method summary for the top-most method containing all the critical sections and their associated memory events. The implementation of the algorithms described in sections IV.2.3.1 and IV.4.5 is achieved by a conventional use of the bulk-processing methods of the various classes in the Collections package to filter, join and intersect the lists of memory events held by the critical section classes. We assume that, where appropriate, these methods will use multi-threading internally to reduce the elapsed times of these processes. We anticipate a continuing series of improvements in efficiency, as the Collections package is refined in successive future releases of Java.

IV.6. Limitations of prototype

We chose to implement a prototype that included support for sufficient of the features of the Java language to establish the correctness and scalability of our data race detection algorithm. In particular, we omitted full support for:

- Catch/try blocks;
- Invocation of Interfaces and Abstract classes.

**Catch/try blocks**

Catch/try blocks can be statically resolved into a conventional control-flow graph that uses only the lower-level conditional primitives. This is a known solved problem. Accordingly, we contend that the omission of support for this feature has no effect on the correctness of our algorithm.

**Invocation of interfaces and abstract classes**

The use of the Abstract Event Graph abstraction together with the need to avoid the full analysis of execution traces prevents the identification of the actual class that will be executed as the result of the invocation of an interface or abstract class. Within the abstraction, such invocations must be handled by in-lining the method summaries of all corresponding methods belonging to sub-classes. The identity of the sub-classes will be known at the time the invocation is processed because of the construction of the
Invocation Hierarchy Explorer. Within the prototype, we provided a further pragmatic way to handle such invocations. On the assumption that packages of sub-classes would be analysed independently and given "trusted" status, we provided a mechanism that allowed for nominated classes to be excluded from our analysis. Invocations of the methods of such classes are treated as if they had empty method summaries.

In the next section, we report the results of our evaluation of our prototype. This includes its ability to correctly detected data race errors, a systematic investigation of its scalability and a comparison with the performance of JavaRaceFinder (JRF) (Kim, Yavuz-Kahveci et al. 2012).

Although it uses very different technology, we initially chose to compare our prototype with JRF because, if allowed to run to completion, it represents an examination of a program that is very precise and, in theory, sound. Having done extensive scalability investigations of our code we were keen to examine the extent to which JRF was affected by scalability issues. Unfortunately, as we explain in the next section, JRF has not been maintained, but we were able to investigate the scalability of JavaPathFinder (JPF) (Visser and Mehlitz 2005) on which JRF relies for its fundamental analysis. Subsequently, we obtained copies of publically available code that included source code that would compile under Java 8 and under a legacy version of Java. This allowed us to measure the performance of our prototype in analysing a representative set of programs and compare our performance with that of the latest version of Chord when run in classic mode against programs compiled from the same source.

IV.7. Observations

We first investigated both the functional correctness of our prototype and its scalability by exercising it with a variety of specifically written synthetic programs. Regarding the prototype’s ability to correctly report data races, we first built a set of unit tests for the major functional classes to demonstrate that each of these classes correctly implements its specification. We then assembled these classes into sub-systems and thence into a full prototype. Years of industrial practice have demonstrated that this approach
delivers more reliable systems than exercising only a fully integrated system with a set of "real" programs. The analysis by Yin (2013), which we introduced in Chapter II, covered seventeen programs. From these he extracted only five distinct patterns of informal synchronisation. Of these only two were used by more than one program. This confirms the work of Curnow and Wichmann (1976) who successfully coded the functional impact of a whole shift of work on a KDF9 computer into a small Algol 60 program. This program and its FORTRAN equivalent achieved fame as the Whetstone benchmark. The use of synthetic test data and test programs based on a "white-box" attitude to testing is cheaper and more effective than a suite of randomly selected real programs. The latter wastes time and resources in repetitively exercising the small subset of features that are commonly used while, usually, leaving the more obscure features un-tested.

We approached the assessment of the scalability of our prototype in a similar analytical and systematic manner. We conducted a complexity analysis of the design to predict how the performance would vary as each of a number of critical external factors was varied independently. We then constructed a series of synthetic programs in which the various factors, such as number of lines of code (SLOC), were varied while all other factors were kept constant. A test suite of "real" programs will provide a range of measured figures for performance. However, without a detailed analysis of the code of each of these programs, it cannot provide the reasons why the performance is different. Again, industrial experience shows that it is more effective to conduct a systematic programme of experimentation rather than trying to analyse a random set of results. We note that in reporting the performance of JavaRaceFinder, (Kim, Yavuz-Kahveci et al. 2012), the authors can only speculate on the reasons for some of the observed results.

Because JavaRaceFinder has not been maintained we were unable to subject it to a systematic analysis. However, we were able to undertake a measurement programme on JavaPathFinder and compare the way in which its performance varies with the way in which the performance of our prototype varies as different aspects of the program being processed are
systematically varied. The following subsections give further detail on these aspects of our work.

**IV.7.1. Measurements of the performance of our prototype**

We established that the prototype implementation correctly implements its design by a sequence of progressive testing. First, we relied on conventional unit testing of the component classes using the JUnit package (Gamma and Beck 1999). Our implementation comprises nineteen major classes (excluding sub-classes) for which we wrote sixty-four unit tests. Then, we tested the prototype with thirty-two synthetic programs that exhaustively exercised the possible interactions between critical sections, events and the control-flow graph. Then, we validated the Invocation Hierarchy Explorer with a set of tests based on synthetic programs with lengthy invocation chains. When the basic functional correctness of the prototype was established, we evaluated the extent to which our prototype correctly finds data races. For this test we used twenty small programs. The first ten programs are implementations of examples drawn from a variety of sources including Dekker’s algorithm (Dijkstra 1971), Peterson’s algorithm (Peterson 1981), patterns identified by Yin (2013), tutorial examples from a number of on-line courses and the test cases in Manson, Pugh et al. (2005). All these were modified as necessary to use volatile synchronization. The remaining ten programs are the same programs, but with the offending data race event removed. In all cases, we planted a data race event outside of critical sections. In some cases the event must occur while in others it may occur.

Figure 85 shows the number and type of each potential data race detected for each of the suite of test programs, and the expected number of data races. Note that the number of reported data races may exceed the number expected where the same data race is reported under multiple classifications. For example, for program 2, there are three reported races, but two refer to the same statement.
The data races are classified with a three-character code. The value T means \textit{must}, the value F means \textit{may}. The first character refers to the planted data race event. A T value means that the planted event \textit{must} occur. The second character refers to the \textit{must/may} property of the critical sections within which the matching data race event occurs. The final character refers to the \textit{must/may} property of that matching event. For example, TFT implies that the non-critical section event must occur and forms a data race with an access to the same variable that must occur within a critical section that, itself, may or may not occur.

The results show that, for our limited set of test programs, the prototype correctly finds all the embedded data races. Despite some reports being duplicate, none of the reported data races are false positives.

The typical data race report shown in Figure 86, identifies the source code line number, where available, for instructions that cause data races.
During our experiments, we used programs in which we planted a small number of data race errors. This is consistent with the expected situation in industrial circumstances. Accordingly we have made no direct attempt to quantify the effect of trying to process a program with a large number of data race errors in a large volume of code.

To investigate scalability, we built three sets of synthetic programs, all based on a simplified version of Peterson’s algorithm (Peterson 1981). With the first set of programs we varied the number of lines of code (LOC) from 50-250 in increments of 50 lines, by adding additional statements without changing the number of critical sections or the number of conditional branches. In the second set of programs we kept the number of LOC constant at 250, but varied the number of critical sections from 2-10 in increments of 2. In the third set of programs, we kept the number of LOC at 250 and the number of critical sections at 10, but varied the number of IF statements from 2-12 in increments of 2.

We analysed each program ten times and measured the elapsed time for each run. The test runs were conducted on Platform 1, a MacBook Air with an Intel Core i5 processor running at 1.7GHz. The operating system was MacOS 10.12.2 and the Java version 1.8_05. The results for the scalability

<table>
<thead>
<tr>
<th>Data races</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTT 1</td>
</tr>
<tr>
<td>Variable au/com/wcc/drd/samples/Sample23.a 28 PUTSTATIC au/com/wcc/drd/samples/Sample23.a</td>
</tr>
<tr>
<td>TTF 1</td>
</tr>
<tr>
<td>Variable au/com/wcc/drd/samples/Sample23.a 28 PUTSTATIC au/com/wcc/drd/samples/Sample23.a</td>
</tr>
<tr>
<td>TFT None</td>
</tr>
<tr>
<td>TFF None</td>
</tr>
<tr>
<td>FTT 1</td>
</tr>
<tr>
<td>Variable au/com/wcc/drd/samples/Sample23.a 30 PUTSTATIC au/com/wcc/drd/samples/Sample23.a</td>
</tr>
<tr>
<td>FTF 2</td>
</tr>
<tr>
<td>Variable au/com/wcc/drd/samples/Sample23.b 30 GETSTATIC au/com/wcc/drd/samples/Sample23.b</td>
</tr>
<tr>
<td>Variable au/com/wcc/drd/samples/Sample23.a 30 PUTSTATIC au/com/wcc/drd/samples/Sample23.a</td>
</tr>
<tr>
<td>FFT None</td>
</tr>
<tr>
<td>FFF None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TTT</th>
<th>TTF</th>
<th>TFT</th>
<th>TFF</th>
<th>FTT</th>
<th>FFT</th>
<th>FFT</th>
<th>FFF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 86 - Sample data race report
assessment are plotted in Figure 87, Figure 88 and Figure 89. These have been plotted in graphs that show each of the measurements as a separate point. The trend lines were produced using a least squares algorithm.

Figure 87 shows the results obtained as the number of LOC in the program is increased. They show significant scatter as no extraordinary efforts were made to inhibit the various background daemons. We discussed the factors that contribute to this spread of measured values in section III.1.4.3 of Chapter III. Accordingly, we estimate the linear rate of increase in elapsed time at about 40 milliseconds per 100 LOC.

Figure 88 shows the result of increasing the number of critical sections. The graph shows a very small increase in processing time of about 23 milliseconds per 100 critical sections. Given the amount of scatter, we suggest that, for practical purposes, the processing time may be considered as unaffected by the number of critical sections in the program.
In our initial implementation, we exhaustively evaluated all the possible execution traces within a method, eliminating only those that had no effect within our chosen abstraction. We then generated instrumented machine code for each trace and executed that code to trap the object references needed to fully resolve the variables accessed by events. When we measured the effect of varying the number of branch statements in a method,

\[ y = 0.2282x + 139.07 \]

Figure 88 - Processing Time (ms) v Number of critical sections

We obtained the results shown in Figure 89. For this experiment, we put all the branches into a single method whose size remained constant, and then increased the number of branches. As expected from a simple complexity analysis, the processing time rises in an exponential manner as the number of branches increase. These results indicate that attempting to resolve addresses in that way is too costly and does not scale well.

We subsequently re-implemented the handling of conditional statements so that the list of instructions in a method is processed in a single pass by treating all accessed variables as if they had been declared static.
This removed the need for an instrumented execution of each trace. It also made the prototype insensitive to the number of branches in a method, so that the performance is fully reflected by the results shown in Figure 87. This provides good scalability but makes it more likely that *must* and *may* events will be incorrectly classified and, as discussed in section IV.2.6.1, increases the number of patterns that will generate false positive reports.

**IV.7.2. Comparison with JavaRaceFinder**

We have taken the performance results for JavaRaceFinder (JRF) as published in (Kim, Yavuz-Kahveci et al. 2012) and presented them on a single graph together with the performance of our prototype. We show this composite graph as Figure 90. The graph plots the processing time in milliseconds on a logarithmic scale against the number of traces examined also presented on a logarithmic scale. The various data series plotted refer to the JRF results for different sample programs, as indicated by the legend. Our prototype always examines all the traces in a method so we have made a fair comparison by plotting its performance for program variants with increasingly complicated control-flow graphs.

The program used to exercise our prototype was a slightly adapted variant of Peterson's algorithm (Peterson 1981). Accordingly, we have added linear trendlines to the results for our prototype and to those achieved by JavaRaceFinder for an example of that algorithm. From the graph it appears that our prototype offers an improvement in the processing speed of about one order of magnitude. We note that the results show that, with larger and more complicated programs, JavaRaceFinder shows very large increases in processing time. We also note that JavaRaceFinder may find a data race relatively rapidly, if one exists. However, proving that a program is free from data races may well involve excessively long processing times.
The only currently available version of JavaRaceFinder is that used in the experiments reported in (Kim, Yavuz-Kahveci et al. 2012). The program depends on support from JavaPathFinder (Visser and Mehlitz 2005). Unfortunately, JavaRaceFinder has not been maintained so that it is now incompatible with all the versions of JavaPathFinder that have been subsequently released. There is no archive of the version of JavaPathFinder on which JavaRaceFinder depends.

We have successfully installed the current version of JavaPathFinder. We have used this version to analyse the synthetic programs that we used to investigate the scalability of our prototype. Because of its radically different analysis technique, the performance envelope exhibited by JavaPathFinder is quite different from that shown by our prototype. For example, the incremental cost of processing a LOC is much higher, so that, as reported for JavaRaceFinder in Figure 90, the overall cost of processing any particular program is much higher.
Our prototype is absolutely insensitive to the number of threads actually used in the execution of a program. It assumes that all critical sections appear in many threads. Conversely, JavaPathFinder is sensitive to the number of threads specified to be used by the program. We investigated this by using the DiningPhil program supplied as part of the standard JavaPathFinder distribution, which allows for the parametric specification of the number of threads. Because JavaPathFinder actually executes the threads, the elapsed analysis time rises significantly with the number of threads. We show this result in Figure 91.

We were particularly interested in comparing the variation in performance with variations in the number of critical sections and with the number of branches in the control-flow graph. As shown in Figure 88, our prototype shows only a small increase in processing time as the number of critical sections is increased. Conversely, with JavaPathFinder, the elapsed analysis time rises with the number of critical sections.

Figure 92 plots the elapsed times for both JavaPathFinder and our prototype as the number of critical sections is steadily increased. The
negative extension of the vertical axis is provided to lift the results for our prototype clear of the horizontal axis.

When we investigated the behaviour of JavaPathFinder as the number of control-flow branches is increased, we discovered that the performance is substantially insensitive to the number of branches, provided that these branches do not contain critical sections.

JavaPathFinder handles loops in a way that inhibits attempts to make a meaningful comparison. In our prototype, we reduce all loops to a single execution of the loop body, and the empty loop case. In JavaPathFinder, if the loop is specified as being executed $n$ times, then the analysis appears to execute the loop $n$ times. In these cases, the analysis time rises with the number of times the loop is executed. We found that the analysis of programs with loops dependent on flags set in other threads, as shown in Figure 93, never terminated.

```java
while (!stop) {
    // keep processing work
}
```

*Figure 93 - Typical worker thread pattern*

To obtain a meaningful comparison, we were forced to modify our test programs to eliminate such loops. This stratagem tends to underestimate the potential for these effects to further increase the elapsed times for JavaPathFinder in analysing real programs.

**IV.7.3. Comparison with Chord**

To compare the performance of our prototype with that of another program that uses static analysis techniques, we chose to evaluate our prototype against the Chord tool (Naik, Aiken et al. 2006)

The current version of Chord cannot handle programs compiled with Java 8 or Java 9, so we obtained a legacy copy of Java 6, the current source code of Chord and the source code of a number of suitable subject programs. We compiled Chord and these subject programs under Java 6 and then analysed them using the compiled Chord. Our prototype relies on certain Java 8 features so we could not execute it under Java 6. For comparison, we took the subject programs, re-compiled them under Java 8 and then analysed
them with our prototype. All this was done on Platform 2 as described in Chapter III, section III.1.4.3, Equipment.

We obtained the results shown in Figure 94.

<table>
<thead>
<tr>
<th>Program</th>
<th>SLOC</th>
<th>Chord time (seconds)</th>
<th>Prototype time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashTable</td>
<td>6056</td>
<td>41.0</td>
<td>0.9</td>
</tr>
<tr>
<td>Vector</td>
<td>5098</td>
<td>11.8</td>
<td>0.6</td>
</tr>
<tr>
<td>SyncArrayList</td>
<td>5339</td>
<td>10.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Bridge</td>
<td>11075</td>
<td>63.3</td>
<td>1.4</td>
</tr>
<tr>
<td>SpaceInvaders</td>
<td>12047</td>
<td>64.4</td>
<td>1.8</td>
</tr>
<tr>
<td>sunflow</td>
<td>18289</td>
<td>188.0</td>
<td>7.2</td>
</tr>
<tr>
<td>chord</td>
<td>17992</td>
<td>369.6</td>
<td>18.2</td>
</tr>
<tr>
<td>Textifier</td>
<td>4904</td>
<td>47.7</td>
<td>3.1</td>
</tr>
<tr>
<td>ASMifier</td>
<td>5230</td>
<td>52.9</td>
<td>6.2</td>
</tr>
<tr>
<td>Garden</td>
<td>7309</td>
<td>65.3</td>
<td>0.4</td>
</tr>
<tr>
<td>CarPark</td>
<td>33</td>
<td>10.2</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Figure 94 - Chord v Prototype performance

Our prototype counts the SLOC by determining for each class processed the number of distinct line numbers that appear in the instructions of the class. This ensures that the only line numbers counted are those referred to by instructions. We note that these numbers are significantly lower than those quoted in (Naik, Aiken et al. 2006). We attribute this discrepancy to the deliberate exclusion of "native" classes.

The CarPark, Garden, Bridge and SpaceInvaders programs were taken from (Magee and Kramer 1999). We analysed the HashTable, Vector and SyncArrayList classes by writing suitable harness classes to wrap the corresponding classes in the Java Development Kit. The sunflow program, was obtained from http://sunflow.sourceforge.net/. The chord program is, of course, that which we compiled from the source code of the latest version of Chord. ASMifier and Textifier are utilities from the ASM package (Bruneton, Lenglet et al. 2002).

As expected, our prototype runs significantly faster reflecting its aggressive approximation choices. We illustrate this graphically in Figure 95 by plotting both sets of results on the same graph. We plotted elapsed analysis time in seconds against number of standardised lines of code.
In Figure 96, we show the same results presented on a chart with a logarithmic scale. This shows more clearly the differentiation between the two analyses at the low end of the scale. We note that Chord appears to have an irreducible minimum overhead of 10 secs, while for our corresponding analyses only one has an elapsed time that exceeds 10 secs.

Chord reported all these test cases as being free from data races. Our prototype reported that all these cases had no data races caused by errors in the use of the *acquire/release* paradigm. We found no cases where the
analysis arbitrarily reported any of the circumstances described in section IV.2.6.1 under Resolution of Cases Case 2, Case 3 or Case 4.

We wrote an ArrayList harness that erroneously invoked the methods of the ArrayList class under two different guards. As expected, both our prototype and Chord correctly identified this as a data race circumstance.

As discussed in section IV.2.3.2, Chord’s technique of beginning by finding all possible competing pairs leads to a high complexity throughout the subsequent processing. Conversely, our algorithm reduces the number of objects being manipulated through a series of passes over the instructions. In our prototype, higher complexity operations are not invoked until the number of objects has been greatly reduced. Naturally, our algorithm is less precise, but, as we showed in section IV.2.6, this does not appear to affect its usefulness in handling otherwise correct and conventional programs that contain errors that cause data races. This appears to be corroborated by our experimental results.

IV.7.4. Summary of results
The published results for JRF-E as displayed graphically in Figure 90 show that our prototype is significantly more efficient. Our systematic investigation of the various differentials with respect to a variety of independent variables shows that in all cases our prototype shows significantly better scalability. We note that JRF-E also uses significantly more memory.

Subject to its limitations, our prototype always finds all the data races that exist. JRF-E can only give an assurance that no data races exist if the analysis is allowed to run to completion. The published results for JRF-E only show the elapsed processing times for a limited number of traces rather than the times needed for the complete analysis of the programs.

The technology used in the Chord tool is closer to ours. However, due to the hiatus in the maintenance of the SOOT (Vallee-Rai 2000) package, it cannot handle programs compiled with Java 8 or later releases. Our prototype, which uses ASM (Bruneton, Lenglet et al. 2002), is fully compatible with Java 8 and Java 9. We use a much simpler algorithm for
building a context-insensitive call graph during which we build the list of critical sections and their events. By ruthlessly avoiding context-sensitivity we avoid the problems of path explosion. However, it appears that for conventional synchronisation designs our prototype can perform a useful service in detecting coding errors that cause data races. Our algorithm has been implemented as a multi-threaded program. Although it has only been evaluated on a platform with limited multi-threading capability, we have conducted specific tests that show that the use of multi-threading concurrency does deliver significant performance benefits.

The careful and systematic investigation of the performance differentials for our prototype provides great confidence that a production prototype could scale to handle programs of industrial size. This is supported by our limited results obtained with larger programs. Our analysis of soundness and completeness set out in section IV.2.6 indicates that our prototype is well suited to detecting errors in otherwise correct programs.

**IV.8. Conclusions**

We have devised an algorithm to find data races caused by the violation of the implicit access protocol associated with the acquire/release paradigm. We have built a prototype implementation that shows that our "must" and "may" summarisations at a method level together with our approximation for alias resolution make a significant contribution towards developing an implementation whose performance could scale to the processing of industrial-sized programs. This prototype supports a sub-set of the Java language sufficient to represent data race conditions and to investigate the factors relevant to scalability. We chose to regard the development of support for the complete Java language as implementation detail beyond the scope of our research.

We have shown that the AEG notation can be extended to handle the actions of streams, and iterations over collections that avoid explicit enumeration of elements. This expands the scope of our data detection algorithm to include the detection of critical sections where the events include summary events.
The experimental work also shows that, with its set of approximations, our implementation scales well, so that it compares favourably with established systems, such as JavaRaceFinder (Kim, Yavuz-Kahveci et al. 2012) and Chord (Naik, Aiken et al. 2006).

The measures that we have deployed to address the challenge of path explosion are effective. However, we note that they introduce some important limitations. We cannot analyse some styles of coding. Particularly, our approach to alias analysis introduces false positive reports for some potential coding patterns, though we argue that these represent poor coding practice.

These limitations form the motivation for our work on integrating the algorithm within the Java Virtual Machine, which we describe fully in Chapter VI.
Chapter V Restoring sequential consistency

"And as in uffish thought he stood,
The Jabberwock with eyes of flame
Came whiffling through the tulgey wood
And burbled as it came."

"Through the Looking Glass"
Lewis Carroll

In the previous chapter, we described how data race errors can be found by associating memory events with critical sections identified by their de-limiter patterns. Critical section de-limiters provide the locking assurance that uniprocessor conditions apply within the critical section. They also provide the assurance that the re-ordering of instructions cannot occur across the de-limiters. All contemporary CPU architectures guarantee sequentially consistent behaviour within a single thread so that within the critical section it is not necessary to consider the effects of re-ordering. However, the code of the de-limiter patterns cannot rely on this assumption. It must cope with the effects of weak memory architectures. In Chapter III, we described our investigation into the relative performance of a number of well-known de-limiter patterns. Here we ask how we should determine the optimally minimal placement of the least-cost fences needed to assure that a particular de-limiter pattern is sequentially consistent and that it correctly isolates the events within the critical section. In responding to this question, we begin with a recapitulation of prior work in this area including:

- hardware characteristics (Sewell, Sarkar et al. 2010, Alglave, Maranget et al. 2011, Alglave, Maranget et al. 2014, Maranget, Sarkar et al. 2015);
- the advice given in the JSR 133 Cookbook (Lea 2008);
- the use of an Abstract Event Graph (AEG) analysis to investigate the sequential consistency of a program (Alglave 2010); and
- Nimal’s algorithm for the selection and placement of memory fences (Nimal 2014).
We show how the analysis of AEGs can be used to evaluate the effectiveness of fence placements in potential delimiter patterns.

Finally we present our adaptation, for Java, of Nimal's algorithm for the automated selection and placement of memory fences (Nimal 2014). We describe our contribution in modifying the algorithm to match the limitations imposed by the Java environment and providing a prototype implementation in Java.

V.1. Memory models and fence placement

Those working on memory models for processor architectures have developed a set of short program fragments, which they refer to as Litmus tests (Alglave, Maranget et al. 2011). These are specifically designed to expose the, often surprising, results of weak memory execution. Because these fragments are so short, it is possible to analyse them manually according to the memory models and predict the results that may be observed. The work includes C/C++ implementations of these tests that were executed in a test framework that performs many executions of the code over an extended time period and records the observed results. Other authors have used their memory models to predict how the placement of memory fences would affect the observed results and what placements will yield sequentially consistent execution. All this work was specific to the C/C++ environment.

Lea in his JSR 133 CookBook (Lea 2008) provided an analysis of the various types of fence discussed in the Litmus tests and offered advice on the way in which access to volatile variables might be implemented within the Java Virtual Machine. He noted that on an x86 architecture the need for fence placement reduces to placing a full fence immediately after a write to a volatile variable. No other fences are required. His analysis pointed out that fence selection and placement for other architectures was more complicated as fences were more often required to restore sequential consistency. He observed that the more relaxed architectures often respected address dependencies so that an explicit memory fence might be replaced by an
artificially created address dependency, but effectively dismissed this option as too hard to implement.

Alglave, in her doctoral thesis (Alglave 2010), proposed a generic framework for describing the execution of programs under weak memory that is referred to as an Abstract Event Graph (AEG). We provided an introduction to this work in Chapter II and used the AEG concept to provide the necessary theoretical foundation for our work on the detection of data races described in Chapter IV. We now consider the application of other consequences of Alglave’s work, notably the ability to detect the lack of sequential consistency.

The nodes of an AEG are the read and write actions that access shared memory. These are called events. The edges include:

- **program order**, that reflects the order of the events as written;
- **competing-pair** relationships, that reflect the fact that events in different threads may be accessing the same shared memory location.

![Diagram showing AEG for MP Litmus test](image)

In Figure 97 we show the AEG for the Message-Passing (MP) Litmus test where there are two concurrently executing threads. Thread 0 comprises the events (1) and (2) and Thread 1 comprises events (3) and (4). There is a competing pair relationship between events (1) and (4) because they access the same variable. This identifies the fact that the value read by (1) may be the value written by (4) or may be some other value. The competing pair relationship records the fact that event (1) may observe all the effects of weak memory as they apply to the variable \( a \). There is a similar competing pair relationship between events (3) and (4) regarding the variable \( b \). Although the competing pair relationships do not represent execution paths, their existence may be used to determine whether the graph contains invalid executions that represent a lack of sequential consistency. Alglave provided a proof that it is not necessary to evaluate all the execution paths in a
program. It suffices to form the AEG and search it for cycles. For example, the sequence (1), (2), (3), (4), (1) represents such a cycle. The program-order relationships are uni-directional but the competing pair relationships are bi-directional. If the graph is acyclic then it is sequentially consistent. If not, then planting memory fences to break the cycles restores sequential consistency. For example, the cycle (1), (2), (3), (4), (1) would be broken by placing fences in the edges 1-2 and 3-4.

In his doctoral thesis, Nimal (2014) offers the opinion that because reasoning about the selection and placement of memory fences is difficult, it should be automated. Although Alglave’s theoretical work is of general applicability, Nimal’s work is highly specific to the C language environment. His analysis was designed to process C program source and used many specific tools developed by the Oxford Computing Laboratory where he worked. As he remarked in his Acknowledgements, his implementation involved a deal of "diving" into locally maintained code. He also chose to generate the machine code fence instructions that pertained to a particular CPU architecture. These characteristics made it technically very difficult to re-use his implementation directly, so we chose to re-implement his design in Java and modify it as required to conform to the constraints of the Java environment. For example, we start from Java bytecode, rather than C source and we target the VarHandle fence methods described in section II.1.4 of Chapter II, rather than their implementations as particular machine code instruction sequences. We present the design and development of a prototype implementation of this approach in section V.3.

In the next section we present a worked example of the application of Alglave’s AEG analysis technique.

V.2. Manual analysis
In this section we provide a sample of the use of the AEG analysis method to investigate the sequential consistency of a multi-threaded program. We show how a lack of sequential consistency can be detected by finding cycles in an AEG and how this can be used to determine the placement of memory fences to restore sequential consistency. We take as our example the
message-passing pattern with one thread that writes to the variable $x$ and another that reads it. The essential Java fragments for this pattern are shown in Figure 98. The `send()` and `receive()` methods would be executed in different threads.

```java
public class MP {
    // the lock variables
    static int u = 0;
    static int v = 0;
    // the guarded variable
    static int x = 0;
    ...
    public void send() {
        int r;
        r = u;     // read u
        x = 42;    // write to x
        v = 43;    // write to v
    }
    public void receive() {
        int r;
        int s;
        r = v;     // read v
        s = x;     // read x
        u = 44;    // write to u
    }
}
```

*Figure 98* - Java code fragments for MP pattern

The corresponding AEG is shown in Figure 99. The double-ended links represent competing-pair relationships. The single-ended links represent program order relationships.

*Figure 99* - MP pattern with guarded events

V.2.1. Alglave's analysis of AEG cycles

The first step in the process is to enumerate the cycles in the AEG. The list in Figure 100 shows the cycles, which we have numbered sequentially for
reference purposes together with the list of events in each cycle. Each event is identified by the number assigned in Figure 99.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Event list</th>
<th>Cycle</th>
<th>Event list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,2,3,4,5,6,1</td>
<td>19</td>
<td>4,5,6,1,2,3,4</td>
</tr>
<tr>
<td>2</td>
<td>1,2,3,4,5,6,3</td>
<td>20</td>
<td>4,5,6,1,2,5</td>
</tr>
<tr>
<td>3</td>
<td>1,2,3,4,5,2</td>
<td>21</td>
<td>4,5,6,1,2,5</td>
</tr>
<tr>
<td>4</td>
<td>1,2,3,6,1</td>
<td>22</td>
<td>4,5,6,3,4</td>
</tr>
<tr>
<td>5</td>
<td>1,2,5,6,1</td>
<td>23</td>
<td>4,5,2,3,4</td>
</tr>
<tr>
<td>6</td>
<td>1,2,5,6,3,4,5</td>
<td>24</td>
<td>4,5,2,3,6,1,2</td>
</tr>
<tr>
<td>7</td>
<td>1,6,3,4,5,6</td>
<td>25</td>
<td>4,3,6,1,2,3</td>
</tr>
<tr>
<td>8</td>
<td>1,6,3,4,5,2,3</td>
<td>26</td>
<td>4,3,6,1,2,5,6</td>
</tr>
<tr>
<td>9</td>
<td>2,3,4,5,2</td>
<td>27</td>
<td>5,6,1,2,5</td>
</tr>
<tr>
<td>10</td>
<td>2,3,4,5,6,1,2</td>
<td>28</td>
<td>5,6,1,2,3,4,5</td>
</tr>
<tr>
<td>11</td>
<td>2,3,4,5,6,3</td>
<td>29</td>
<td>5,6,1,2,3,6</td>
</tr>
<tr>
<td>12</td>
<td>2,5,6,1,2</td>
<td>30</td>
<td>5,2,3,4,5</td>
</tr>
<tr>
<td>13</td>
<td>2,5,6,3,4,5</td>
<td>31</td>
<td>5,2,3,6,1,2</td>
</tr>
<tr>
<td>14</td>
<td>3,4,5,6,3</td>
<td>32</td>
<td>6,1,2,3,6</td>
</tr>
<tr>
<td>15</td>
<td>3,4,5,6,1,2,3</td>
<td>33</td>
<td>6,1,2,3,4,5,6</td>
</tr>
<tr>
<td>16</td>
<td>3,4,5,2,3</td>
<td>34</td>
<td>6,1,2,5,6</td>
</tr>
<tr>
<td>17</td>
<td>3,6,1,2,3</td>
<td>35</td>
<td>6,3,4,5,6</td>
</tr>
<tr>
<td>18</td>
<td>3,6,1,2,5,6</td>
<td>36</td>
<td>6,3,4,5,2,3</td>
</tr>
</tbody>
</table>

**Figure 100 - List of cycles for MP example AEG**

The next step in the process is to filter the program order edges from all the edges in the cycles. These are the only physical edges and are, therefore, the only edges that can be broken by the placement of memory fences.

<table>
<thead>
<tr>
<th>Thread</th>
<th>PO events</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,2,3</td>
<td>1,2,3,4,15,17,19,20,25,28,29,32,33</td>
</tr>
<tr>
<td></td>
<td>1,2, not 3</td>
<td>5,6,10,12,18,21,24,26,27,31,34</td>
</tr>
<tr>
<td></td>
<td>2,3, not 1</td>
<td>8,9,10,11,16,17,23,24,25,30,31,36</td>
</tr>
<tr>
<td>1</td>
<td>4,5,6</td>
<td>1,2,7,10,11,14,15,19,20,21,22,33,35</td>
</tr>
<tr>
<td></td>
<td>4,5, not 6</td>
<td>3,6,8,9,13,16,23,24,28,30,36</td>
</tr>
<tr>
<td></td>
<td>5,6, not 4</td>
<td>5,6,7,12,13,18,26,27,28,29,34,35</td>
</tr>
</tbody>
</table>

**Figure 101 - Program order edges with their cycles**

Accordingly, we prepare, for each thread, the list of event sequences in their program order together with the list of cycles in which they participate. The cycles are identified by the numbers assigned in Figure 100. This analysis is shown in Figure 101. It reveals that, in common with many de-limiter patterns, there are no edges that do not participate in cycles. On the contrary, all the program order edges participate in many cycles.
To allow for the re-ordering effects of weak memory, the event sequence 1,2,3 must be represented by the transitive closure over the program order so that the full set of edges to be considered is {1-2, 2-3, 1-3}. The same argument must be applied to the event sequence 4,5,6.

In Chapter II, section II.1.2, we introduced the notion of restoring sequential consistency through the insertion of memory fences into the program-order edges between memory access events. In Chapter II, section II.1.4, we introduced the JSR 133 CookBook that defined the various generic fence types that are required and JEP 193 as a part of Java 9, which defines the set of fences provided by the methods of the VarHandle class. If we consider a target architecture represented by the fence methods defined in JEP 193 then we have the following available fences:

<table>
<thead>
<tr>
<th>JEP 193 fence</th>
<th>Implemented fences</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquireFence</td>
<td>LoadLoad + LoadStore</td>
</tr>
<tr>
<td>releaseFence</td>
<td>LoadStore + StoreStore</td>
</tr>
<tr>
<td>loadLoadFence</td>
<td>LoadLoad</td>
</tr>
<tr>
<td>storeStoreFence</td>
<td>StoreStore</td>
</tr>
<tr>
<td>fullFence</td>
<td>LoadLoad + LoadStore + StoreLoad + StoreStore</td>
</tr>
</tbody>
</table>

Figure 102 shows that JEP 193 provides no individual StoreLoad and LoadStore fences. Instead the composite acquireFence, releaseFence and fullFence methods are provided. Examination of Figure 101 shows that, in this example, every one of the edges must be broken by the placement of a fence. The fence used must be at least as strong as that determined by the type of edge as set out in Figure 103.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Type</th>
<th>Suitable Fences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>LS</td>
<td>fullFence, releaseFence, acquireFence</td>
</tr>
<tr>
<td>1-3</td>
<td>LS</td>
<td>fullFence, releaseFence, acquireFence</td>
</tr>
<tr>
<td>2-3</td>
<td>SS</td>
<td>fullFence, releaseFence, storeStoreFence</td>
</tr>
<tr>
<td>4-5</td>
<td>LL</td>
<td>fullFence, acquireFence, loadLoadFence</td>
</tr>
<tr>
<td>4-6</td>
<td>LS</td>
<td>fullFence, releaseFence, acquireFence</td>
</tr>
<tr>
<td>5-6</td>
<td>LS</td>
<td>fullFence, releaseFence, acquireFence</td>
</tr>
</tbody>
</table>

Figure 103 - Required fence types by edge type
It appears that those who provide hardware implementations of CPU architectures are generally unwilling to provide data regarding the execution costs of memory fence instructions. Consequently, it is not possible to provide a quantitative estimate of the relative costs of different fences. Nimal (2014) suggests that qualitatively

\[ \text{fullFence} > \text{releaseFence} > \text{acquireFence} \]

but provides no justification for this relativity and makes no attempt to quantify it. His algorithm for the selection and placement of machine code fence implementations arbitrarily assigns successive increasing integer values to the cost of different instructions. This is clearly correct only in a qualitative sense.

The results obtained by Ritson and Owens (2016) indicate that the Store-Store fence is significantly more expensive than other fences. However, the magnitude of the effect differs on the ARM and POWER architectures. The information from Haley (2017) further suggests that it differs on different implementations of the same architecture.

Accordingly, we chose to note the inaccuracy and continue with Nimal's relativity values.

Analysis of the data in Figure 103 together with this assumed relative cost for fences shows that the sequence 1-2-3 can be sufficiently broken by an acquireFence in the edge 1-2 and a releaseFence in the edge 2-3. The same pattern suffices for the sequence 4-5-6. However, because the options are different for the two threads, it is intuitively obvious that there will be cases involving more complicated patterns of events where the choice of fences will be different. Nimal (2014) provides the analysis of an AEG involving six threads where the overlap of the cycles in the central thread offers significant scope for this sort of optimisation.

In our example, we have constructed an AEG that represents a single execution of the events 1, 2 and 3 in one thread interacting with a single execution of the events 4, 5, and 6 in a second thread. This has been done to show the principle of the application of the analysis technique. However, it is obvious that in any realistic program the events would be enclosed within an iteration so that each of the sequences would be continually executed. To
provide a model of such a program, we would need to provide at least two instances of the appropriate pattern of events in each thread. We have not presented such an example because doubling the number of events more than doubles the number of competing pair relationships and the consequential growth in the number of cycles to be identified makes manual analysis very time-consuming. This is a strong motivation for the use of automation that we describe in section V.3.

V.2.2. JSR 133 Cookbook

We applied the full analysis technique to a substantial selection of the Litmus tests. In these simple de-limiter patterns, a small number of events access a comparable number of shared variables. This yields the observed result that every program order edge participates in at least one cycle and must, therefore, be broken with a fence. For these simple de-limiter patterns, it is, therefore, a good working approximation to assume that all the program order edges require fences, without going to the effort of enumerating the cycles.

We compare the results from this conclusion with those obtained by applying the recipe provided by Lea (2008) for the implementation of volatile variables within the Java Virtual Machine (JVM). His recipe assumes that determining the preceding or following instruction is too hard. He recommends a worst-case conservative approach to the placement of memory fences around events that access volatile variables. We summarise his recipe in Figure 104.

<table>
<thead>
<tr>
<th>Volatile event</th>
<th>Fence before</th>
<th>Fences after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store</td>
<td>StoreStore</td>
<td>StoreLoad</td>
</tr>
<tr>
<td>Load</td>
<td>LoadLoad + LoadStore</td>
<td></td>
</tr>
</tbody>
</table>

Figure 104 - JSR 133 Cookbook recipe

The JEP 193 acquireFence is, of course, the LoadLoad + LoadStore combination (see Figure 102).

The default use of volatile within the example shown in Figure 99 would be to make the u and v variables volatile but assume that the de-limiter pattern delivers sequential consistency so that x does not need to be
made volatile. If we then apply the JSR 133 recipe to event (1), we obtain the same result as that predicted by our sequential consistency analysis, that is, the insertion of an acquireFence in the 1-2 edge. However, when we consider event (3), the JSR 133 recipe recommends a StoreStore fence before the event and a StoreLoad fence afterwards. In the simple, single iteration example that we show in Figure 99 there is no edge that follows event (3) so that the process does not yield the same result as that predicted by the Cookbook. In Figure 105 we show the AEG for two iterations of the events in Thread 0. Our analysis of this pattern shows that the program order edge that follows the event (3) is a StoreLoad edge between event (3) and event (1) in the next iteration. This more extensive analysis concurs with the JSR 133 recipe.

In the x86/AMD architecture, the architecture is generally sequentially consistent so that the only edge that requires a fence is the StoreLoad edge. In all other cases, the architecture performs cache coherency actions automatically. The only fence provided is a fullFence. This may be invoked by using the lock prefix on instructions where it is allowed or, more explicitly, by the mFence instruction. All the other fences may be
implemented as no-ops or simply omitted. Pragmatically, this greatly reduces the motivation to consider whether these fences are really needed. Conversely, in the ARM architecture, all fences require a *dmb* instruction or some other instruction sequence that triggers cache coherency actions within the processor. Although the ARM architecture does not, generally, provide sequential consistency, it does provide some more limited guarantees:

- Sequential consistency is provided to the instructions within a single thread as observed by that thread;
- In the cases of LoadStore and LoadLoad edges, sequential consistency may be caused by address dependencies.

Nimal’s selection and placement algorithm assigns cost values for different fence implementations that assume that address dependencies are cheaper than the use of the *dmb* fence. If this is true at all, the difference is quite marginal and relates to the cost of executing the *dmb* instruction as opposed, for example, to the cost of executing the instructions needed to artificially introduce an address dependency. Both these costs of execution within the thread are a small fraction of the cost to the overall performance of the processor that results from the cache coherency actions themselves. The cache coherency actions are the same irrespective of the way in which they are triggered. Although there may well be some small efficiency gains to be made from replacing *dmb* instructions by other instruction sequences, the bigger gains are to be made by analysing the AEG to detect where a fence is not required because the existing instructions already include an address dependency.

![Figure 106 - AEG for MP with common object](image)

Figure 106 - AEG for MP with common object
For example, if the flag variables, \( u \) and \( v \), are made fields of the same object as \( x \), then an address dependency certainly exists at the bytecode level. If this common object is called \( c \), then the AEG for our example becomes transformed into that shown in Figure 106.

If this same address dependency becomes reflected in the machine code generated by the Java Virtual Machine, then there is scope for an optimisation that omits the \textit{dmb} instruction recommended by the JSR 133 Cookbook. This avoids invoking the cache coherency overheads twice. We examine how this may be mechanised within the Java Virtual Machine (JVM) in Chapter VI.

In this section we have provided a worked example of the use of AEGs and cycles to place the memory fences needed to ensure the sequential consistency of a program. In the next section we discuss our adaptation of Nimal’s work on automation so that it conforms to the features and constraints of the Java environment.

**V.3. Our extension of automation to the Java environment**

Nimal’s work (Nimal 2014), introduced in section II.1.6 of Chapter II, was restricted to the C/C++ environment. We have extended Nimal’s algorithms to make them suitable for use in the Java environment with the following significant differences:

- Start with the bytecode contained in Java class files;
- Provide a mitigation for the problem of resolving addresses;
- Search for critical cycles using a novel parallel-processing algorithm;
- Choose fences from those supported by the VarHandle class;
- Leave the implementation of these generic fences in the target architecture to the Java Virtual Machine.

**V.3.1. Our transformation of bytecode to an Abstract Event Graph (AEG)**

As a result of our work described in Chapter IV, section IV.5, we have a set of classes that transforms Java bytecode into a Java data structure that holds all the relevant information regarding Java Classes and Methods. Without the implementation of an algorithm to transform bytecode into an AEG, it is not possible to accomplish the task of applying Nimal’s algorithm to Java
programs. The essential point is that the implementation work that supports Chapter IV is modular and has well-defined interfaces so that it can be re-used to support the algorithm described in Chapter V. This is an important piece of novel practical engineering. We illustrate this in Figure 107 that recalls Figure 80.

![Figure 107 - Analysis framework](image)

This is sufficient to support both the analysis of critical sections and the analysis of the AEG for the selection and placement of fences.

### V.3.2. Resolving addresses

Nimal (2014) quotes Alglave’s conditions for the validity of her AEG:

- **C1.** the instructions themselves: the processor cannot skip some instructions—unless told to do so—or execute some unwritten instruction;

- **C2.** the control flow graph (CFG) that is formed from all the conditional and unconditional jumps inside a function (including loops), should be statically resolved;

- **C3.** the functions called should be statically resolved;

- **C4.** the threads running should be statically determined;

- **C5.** all the expressions should be evaluated, including the values read from and written to memory (including the addresses of memory locations).

These criteria are similar to and compatible with those we define in section IV.2.2 of Chapter IV. The fulfilment of C1, C2 and C3 follows naturally from our static analysis of the bytecode. We control the effect of C4 by applying the AEG analysis for sequential consistency to code that is specifically known to be invoked within concurrently executing threads. It is easy to identify events, but difficult to distinguish between events that access
class instance variables. In analysing an AEG for the placement of memory fences we are concerned to ensure that the placement is absolutely sound even if it is over-approximate. As a result, there are fewer objections to our approximation technique of treating all class instance variables as if they are static variables. Our implementation specifically excludes the analysis of circumstances that generate false negatives. The effect of false positives is simply to add additional spurious competing-pair relationships.

V.3.3. Our derivation of competing pair relationships

Once we have derived the basic program order list of memory events, we apply the multi-threaded condition as follows. We assume that there are two threads, each of which executes the same list of events. We then process every event in the first thread by searching for events with the same variable in the second thread and setting up competing-pair relationships between the events in the two threads. We then merge all this information into a composite Abstract Event Graph (AEG) that covers the two threads.

V.3.4. Finding critical cycles - our novel implementation

Tarjan’s algorithm (Tarjan 1972) for finding cycles in a graph is a Depth-First-Search (DFS) that maintains a list of the nodes in the current path. As each new node is processed, the list is searched to detect the re-occurrence of a node. More efficient variants of this algorithm modify the original nodes as they are processed to save time and space. This means that these variant algorithms must use serial processing. Nimal (2014) showed empirically that, by itself, Tarjan’s algorithm is too computationally expensive. However, in this application, the incorporation of Alglave’s heuristics is crucial to providing a viable solution. Our novel contribution is the integration of our particular variant of the algorithm with the Java stream framework.

We have extended the Abstract Event Graph to include events that summarise the actions of a stream invocation. This allows us to extend our general-purpose search for data races to analyse programs that process Collections using the streams feature.

A key aspect of streams is the Spliterator interface. Every Collection must define a subclass that implements this interface. The most important
methods of this interface are tryAdvance() and trySplit(). The tryAdvance() method replaces the hasNext() and next() methods of the Iterator interface. If there is a next element, tryAdvance() performs the specified action on that element and returns true. If there is no next element, tryAdvance() returns false.

The trySplit() method is key to the support for parallel processing. Based on the size of the collection being handled and the number of threads effectively supported on the processor, the method decides whether it is appropriate to split the collection. If it decides against a split, it returns null. If the collection is split, it is divided into two dis-joint subsets, one of which is retained by the current Spliterator instance. The other is entrusted to a new Spliterator instance, which is returned by the method. This means that these two Spliterator instances can work independently without locks.

If parallel processing is asked for, the framework repeatedly invokes trySplit() and forks off the returned Spliterator instances to threads until null is returned. It then organises the joins and the merging of results.

The benefit of these features is that it is relatively easy to organise a parallel algorithm for the processing of a new style of Collection. Provided that the class includes a sub-class that correctly implements the Spliterator interface and conforms to its few constraints, all the thread manipulation actions are handled within the framework without intervention by the developer.

In our work we have implemented the search for critical cycles in an Abstract Event Graph by defining a novel Graph class and implementing the search within a specifically designed Spliterator sub-class. Alglave's heuristics reduce the execution time to a manageable value. The integration with the stream framework ensures that its implementation requires no further development to exploit the benefits of multi-threaded execution. Furthermore, this exploitation of standard Java features ensures that our implementation will automatically benefit from future enhancements of the stream framework. Although our experimental platform provided only limited multi-threading capability, we were able to demonstrate that invoking the parallelStream option significantly reduced the runtime of the
algorithm. Clearly, the use of a multi-core, multi-threaded processor would yield further tangible benefits.

We chose to leave open the possibility of parallel execution by using an algorithm that does not alter the graph that is being searched. In our implementation of this novel algorithm we created a Graph class to hold the events as nodes and encapsulated the maintenance of the lists of encountered nodes within a Spliterator sub-class. The search process is initiated by invoking the stream() method of the Graph class. This stream delivers a list of container classes each of which holds the event that caused the cycle to be detected together with the list of events that constitute that cycle.

Measurements of the performance of our initial implementation demonstrated that this exhaustive search is too costly to be practical. We then followed Nimal's example and incorporated Alglave's heuristics. We achieved this integration efficiently by including them within our Spliterator sub-class. We discuss the detail of these heuristics in section V.3.5. They dramatically improved the elapsed runtime of the process.

Regarding multi-threading, we obtained evidence that invoking the parallelStream() option halved the runtime on our test platform compared with the serialStream() option.

V.3.5. Alglave's heuristics

There are two important criteria that provide for the rapid elimination of unfruitful exploration paths:

- There are at most two accesses per thread in a critical cycle;
- There are at most two writes and one read per variable in a critical cycle.

The first of these criteria guarantees that, if $t$ is the number of threads, then the number of events in a critical cycle cannot be greater than $2t$. Deriving a corresponding limit for the second criterion is more difficult. In the worst case, if there is an infinite number of threads, then there may be an infinite number of write events in a cycle before the first read event occurs or an infinite number of read events before the second write event occurs.
However, we can say that it is reasonably probable that this criterion will eliminate cycles whose size is \(3v\) where \(v\) is the number of variables. In our test case, where \(t\) is 2, the first criterion eliminates cycles with more than four events. If we consider two threads that are synchronising using atomic CompareAndSet operations, then the second criterion would restrict the size of its critical cycles to three events.

These limits on the length of cycles are crucial to the achievement of acceptable runtimes for the finding of critical cycles.

V.3.6. Evaluation

We evaluated our implementation in two stages. First we investigated its correctness by using a sub-set of the Litmus tests (Alglave, Maranget et al. 2011) and then we used a variant of Dekker’s mutual exclusion algorithm (Dijkstra 1971) to provide a more extensive test of capacity and scalability.

V.3.6.1. Litmus tests

We validated our implementation by exercising it against that sub-set of the Litmus tests that demonstrate the restoration of sequential consistency by the placement of memory fences. We removed the placed fences and then verified by inspection that the placements and types of fence predicted by our algorithm accorded with those we had removed.

Maranget, Sarkar et al. (2015) classified the Litmus test families with a compact notation that we use here. Our selected tests are limited to two memory events in each of two threads. Maranget, Sarkar et al. specifies the tests by giving a diagram of the memory events and program order edges for each test. We took the specification of the tests directly from these diagrams, which we include here as Figure 108. The figure includes six tests. In each case, the threads are shown as vertical sequences of events, with the program-order represented by the arrow. The notation conforms to that described for Abstract Event Graphs. Below each test is its identifier, as specified in the compact notation previously mentioned.
We chose to use the features of the JUnit test framework (Gamma and Beck 1999) to manually construct the AEG for each test as the nodes and edges of our AEG graph class. We gave our AEG class a static method that found the nodes belonging to one thread and scanned for the nodes in the other thread. It found pairs of nodes that accessed the same variable and created the correct competing-pair relationships as edges. This addition made the AEG complete.

We were able to find cycles, construct the Integer Linear Program (ILP) and run the solver all within acceptable elapsed times of the order of tens of milliseconds.

We first verified that the algorithm gave the known correct results for the selection and placement of fences for the six Litmus tests. We encapsulated the definitions of fences for hardware architectures within a separate Java class. We developed definitions for the ARM and POWER architectures and also for an "architecture" that represented the model supported by the VarHandle fence methods. The POWER architecture has a greater variety of fence instructions available than the ARM architecture, so we exercised our system with the definitions appropriate to the POWER architecture to exploit the full versatility of our implementation and provide results that we could compare with the published "correct" results for the Litmus tests (Maranget, Sarkar et al. 2015).
Figure 109 - Results for Litmus tests

Figure 109 shows the output from our prototype for the six selected Litmus tests. The short code identifies the Litmus test. The expected results show the fences that should be placed. The rest of the printout shows the events for the two threads and the fences placed by the algorithm. The mnemonics used to identify the fence instructions are taken from the POWER instruction code.

In the C environment it is possible to force the generation of specific sequences of machine instructions. In Java bytecode, the explicit placement of fences can be achieved only by the use of the VarHandle fence methods.
Accordingly, we extended our system to generate the placements for an “architecture” that presented those methods. The results of the tests against this architecture are shown in Figure 110.

These tests were always run as individual unit tests within a test suite. The elapsed times for these tests showed that cost of searching for cycles and solving the derived Integer Linear Program (ILP) is represented by 5-20ms.

**V.3.6.2. Dekker’s algorithm**

We then encoded the AEG corresponding to an implementation of Dekker’s mutual exclusion algorithm as quoted by Dijkstra (Dijkstra 1971). The AEG for a single instance of the algorithm is shown in Figure 111. We used two threads and two instances of the AEG in each thread to simulate the effect of a continuously cycling implementation. The composite AEG contained 36 events and similar number of competing-pair relationships. The first thread had two successive instances of this pattern to represent the execution of the pattern in a loop. The second thread had a similar pattern with the use of the variables u and v interchanged.

![AEG for Dekker’s mutual exclusion algorithm](image)

We chose Dekker’s algorithm solely because it is generally familiar, not because it is actually particularly suitable, given the contemporary general
availability of CompareAndSet instructions or their equivalents. Figure 112 shows the test output for this test.

```plaintext
Dekker's algorithm
Nodes 41
Cmps 100
211 cycles in 146ms
14 fences in 1413ms
```

Figure 112 - Test result for iterated Dekker's algorithm in two threads

For comparison, the corresponding output for a test with two threads each with only one instance of the pattern shown in Figure 113.

```plaintext
Dekker's algorithm
Nodes 21
Cmps 25
90 cycles in 118ms
6 fences in 290ms
```

Figure 113 - Result for single instances of Dekker's algorithm

We note that the time taken to find the critical cycles scales well with the number of nodes and the number of competing-pair relationships. An early run of the test shown in Figure 112 but without Alglave's heuristics was still running after 45mins. This shows that these heuristics are the most important component of the code that searches for cycles.

The time taken to build and solve the ILP inequalities is clearly not scaling well. This test case is not a large piece of code and the runtime is already measured in seconds.

V.3.7. Observations

We report the following observations.

The choice of algorithm used for finding cycles in the AEG is largely irrelevant. They all yield impractically long run times even for this relatively small example. Alglave's exclusion heuristics for critical sections are much more important and make the process acceptably fast. We built them into our Spliterator implementation so that we can use either the serialStream or the parallelStream options for the best performance on multi-core multi-thread platforms.

As anticipated, the algorithm places memory fences that accord with the result of applying the advice set out in the JSR 133 CookBook (Lea 2008) for volatile variables. In the x86 environment, this means that the two flag
variables and the turn variable are treated as volatile so that a full fence lock: add is placed after every write access to these variables. When the algorithm is run as if it were selecting and placing fences for the ARM architecture, full dmb fences are required both before and after the write accesses and after the read accesses.

Nimal’s fence selection algorithm uses assigned cost values so that some fence types are considered more expensive than others. For the ARM architecture, a higher value is assigned to dmb than to dp (Nimal’s shorthand for a fabricated address dependency). Working in the C environment, Nimal was able to insert machine code instructions into C source and to force the compiler to omit code optimisations such as re-ordering of instructions. Java 9 provides the VarHandle fence methods that can be invoked directly from Java source code. At the bytecode level they appear as method invocations and the Java Virtual Machine provides compiler intrinsics so that they are mapped on to the equivalent machine code memory fence instructions. However, the extent to which it is possible to use bytecode features to substitute an artificial address dependency or other code patterns for a Load-Load fence, a Load-Store fence or the VarHandle.acquireFence(), is limited by the constraints imposed by the definition of Java bytecode. Address arithmetic is forbidden. Addresses may not be cast to integers nor integers to addresses. This means that the only relevant fence architecture that can be used for optimisation by a static analysis is that represented by the VarHandle methods. The bytecode is specifically intended to be hardware agnostic. We are forced to reduce the scope of Nimal’s algorithm and recognise that the optimisation of the machine code implementation of the VarHandle methods must be delegated to the Java Virtual Machine. We contend that the analysis leading to these consequences is a novel and valuable contribution.

Because of these constraints, we limited our static analysis and fence optimisation to the selection and placement of fence types that are supported as VarHandle methods. The optimisation of the machine code corresponding to these fence types is delegated to the Java Virtual Machine.
V.4. Conclusions

Our experimental results in Chapter III showed that using alternative delimiter patterns to the standard `synchronized` construct delivers significant performance improvements particularly where there is significant contention for the shared data. The conventional advice for the placement and implementation of memory fences, while satisfactory for relatively strong memory models such as that provided by the x86 architecture, is inefficiently conservative when used for the implementation of the Java Virtual Machine on architectures with weak memory models. In this chapter, we have shown that, with appropriate adaptations, the technique of Abstract Event Graph (AEG) analysis can be used on Java bytecode to evaluate the correctness of delimiter patterns that purport to deliver sequential consistency more efficiently.

This research provided a valuable insight into the best way to efficiently ensure sequential consistency. We have used this knowledge to design and build an efficient thread-safe DataStore that facilitates the controlled access to a stored data hierarchy that is shared across many threads. We present this design in Chapter VII.

We have further shown that the algorithm developed for C language code for the automated selection and placement of memory fences, can be adapted for use on AEGs derived from Java bytecode. We note that our implementation of this algorithm suffers from scalability issues. Processing AEGs with a reasonable number of events linked by a reasonable number of competing-pair relationships is computationally intensive, even when heuristics are applied. This imposes a practical limit on the usefulness of the algorithm.

This automation algorithm delivers an optimised fence placement for the machine-independent fence architecture represented by the VarHandle methods. In the next chapter we discuss the implementation within the Java Virtual Machine (JVM) of the techniques discussed in this chapter for providing sequential consistency on weak-memory architectures. In particular, we show how to optimise the implementation of redundant fences.
Chapter VI Optimisations within the JVM

"He who would sup with the Devil should have a long spoon."

14th. cent. Proverb

We divide this chapter into three parts. In the first part, we describe our research into the current implementations of CompareAndSet and the volatile construct, and explore the ramifications for Java Virtual Machine implementations that are intended to target architectures with weaker memory models. These other architectures generally have alternative ways of providing CompareAndSet functionality and of implementing memory fences that differ significantly from those provided by the x86 architecture. We propose extensions to the set of VarHandle methods to recognise the particular characteristics of weak memory architectures. This work is described in section VI.1.

In Chapter V, section V.1 we recalled Lea's discussion of the alternative instruction sequences that may be used to effect cache coherency on weak memory architectures. In the second part of this chapter, we show how code can be introduced into the execution of the Just-In-Time (JIT) compilers that form part of the Java Virtual Machine (JVM). This allows us to consider optimising the implementation of memory fences. The Wikipedia (https://en.wikipedia.org/wiki/List_of_Java_virtual_machines 2017) lists 24 active JVM implementations, of which half are open source and half are proprietary. There are at least as many listed as inactive. We have chosen to restrict our research to the standard JVM and the Graal project (http://openjdk.java.net/projects/graal/). The Graal JIT compiler is implemented in Java, which satisfies our need for accessibility, and is part of the Open JDK initiative, which ensures compatibility with the latest release of Java. We provide a design and describe a proof of concept for modifying the nodes of the Graal Internal Representation (IR) Graph to eliminate redundant fences and to substitute address dependencies for an acquireFence. In section VI.2 we provide a more extensive introduction to the features of the Graal compiler and following this, in section VI.3, we
describe our use of these features to implement optimisations of the implementation of fences.

Finally, in section VI.4, we consider the benefits and drawbacks of hosting our static analysis algorithms within the Graal environment. We provide a design and proof-of-concept that shows how we transform the Graal Internal Representation (IR) graph into an Abstract Event Graph (AEG) that forms the starting point for our static analysis algorithms.

**VI.1. Our analysis of JVM bias towards x86 architecture**

In our research we have examined the available documentation, inspected the classes in the Java Development Kit (JDK) and, where possible, examined the machine code generated by the Java Virtual Machine (JVM). This provides a body of evidence, some of which we present here, that, prior to Java 9, the classes in the JDK and the JVM itself were built to provide support for the x86 architecture. We argue in the following sub-sections that this impairs the performance of the executed Java code so that it will incur significant inefficiencies if the design of the JVM is naïvely ported to weak-memory architectures.

**VI.1.1. CompareAndSet and weakCompareAndSet**

The IBM 370 series architecture (IBM 1983) defined an instruction that provided *CompareAndSet* functionality. This greatly reduced the complication of organising co-operation between the many processes active within an operating system. In particular, it made possible the development of operating system support for efficient symmetric multi-processor configurations. The x86 architecture provides this functionality.

Here we use as an example the AtomicInteger class, though similar methods are provided for all the classes in the *java/util/atomic* package.

The *compareAndSet* method applied to a shared variable \( v \) atomically implements the logic as shown in Figure 114.

\[
\begin{align*}
  r &= v; \\
  \text{if} \ (r \neq \text{expected}) \ {\text{return} \ false;} \\
  v &= \text{newvalue}; \\
  \text{return} \ true;
\end{align*}
\]

*Figure 114 - Pseudo-code for compareAndSet*
The specification states that the variable \( v \) shall be treated as if it had been declared to be \textbf{volatile}.

The corresponding \texttt{weakCompareAndSet} method has the same logic but omits the requirement for the \textbf{volatile} treatment of the variable \( v \).

In the x86 architecture, the \texttt{lock:xchg} instruction performs a \texttt{CompareAndSet} operation and has the effect of a full memory fence. The \texttt{xchg} instruction cannot be used on a shared memory address without the \texttt{lock} prefix, so there is no way to implement \texttt{weakCompareAndSet}. This is confirmed by an inspection of the source code of the Java 8 version of the AtomicInteger class. This shows that both the \texttt{compareAndSet} and the \texttt{weakCompareAndSet} methods map on to the same method of the Unsafe class. As the detailed specification of the \texttt{weakCompareAndSet} method, documented within the source code of the AtomicInteger class, uses the qualifier "may", the effective substitution of the full \texttt{CompareAndSet} functionality is a legitimate, though potentially costly, over-approximation.

Weak memory RISC architectures, such as ARM (ARM_Holdings 2014) and POWER (May, Silha et al. 1994), do not directly support \texttt{CompareAndSet} functionality. Instead they use the LoadLinked/StoreConditional paradigm, which we introduced in Chapter II. The LoadLinked instruction retrieves the value of the shared variable and sets a processor flag. If the flag is set, a Store action to the shared variable, or any "adjacent" variable, un-sets the flag. The StoreConditional instruction checks the flag. If it is still set, it writes the supplied new value to the variable. If the flag has been un-set, it does not perform this store action. Some implementations return a condition code to indicate the success or failure of the StoreConditional instruction. There is no definition of "adjacent". Some early ARM implementations used a flag that covered the whole of the shared memory. Most contemporary implementations seem to have chosen to place flags on cache-lines and the equivalent sections of shared memory. Using these instructions, \texttt{weakCompareAndSet} functionality may be implemented with:

\begin{verbatim}
r = ll(v);
if (r != expected) {return false;}
return sc(v, newvalue);
\end{verbatim}

\textit{Figure 115 - CompareAndSet from LL/SC}
In Figure 115 we use \texttt{ll(v)} to represent the use of an LL instruction and \texttt{sc(v, newvalue)} to represent the use of an SC instruction.

The ARM specification of \texttt{ldaxr/strlxr}, which is used to implement \texttt{CompareAndSet} functionality, states that not more than 128 instructions may intervene between the two parts of the pair. We have been unable to ascertain the reason for this limitation, but we speculate that this allows for implementations that impose an instruction count limit on the period for which a flag setting is recognised. This would regularise the case shown in Figure 115 where the code may exit without having executed an SC instruction to match the LL instruction.

In the ARM architecture, the \texttt{ldaxr/strlxr} combination, has a weak memory implementation. The combination is guaranteed to be atomic for the accessed memory location, but re-ordering of other Load and Store operations across the combination is permitted. This corresponds well with the definition of \texttt{weakCompareAndSet} as provided in the Java documentation (Gosling, Joy et al. 2014).

Doko and Vafeiades (2016) note that their method for reasoning over weak memory confirms the intuitive result that overall performance is improved if fences are removed from the body of spinlock loops.

\begin{verbatim}
AtomicInteger ai;
...  while (!ai.compareAndSet(FREE, TAKEN)) {Thread.yield();}
...

Figure 116 - Conventional CompareAndSet spinlock
\end{verbatim}

For example, the conventional CompareAndSet spinlock shown in Figure 116 is better replaced by the code shown in Figure 117.

\begin{verbatim}
AtomicInteger ai;
...  while (!ai.weakCompareAndSet(FREE, TAKEN)){Thread.yield();}
  VarHandle.fullFence();
...

Figure 117 - Improved weakCompareAndSet spinlock
\end{verbatim}

However, if this "weak memory friendly" code is executed on a JVM that is targeted to the x86 architecture there is a danger that the implicit full fence effect of the \texttt{lock:xchg} instruction will be followed immediately by another
explicit \textit{fullFence}, thus incurring a second and unnecessary set of cache coherency costs.

Although there is an obvious benefit in having the logic of a JVM common across multiple architectures, it is equally clear that efficiency demands that the situation described here requires different pieces of logic for different architectures and even different implementations of the same architecture. The decision table shown in Figure 118 provides a succinct description of the required logic.

<table>
<thead>
<tr>
<th></th>
<th>\textbf{compareAndSet(CAS)}</th>
<th>\textbf{weakCompareAndSet(wCAS)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{x86}</td>
<td>\textit{Use lock:xchg.}</td>
<td>\textit{Use lock:xchg. If there is a fullFence before the next Load or Store, then it can be optimised away.}</td>
</tr>
<tr>
<td>ARM</td>
<td>\textit{Use ldaxr/strlxr combination to implement logic similar to that shown in Figure 115, followed by dmb.}</td>
<td>\textit{Use ldaxr/strlxr combination to implement logic similar to that shown in Figure 115, but omit the succeeding dmb.}</td>
</tr>
</tbody>
</table>

\textbf{Figure 118 - Decision table for CAS/wCAS v architecture}

\textbf{VI.1.2. Our proposed extension to VarHandle methods}

The ARM architecture offers the possibility of a more efficient implementation of the logic of Figure 117. We show a possible example of appropriate pseudo-code in Figure 119 using the same conventions as those of Figure 115.

\begin{verbatim}
r = ll(v);
while (r != expected) {
    Thread.yield();
    r = ll(v);
}
return sc(v, newvalue);
\end{verbatim}

\textbf{Figure 119 - ARM optimisation of spinlock}

To enable this possibility directly within the bytecode, it would be necessary to define additional VarHandle methods. Following the complex instruction set style, we might propose, as an extension to the current set of methods, a signature-polymorphic equivalent of the method in Figure 120.
```java
void spinLock(int v, final int expected,
         final int newvalue, Consumer(int) lambda);
```

Figure 120 - Possible definition for `VarHandle.spinlock`

This would support code similar to that shown in Figure 121.

```java
class MyLock implements Lock{
    enum State {FREE, TAKEN};
    int v = 0;
    ...
    void lock() {
        VarHandle.spinlock(v, State.FREE,
                           State.TAKEN,
                           {Thread.yield();});
    }
}
```

Figure 121 - Example use of spinlock

The alternative proposal, which conforms better to the reduced instruction set style, would be to add signature-polymorphic equivalents of the following methods:

```java
int ll(int v);
boolean sc(int v, int newvalue);
```

In this case, the code equivalent to that shown in Figure 121 would be that shown in Figure 122.

```java
class MyLock implements Lock{
    enum State {FREE, TAKEN, NULL};
    int v = 0;
    ...
    void lock() {
        boolean success = false;
        int r = NULL;
        while (!success) {
            r = VarHandle.ll(v);
            while (r != FREE) {
                Thread.yield();
                r = VarHandle.ll(v);
            }
            success = VarHandle.sc(v, TAKEN);
            if (!success) {Thread.yield();}
        }
    }
}
```

Figure 122 - Example of use of RISC-style spinlock
In both examples, the immediate `Thread.yield()` might be replaced by a more sophisticated exponential back-off strategy, though the principles of operation would remain the same.

Based on the experimental evidence gathered in Chapter III, we suggest that the benefits of such strategies must be weighed carefully against the benefits of reducing the code paths within `lock()` methods to a minimum. It is a moot point which of these styles might provide the least-cost solution and further research beyond the scope of our present work would be needed to measure the performance of competitive implementations. What is certain is that without some proposal of this nature, a weak-memory implementation of the JVM must sacrifice some potential gains in efficiency.

We follow this discussion of the implementation of `CompareAndSet` with a discussion of the effects of applying the advice given in the JSR 133 Cookbook (Lea 2008) for the placement of fences around accesses to a `volatile` variable.

**VI.1.3. Implementation of volatile**

In section V.2.2 of Chapter V we noted that the conservative recipe for the placement of memory fences around accesses to a `volatile` variable is perfectly satisfactory for the x86 architecture, but has the potential for significant inefficiency when applied to an architecture, such as ARM, that generally requires the non-null implementation of more memory fences.

An Abstract Event Graph shows the program order links between events and omits the sequence of other types of instruction that may also form part of the program order between the events. This tends to obscure the fact that if a memory fence is to be inserted into an AEG it may be inserted after the first event or before the second event and that these are two distinct locations within the program order.

The JSR 133 Cookbook (Lea 2008) includes the analysis shown in Figure 123 that shows how edges containing more than one memory fence can be reduced to a single fence. However, it adds the comment that the analysis needed to supply correctly the conditions for the operations column
"in the presence of loops, calls, and branches is left as an exercise for the reader. :-)"

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st</th>
<th>operations</th>
<th>2nd</th>
<th>1st</th>
<th>operations</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadLoad</td>
<td>no loads</td>
<td>LoadLoad</td>
<td>no loads</td>
<td>LoadLoad</td>
<td></td>
</tr>
<tr>
<td>LoadLoad</td>
<td>no loads</td>
<td>StoreLoad</td>
<td>no loads</td>
<td>StoreLoad</td>
<td></td>
</tr>
<tr>
<td>StoreStore</td>
<td>no stores</td>
<td>StoreStore</td>
<td>no stores</td>
<td>StoreStore</td>
<td></td>
</tr>
<tr>
<td>StoreStore</td>
<td>no stores</td>
<td>StoreLoad</td>
<td>no loads</td>
<td>StoreLoad</td>
<td></td>
</tr>
<tr>
<td>StoreLoad</td>
<td>no loads</td>
<td>LoadLoad</td>
<td>StoreLoad</td>
<td>no loads</td>
<td></td>
</tr>
<tr>
<td>StoreLoad</td>
<td>no loads</td>
<td>StoreLoad</td>
<td></td>
<td>StoreLoad</td>
<td></td>
</tr>
<tr>
<td>StoreLoad</td>
<td>no volatile loads</td>
<td>StoreLoad</td>
<td>no volatile loads</td>
<td>StoreLoad</td>
<td></td>
</tr>
</tbody>
</table>

Figure 123 - Eliminating redundant memory barriers (Lea 2008)

On an x86 architecture where all memory barriers except StoreLoad are no-ops, the recipe described in Figure 104 in section V.2.2 of Chapter V is perfectly acceptable. However, on an architecture, such as ARM, that recipe, when coupled with the omission of the optimisations shown in Figure 123, would result in a proliferation of instances of the only available fence, dmb. Sadly, as McKenney (2010) remarks, the only variant of the dmb instruction that matches the required semantics is the global scope variant, which is a full fence. Clearly, something has to be done to avoid these costs where they are unnecessary.

VI.1.4. Implementing fences

In Chapter V, we discussed our algorithm for predicting an optimal low-cost selection and placement of memory fences to restore sequential consistency to a multi-threaded program that is executing in a weak-memory environment. We noted that the Java bytecode interface, as extended by the VarHandle fence methods introduced at Java 9, allows us to perform a static analysis of the bytecode and modify it so that the corresponding generic fence instructions are generated at the appropriate points in bytecode. The JSR 133 CookBook (Lea 2008) notes that, in the ARM environment and in a number of similar architectures, Load-Load and Load-Store fences may be replaced by artificially constructed address dependencies with the implication that this might improve performance.
The restoration of sequential consistency is achieved by preventing the hardware from re-ordering instructions. When this is done the program incurs two types of cost:

- Cache coherence;
- Execution cost of instructions.

It is not difficult to demonstrate the relative uniprocessor costs of executing particular instructions that restore sequential consistency to a sequence of read operations. Experimentally, this shows that relative costs of using memory fences or other techniques are different on different implementations of the same architecture. In some cases, it is worth replacing \textit{dmb} instructions by address dependencies and in other cases it is detrimental. The measurements for two different ARM implementations are shown in Figure 124. This set of results shows the relative performance of a set of read instructions separated by XOR, DMB and LDAR implementations of fences. For commercial reasons, it is not possible to identify the processors used in this experiment.

Processor 1:
- XOR: 0m7.374s
- DMB: 0m4.210s
- LDAR: 0m2.635s

Processor 2:
- XOR: 0m3.961s
- DMB: 0m7.503s
- LDAR: 0m10.004s

\textit{Figure 124 - Relative performance of address dependency code (Haley 2017)}

The crucial fact is that the cost of cache coherence is not dependent on the way in which the coherence actions are caused. This is apparent from the explanation of the behaviour of weak memory processors provided by McKenney (2010). Clearly, this cost will show large stochastic variations depending on the way the concurrent threads interact, the degree of contention, etc. However, our measurements in the x86 environment, which were described in section III.1.1 of Chapter III, suggest that cache consistency costs are one, and possibly two, orders of magnitude greater than the uniprocessor cost of executing a read or write instruction. As shown in
Figure 124, the ratio of uniprocessor costs for different techniques is much smaller, of the order of 1.5 to 2.

If $P$ is the cost to the program of the fence operations, $u$ is the uniprocessor cost of the instructions and $C$ is the cost of ensuring cache coherence then

$$P = u + C$$  \hfill (117)

$$C \cong 10u$$  \hfill (118)

$$u \cong P/11$$  \hfill (119)

$$\partial u < u$$  \hfill (120)

where $\partial u$ is the difference in cost between different techniques for forcing re-ordering. So that, at best, using the optimal technique would save $\cong 10\%$ of the overall cost, $P$. If the cost of coherence is two orders of magnitude more expensive than the average instruction, then the potential saving shrinks to 1%.

Figure 124 also shows that this optimisation would have to be made specific to the particular implementation of the target architecture. It would not be effective to make it dependent just on the architecture.

**VI.1.5. Summary of current implementations**

Java Enhancement Proposal 193 (Lea and Sandoz 2015) that specifies the VarHandle features implies that the classes in the `atomic` package ought to be upgraded so that the use of the UnSafe class is replaced by appropriate use of the VarHandle features. It is not clear whether this change will be introduced with the release of Java 9. At the time of writing this thesis, the available release of Java 9-ea did not include this change and there is no indication when the UnSafe class will be declared obsolete.

Although there appears to be an active commercial project to port the JVM to the ARM architecture, it is unclear whether this is targeted at Java 9 or a later release and whether the potential for inefficiency that we identify will be addressed in that release. The JIKES RVM (Alpern, Augart et al. 2005) and the Maxine VM (Wimmer, Haupt et al. 2013) research projects appear to be targeting the ARM architecture, but it is not clear whether the changes discussed here are receiving attention within those projects and whether the
necessary co-ordination with the JDK class libraries has been organised. Our research, presented here, shows that the goal of a single-source, portable JVM written in Java must be tempered by the recognition that efficiency demands that the executed code must be specifically tailored to match the features of the target architecture and, probably, the host implementation of that architecture. Our proposals, detailed in section VI.1.2 above are part of the solution to this problem.

In preparation for our discussion of the other necessary changes, we briefly recall the JVM components which were introduced in Chapter II, section II.1.5.

**VI.2. JVM components**

The JVM includes three execution mechanisms:

- Interpreter;
- C1 compiler;
- C2 compiler.

The C2 compiler uses the profile information generated by the C1 compiled code to identify the "normal" execution path. It then builds a machine code implementation of this "normal" path as a linear sequence of instructions. All branches are re-structured so that the unsuccessful branch that "falls through" into the next instruction in sequence is given to the "normal" path. Abnormal paths are treated as exceptional and handled by branching to code that often simply returns control to the Interpreter. The compiler also un-wraps loops and in-lines methods. This technique provides the optimal conditions for the efficient execution of code on heavily cached and pipe-lined CPU architectures. It also satisfies the conditions that the JSR 133 Cookbook (Lea 2008) suggested would be too hard to implement.

The C2 compiler is implemented in C/C++. One of the objectives of the Graal project (http://openjdk.java.net/projects/graal/) is the development of a compiler, written in standard Java, which can support all the features of the current C2 compiler and extend them to provide other optimisations. Duboscq, Stadler et al.(2013) argue that the complexities of optimising compiled code are better handled in the Java language.
Java 9 formally supports the prototype use of the Java Virtual Machine Compiler Interface (JVMCI) (Rose 2016). This defined API provides the necessary communication channel between the JVM and an external compiler. It is not restricted to the support of the Graal compiler, though, of course, the Graal compiler relies on its features.

In the next section we describe how the Graal compiler uses the JVMCI to interact with the JVM.

**VI.2.1. Java Virtual Machine Compiler Interface (JVMCI)**

The JVMCI is the formal interface through which external compilers may cooperate with the Java Virtual Machine. It is defined in Java Enhancement Proposal (JEP) 243 (Rose 2016). In Figure 125 we show a schematic of the JVMCI.

![JVMCI schematic](image)

The JVMCI is in three parts:

- An API that provides the bytecode of a method to an external compiler and allows the external compiler to call on services from the JVM;
- An API that gives the compiler access to the JVM’s library of low-level Java fragments called *snippets*;
- An API that allows the compiler to deliver compiled machine code sequences to the JVM for execution.
We provide this information as background on the Graal compiler. Within the classes that comprise the compiler, the detail of the handling of the JVMCI is encapsulated in particular classes so that, to some extent, it is possible to add extra features to the compiler without knowledge of this detail.

VI.2.2. Graal compiler - Internal mechanisms

The most useful feature of the Graal compiler is the way in which it represents the code of methods to be executed. The Internal Representation (IR), referred to in Figure 125, is a graph with nodes and edges. The nodes correspond to instructions from the bytecode or higher-level concepts such as method invocations and loops. Every node has a single predecessor in the set of edges that forms the program order. There are other sets of edges that represent associations between nodes. For example, there is an association between a Load node and the Address node that represents the resolved address of the memory location accessed by the Load node. There is an internal graph-handling package that maintains the integrity of the links that implement the edges. Every edge is maintained as a bi-directional link. All this is encapsulated within the package of classes and accessed through a defined API of methods. The graph is built and maintained on Static-Single-Assignment (SSA) principles (Cytron, Ferrante et al. 1991) and is, thus, representative of contemporary compiler technology.

The compiler performs its optimisations and other processes by following a common pattern. The work done by the compiler is divided into phases. The phases are invoked successively by a control method. Each phase has an associated named Java interface. If a node is to participate in a phase then it must implement that named interface. For example, if a node is to participate in the Canonicalization phase then it must implement the Canonicalizable interface. Each of these interfaces specifies a method that must be implemented. For example, the Canonicalizable interface demands a canonical method that implements all the actions that must be performed on the node in that phase. The phase proceeds by processing all the nodes in the graph in turn. If the node implements the interface associated with the
phase then the node must support the required method, which is then invoked. If the node does not implement the interface, it is ignored.

The graph is successively transformed by the phases to effect the various desired optimisations. Finally, a phase specific to the target architecture is invoked to transform the low-level internal representation graph into an ordered set of machine code instructions.

**VI.3. Modifying the behaviour of the compiler**

If we wish to change the way the compiler implements features, such as the machine code implementation of fences, we must modify the source code of the compiler and generate new executable class files. Presently, it is not possible to change the behaviour of the compiler dynamically.

As explained in (Duboscq, Stadler et al. 2013), it is expected that development of new optimisation features will proceed as follows:

- Inspect the IR graphs that show the mutation of the graph throughout the existing compilation process as applied to suitable piece of test code that exercises the Java feature to be optimised;
- Determine the desired optimised state of the graph prior to the generation of machine code;
- Identify the existing phases in which the desired changes might be introduced, noting the constraints on the effects that may be achieved in particular phases;
- Mechanise the changes by modifying existing nodes and defining new nodes that implement the interfaces needed to organise their participation in the selected phases.

The internal structure of the compiler imposes limitations on the design of proposed extensions. Although the nodes in the graph are linked by edges, the execution of a phase ignores them. The list of all nodes in the graph has an order that simply reflects the order of creation of the nodes. The nodes are processed in that order. Replacing a node is achieved by removing the current node and adding a new node to the list. This changes the position within the list of the node's replacement with respect to other nodes. This explains the rule that the code in the phase-specific method of a node must
not change other nodes. There is a further rule that the code in a node cannot rely on the state of another node that participates in the same phase. In effect, the code in a node must assume that it is being processed in parallel with the code in all other nodes that participate in the phase.

We can illustrate the effect of these rules by means of an example. Suppose that a graph contains nodes A and B, and that we wish to transform these nodes into nodes P and Q respectively. Further suppose that we wish to create a specific edge between P and Q. To comply with the rules, our desired changes must be distributed across the nodes and split across two phases. If we assume that P is essentially the same as A, but includes a "Qlink" attribute, then the change can be effected by distributing the following actions across the phase-specific methods of the various nodes.

Phase 1
- Node A: replace this node with a node P;
- Node B: replace this node with a node Q;

Phase 2
- Node P: search for node Q store its identity in the "Qlink".

To facilitate this sort of design, the Canonicalization phase keeps a list of the nodes created during the first execution stage of the phase, it then proceeds to a second stage of execution that processes only those nodes created in the previous stage. This process is repeated until there are no remaining new nodes to be processed.

Using this optimisation, we might code our example as follows:

```java
Canonicalization
Node A implements Canonicalizable
    method canonical : Replace node by node P
Node B implements Canonicalizable
    method canonical : Replace node by node Q
Node P implements Canonicalizable
    method canonical : Search for node Q and update Qlink field
```

In the first stage, the methods for A and B are executed (though the order of this is not determinate). The method of P is not executed during this stage, but P and Q are added to the list of created nodes. In the second stage, the
method of P is executed and the link created by storing the identity of Q in the Qlink field, which is declared to be an edge. The node Q, which is not Canonicalizable does not participate in the phase.

In the following sub-sections we describe how these general techniques are used to achieve particular goals.

VI.3.1. Our design for eliminating redundant fences

In this sub-section, we describe how we might implement a process to recognise an existing address dependency and eliminate a redundant memory fence.

This involves injecting code into the execution of the phases of the Graal compiler. We represent this schematically as shown in Figure 126.

![Figure 126 - Optimisation schematic](optimisation.png)

Suppose that our example code fragment has addresses that lie within the same object. We illustrate this with the code shown in Figure 127.

```java
x = c.v;
VarHandle.acquireFence();
y = c.a;
c.b = 42;
```

![Figure 127 - Fragment with address dependency](dependency.png)

The corresponding initial IR graph is shown in Figure 128. Here all the Load nodes that are guarded by the MemBar node have the same root address dependency, Address c.
To ascertain that this is the case, the MemBar node must be modified so that it performs the following tasks:

1. Search backward from the MemBar node along the predecessor chain to find the immediately preceding Load node.
2. If not found, do nothing and exit the process.
3. If found, follow the links to Address c and retain its identity.
4. Search forward from the MemBar node for succeeding Load nodes. Finding another MemBar node or the end of the graph terminates the search.
5. If not found, do nothing and exit the process.
6. If found, follow the links and if they lead to Address c, validate that it is the same address as that found in step 3.
7. If all the Load nodes processed in step 6 lead to the same node found in step 3, then replace the MemBar node by a node that will generate a no-op instruction.

We have successfully implemented a prototype of this process by modifying the MemBar node to participate in the Canonicalisation phase. We have observed that the IR graph is amended as expected and that the subsequently generated code does not contain a fence.

This prototype generates x86 code. In this target environment, a MemBar acq node would naturally generate no fence instructions. Accordingly, we artificially validated our process by changing the MemBar node to generate a full fence. We observed that the generated code did, indeed, contain a full fence instruction.
This prototype indicates that modifications to the generated code to reflect alternative implementations of fences are technically feasible.

Although we have not pursued the idea through to implementation, we believe that this technique could also be applied to the elimination of redundant fences according to the logic set out in Figure 123. Further examination of the Graal source code has revealed that a deep investigation of its implementation would be needed to achieve this more extensive application of the principle. Such an investigation lies beyond the scope of our current research.

VI.3.2. Our design for replacing a fence with an address dependency

As noted in section VI.1.4 above, it is not certain whether this substitution would necessarily provide a performance benefit. The evidence presented in that section suggests that it might be necessary to benchmark a variety of different machine code sequences to find the most effective solution for each individual implementation of a target architecture. We present here our design for implementing an address dependency as an example of the technique.

We take as an example the code fragment shown in Figure 129, which is a simplification of the start of a critical section.

```
1: x = v;
2: VarHandle.acquireFence();
3: y = a;
4: b = 42;
```

*Figure 129 - Fragment with acquireFence*

The variable v is a shared flag variable that controls the access to the critical section. For clarity, we have omitted the SpinLock that would regulate this access. The variables x and y are local to the method. Variables a and b are shared variables that are guarded by v. Figure 130 shows the corresponding initial IR graph that would be used by the Graal compiler.
Instruction 1: maps into the first LoadField node that has a link to a separate Address node that holds the address. Instruction 2: is recognised as a compiler intrinsic and mapped directly into a MemBar node which has an acquireFence value for its barrier attribute. The remaining instructions are similarly mapped into Load and Store nodes with attached Address nodes. The order of the instructions is reflected in the program order edges.

Suppose that we wish to replace the acquireFence with artificial address dependencies. The intention is illustrated by the pseudo-code shown in Figure 131, where \( d \) is a local variable.

\[
\begin{align*}
x &= v \\
d &= v \text{xor} v \\
y &= a(d) \\
b(d) &= 42
\end{align*}
\]

This address arithmetic is intended to leave the eventual address unchanged, but dependent on \( v \).

During the course of the Lowering phases of the Graal compilation, the initial IR graph becomes transformed into different nodes for the purposes of different types of optimisation. For example, Load and Store nodes are fixed in the program order and the Scheduling phase respects that order. Conversely, Read and Write nodes are floating and are scheduled for execution according to their address dependencies. This means that they may be scheduled in an order that is radically different from their program order. We illustrate this transformation with the corresponding IR graph shown in Figure 132.
To achieve our desired result we need to transform this graph into something similar to Figure 133. The addresses have been made artificially dependent on the address of the flag variable and a node that will not generate any instructions replaces the original memory fence.

It is intuitively apparent that this design is consistent with the general principles of operation of the Graal compiler. However, a deep investigation of its implementation lies beyond the scope of our research.

In this section we have shown how the features of the Graal compiler can be used to deliver an efficient implementation of fence operations for weak memory architectures. In the next section, we discuss the feasibility of hosting the algorithms described in Chapter IV and Chapter V within the environment provided by the Graal compiler, and the benefits that this might provide.

VI.4. Hosting our algorithms within Graal

By inspection of the Graal source code we have determined that the nodes and edges of the IR graph hold all the information needed to construct the same list of Tokens that we extract from the bytecode as described in section IV.4 of Chapter IV.
Accordingly, we can provide an interface between the code that we have developed for static analysis and the internals of the Graal compiler by implementing an adapter or adapters as shown schematically in Figure 134.

This establishes the feasibility of integrating our code within the Graal environment. In the following sub-section, we consider the benefits and limitations of hosting our data race detection algorithm within the compiler.

**VI.4.1. Hosting data race detection**

The algorithm developed in Chapter IV assumes that the program intends to respect the implicit access protocol associated with the acquire/release paradigm. It cannot and does not distinguish between a program of this type with errors and a valid program that uses a different paradigm to achieve freedom from data races. The algorithm relies on its ability to identify critical section de-limiters and, thence, to separate a stream of events into those that are guarded within critical sections and those that are not.

The Graal compiler aggressively in-lines invoked methods and re-writes IF statements to deliver a "straight-through" sequence of instructions. This list of instructions expects that the execution will proceed directly through the list with no "out-of-line" execution. For our algorithm, this means that the process of recognising de-limiter patterns cannot be directly applied to the Tokens derived from the Internal Representation (IR) graph and must be adapted to recognise Tokens that are generated when the de-limiter patterns have been lowered by the compiler. The principal effect here is that the invocations of the Lock interface, which we explicitly
recognise, are no longer apparent in the lowered IR graph. The de-limiters must be recognised by parsing the primitive actions for patterns. This technique is already part of our algorithm, but handling the Tokens derived from an IR graph will require development of the grammar used by the parser.

The conditions that must be satisfied for an effective static search for data races, as described in Chapter IV, are summarised here.

C1. the instructions themselves;
C2. the control flow graph (CFG) should be statically resolved;
C3. the functions called should be statically resolved;
C4. the threads running should be statically determined;
C5. all references should be resolved.

We now compare these characteristics with those of a list of Tokens derived from the Internal Representation (IR) graph used by the Graal compiler.

The instructions
In a static search, we perform a comprehensive examination of all the sequences of instructions that may be executed. Within the Graal compiler environment at run-time, the algorithm is examining the list of instructions that is about to be executed. The only dubious element is the assumption that the code is multi-threaded against itself, which we discuss further in the subsection on threading.

Control-flow-graph statically resolved
The compiler goes further than a static resolution of the control-flow-graph (CFG). The branch instructions are transformed so that, in effect, they no longer exist. What remains to be analysed is a simple single list of events in the order of their execution. This removes all the approximations from our algorithm.

Functions (methods) statically resolved
Because the Graal compiler aggressively in-lines invoked methods without restriction on the number of instructions in-lined or the depth of the
invocation chain, the Token list derived from its IR graph does not require the complication of the Invocation Hierarchy Explorer (IHE) described in section IV.5.1 of Chapter IV.

**Threads statically determined**

In general, the compiler may assume that the code represented by its IR graph will be executed concurrently in multiple threads. However, the *Context* supplied by the JVM across the JVMCI includes sufficient information for the compiler to deduce that the code is being executed in a single process and tailor its optimisations accordingly. We have not researched the possibility that the JVMCI could be extended to support requests from the JVM for the compiler to investigate the interactions between two threads that are executing different but related pieces of bytecode, or for the compiler to request the information needed to support such an investigation. This deep knowledge of the detail of the JVMCI lies beyond the scope of our research. In the absence of better information, our algorithm assumes that the code corresponding to the method being processed will be executed concurrently in multiple threads. This is less comprehensive than the static analysis situation where the algorithm correctly caters for the situation where different threads may execute different, if complementary methods. On the other hand, it is representative of the common implementation where both Consumers and Producers use the same critical section logic.

Without an examination of the actual code being executed in other threads, the analysis ceases to be sound, though it may be useful in many practical circumstances.

**References resolved**

By the time that the initial IR has been transformed into the Low-level Internal Representation (LIR), the compiler has used the metadata interfaces of the JVMCI to resolve all addresses into memory locations. This resolves all addresses absolutely so that there can be no question of aliasing. It also eliminates the need for the approximation that blurs the distinction between static and class instance variables. This makes the analysis intrinsically more sound and more complete.
Enforced block-structuring of critical sections

In section IV.2.5 of Chapter IV we presented an argument that a program design that does not align critical sections with the lexical block-structure was poor practice. We used this argument to justify our implementation decisions to provide no support for critical sections: that did not conform to the lexical block structure; and that crossed inter-method boundaries. This decision was crucial to our achievement of adequate scalability. However, this prevents our prototype from handling a whole class of legitimate programs.

The list of Tokens derived from the IR graph has no internal block structuring. Accordingly, our parsing for de-limiter patterns will be effective irrespective of the alignment of critical sections with lexical blocks or with method boundaries. This increases the applicability of our algorithm to include a greater proportion of arbitrarily selected real programs.

Data race detection - suitability for run-time use

Based on the performance measurements reported in Chapter IV section IV.7, we estimate that the cost of performing data race detection using our algorithm is about 350millisecs per KLOC. The performance reported by Thompson, Farley et al. (2011) suggests that a high-performance in-memory transaction system uses about 1000 LOC per transaction. From which we conclude that using our algorithm at runtime would incur a one-time cost of about 350millisecs for a typical piece of transactional code. This is commensurate with the other costs of JIT compilation.

The detection of a data race condition by our algorithm does not provide a guarantee that an actual data race is about to occur. The actual occurrence of a data race requires that the relevant interaction is actually realised by the stochastic scheduling of the competing threads. Even where a data race condition exists, its actual occurrence is rare. The previous research in this area was introduced in section II.1.2 of Chapter II. Accordingly, we suggest that the run-time detection of a data race condition by our algorithm should be reported or logged, but should not cause a run-
time Exception. It is a moot point whether the generated code containing the data race condition should be allowed to run.

Without an extension to the JVMCI to allow the algorithm to view the actual code about to be executed across multiple threads, the analysis is not sound. However, we suggest that even when restricted to the assumption that the IR graph will execute against itself in multiple threads, the use of our algorithm at run-time represents a useful check against data races for a commonly used class of programs.

In the next sub-section, we consider the benefits and limitations of hosting our algorithm for the restoration of sequential consistency within the Graal compiler.

**VI.4.2. Restoring sequential consistency**

In sections VI.1 and VI.3 we discussed the changes to the Java Virtual Machine that are needed to provide efficient support for memory fences on weak-memory architectures and showed how these can be implemented within the existing Graal compiler. As shown in Figure 134, it is technically feasible to integrate code from our algorithms within the Graal environment. In this section we consider what benefit, if any, may be derived from such a development.

In Chapter V we conclude that the algorithm for selecting and placing memory fences does not scale sufficiently well for development into a practical tool that could handle industrial-sized programs. Conversely, we suggest that it would be useful for determining the minimum configuration of fences needed to ensure the sequential consistency of novel de-limiter patterns. This provides a partial solution couched in terms of the placement of VarHandle fence methods. The other part of this solution is the optimal implementation of those methods as discussed in section VI.1.

As we explain in Chapter V, the full analysis of Abstract Event Graph cycles is valuable only where it is applied across multiple threads that are executing different code sequences. In simpler cases, the logic applied in section VI.1 is sufficient. Presently, the environment within the Graal compiler presents only the optimised execution path for a single method.
Although it satisfies the criteria for the resolution of methods and the alias resolution of variables, the lack of access to the corresponding execution paths of methods that might be executed concurrently by other threads means that there is nothing to be gained from executing what is a computer-intensive algorithm at runtime.

VI.5. Conclusions

We have shown how code introduced into the execution of the Graal compiler can complete the provision of efficient sequential consistency for novel de-limiter patterns analysed by the static algorithm described in Chapter V. We provide a proof-of-concept that demonstrates the technical feasibility of this approach.

We have shown how our static analysis algorithm for the detection of data races can be simplified and integrated within the execution of the Graal compiler. The compiler's optimised sequence of machine code instructions represents only the immediate execution path. Accordingly, when integrated within the compiler, the algorithm lacks the ability to consider data races between this code and other different code executed in concurrent threads. This is less sound than the static analysis. On the other hand, the optimised execution path has in-lined all invoked methods, resolved all addresses and resolved all block structuring. This eliminates the need for the palliative approximations that impair the generality of the static algorithm. We conclude that it would be worthwhile to pursue this integration into a full-scale prototype.

The static analysis for the sequential consistency of de-limiter patterns can only progress as far as an optimal sequence of VarHandle methods. The full solution of the problem requires the enhancements to the Java Virtual Machine set out in this chapter. However, we can see no advantage in integrating that static analysis within the JVM.
Chapter VII Avoiding data races by construction

"Merely corroborative detail intended to lend attractive verisimilitude to a bald and otherwise unconvincing narrative"

"The Mikado"

W. S. Gilbert

In Chapter III, we established that, although different synchronisation techniques have different overheads, synchronisation always incurs a significant cost. In that chapter, we argued that efficiency is best served by keeping critical sections short and simple. In Chapter IV we showed how data race errors are caused by the failure to respect the implicit protocol associated with the acquire/release paradigm and provided a static data race detection algorithm to mitigate this problem. In Chapter V, we provided an algorithm to ensure that the de-limiters of critical sections use optimally placed memory fences and in Chapter VI we discussed the optimisation of fence operations to take the best advantage of the features of different target machine architectures.

We argued the case for a minimal use of synchronisation in Chapter III. The logical extension of this argument is the development of a mechanism for sharing of data between concurrently executing threads that enforces the necessary control and makes minimal use of expensive synchronisation. In this chapter, we describe the design and construction of a novel DataStore class that uses the features of the Java language to enforce a lock-free protocol and takes full advantage of the results of our research into efficient ways of ensuring sequential consistency, which we described in Chapter V and Chapter VI.

Our objective in undertaking this work was to demonstrate the practicality of building such a class and to evaluate its performance. We recognised that there would be a gap between such a prototype implementation and a worthy production product. We begin this section by
providing the practical motivation for such a class and then describe its
design and specification.

VII.1. Motivation
Suppose there is a program, designed according to agent-oriented design
principles, that comprises a set of independent concurrently executing
agents that communicate through a shared body of knowledge. Historically,
this might have been implemented as a set of Enterprise Java Beans
communicating through SQL queries dispatched to a common relational
database server. As we discussed in Chapter III, this approach incurs
substantial overheads with severely detrimental consequences. It effectively
precludes close and frequent agent interaction. Accordingly, we consider
here an approach that implements agents as threads and seek a
correspondingly low-overhead approach to the implementation of the
shared data store.

VII.2. Overview
This section describes our proposal for a DataStore class. The design of the
class is based on the following set of goals:

- There is a segregation between readers and writers of the data held
  by the DataStore;
- The segregation is enforced by the rules of the Java language;
- Readers are prevented from writing to the data held by the DataStore
class through the use of the features of the Java language;
- The mechanism through which write access is permitted is
  encapsulated within the methods of the DataStore class and its
  associated classes. It cannot be circumvented by writers.

We achieve these goals as follows. The DataStore class holds an object that is
private to the class. The methods of the DataStore class regulate access to
the stored object. Our objective is to provide lock-free read access for many
threads. Changes to the structure of the stored object or changes to the value
of its fields must not cause inconsistent results for readers. There must be no
data races between multiple writers or between a writer and readers. We
note that these requirements have great similarity with those described for
thread-safe objects by Daloze, Marr et al. (2016), which we discuss in section VII.4. The overall performance must be competitive with that provided by a conventional \textit{synchronized} implementation. Our original goal was that the performance should not be "markedly inferior" because we assumed that the overheads of CopyOnWrite would predominate. As we report in section VII.10.2, the performance of the DataStore class is superior over a wide range of operating conditions.

We begin by specifying with greater precision the required characteristics of the DataStore class and the constraints placed on the object that it stores. We follow this with a detailed description of the way in which the features of the Java language are used to ensure that these requirements are met. By analysing the potential error conditions, we reveal what additional work is needed to make the correct detection and handling of these conditions complete. We show that, subject to the correct handling of these error conditions, our design guarantees freedom from data races.

We report our empirical evidence that the DataStore class provides good performance over a wide range of operating conditions and conclude that its performance is competitive with a more conventional implementation while offering a superior freedom from data races.

\textbf{VII.3. Specification}

Here we use the term \textit{thread} to imply a concurrently executing and, therefore, potentially interfering thread.

The DataStore class shall:

1. provide complete integrity and ensure the absence of data races. This implies that the reference to the stored object itself is \textbf{private} to the DataStore class and is externally accessed only through accessor methods;

2. hold an instance of a class that has no limits on its internal complication. That class may be the head of a hierarchy of class instances of arbitrary breadth and depth;

3. support unfettered read access for an arbitrary number of threads. This implies that, if appropriate, all the attributes of all the classes
within the stored data hierarchy can be made **public**. For this implementation, accessor methods are deemed too costly. This implies that all these attributes must be made immutable;

4. ensure that mutation operations of the data structure occur with minimal effect on read accesses;

5. ensure that changes to the values of any attribute of any class instance within the stored class hierarchy occur with minimal effect on read accesses;

6. ensure that conflicts between concurrent attempts to mutate the data structure or change the value of an attribute are resolved with minimal effect on read accesses.

7. ensure that, with the exception of Durability, attempts to mutate the data shall satisfy the ACID (Atomicity, Consistency, Isolation, Durability) properties (Haerder and Reuter 1983) required of database transactions.

8. use the available features of the Java Language Specification effectively to enforce these requirements. It is not sufficient to rely on the threads to respect an access protocol.

It is not intended that DataStore instances should be persisted or serialised for transmission between Java Virtual Machine instances. Accordingly, the class does not need to implement the Serializable interface.

**VII.4. Related work**

Daloze, Marr et al. (2016) describe their work in providing thread-safe objects for dynamically typed languages such as JRuby (Nutter, Enebo et al. 2011). This work was conducted in the Truffle (Wurthinger, Woess et al. 2013) environment. Although Daloze, Marr et al. provide similar arguments for using lock-free read access and concentrating the overheads within the support for write actions, their work is particularly concerned with supporting languages such as JRuby with the Truffle environment. Conversely, our objective was to provide a thread-safe object for Java using only the existing features of the Java language. Accordingly, we have found
that although we share the same philosophical approach, we have been unable to re-use any parts of their implementation.

**VII.5. Meeting the specification**

In the overview section we provide a high-level description of the concurrency provisions within our DataStore class.

**VII.5.1. Overview**

Processes that wish to read the data held in the DataStore retrieve a reference to the stored data object through an accessor method. The stored data object is a hierarchy of immutable objects so that many threads can perform read accesses to that object without locks or fences, secure in the knowledge that write actions are impossible.

The management of concurrency is concentrated within the accessor method that applies changes to the stored data object. Our algorithm allows the parallel execution of this accessor method by multiple threads. To maintain integrity, only one of these threads is allowed to succeed. The other thread or threads fail. They must re-consider and re-present their proposed changes to take account of the new values held in the stored data object. The only synchronisation point is an AtomicCompareAndSet method that replaces the reference to the old immutable copy with a reference to a new stored data object that is an immutable copy of the data object to which the desired changes have been applied. Each of the concurrent updating threads takes its own mutable copy of the current stored data object, retaining a local copy of the reference to the immutable object. It then applies changes to the mutable copy and creates an immutable copy of this changed object. Finally, it tries to replace the reference to the old immutable object by a reference to its new immutable object. The AtomicCompareAndSet method fails if the current reference does not match the local copy retained by the updating thread. This algorithm deliberately and aggressively uses the CopyOnWrite principle. It accepts that where there is contention for updates there is the possibility of significant overheads. There is also the possibility that a particular thread might be continually thwarted in its attempt to change values held by the stored data object. Although it is, perhaps, aesthetically
unpleasing, there is solid engineering experience that keeping the solution simple is often more effective than a more correct, but expensive implementation, such as a FIFO queue.

The rest of this section describes the way in which we use the standard features of the Java language to meet the requirements set out in section VII.3 above.

**VII.5.2. General constraints**

The object reference to the stored data hierarchy must be mutable. However, these changes must be controlled. The attribute that holds the object reference must be **private** so that it can only be read or written through accessor methods. The stored object must conform to a set of limitations so that the requirements are met. We enforce this by insisting that the stored objects are sub-classes of the inner class DataStore.DSObject. We use an abstract class rather than an interface because we wish to enforce a specific construction of the `modify` and `equals` methods.

**VII.5.3. DSObject and DSObjectMutable**

All attributes of all the class instances in the stored data hierarchy must be immutable.

To achieve this we impose the following restrictions:

- All attributes of classes that extend DSObject must be made immutable by declaring them **final**;
- All attributes of such classes must not be declared to be **static**;
- All attributes of such classes must refer to primitive objects or to classes that extend DSObject.

The DSObjectMutable class is also defined. Every class that extends DSObject must have a corresponding class that extends DSObjectMutable. Sub-classes of DSObjectMutable must have the same data structure as the corresponding sub-class of DSObject but provide mutable attributes. This is achieved by omitting the **final** property in the declarations of the attributes.
public class DSObject {
    public DSObject(DSObjectMutable dsm){};
    public DSObjectMutable getMutable(DSObject dso){
        throw new Error("Stored object must override getMutable method");
    }
    final DSObject modify(DSObject dso, Consumer<DSObjectMutable> lambda) {
        DSObjectMutable newdata = dso.getMutable(dso);
        lambda.accept(newdata);
        return newdata.getImmutable(newdata);
    }
    // enforce simple equality of objects
    @Override
    final boolean equals(Object obj) {
        return super.equals(obj);
    }
}

Figure 135 - Java code for DSObject

DSObject defines a constructor that builds an immutable copy from an instance of the corresponding DSObjectMutable object. It also defines a DSObjectMutable.getImmutable() method that must be overridden by its subclasses. This method builds an immutable copy from the mutable attributes of the class instance. DSObjectMutable defines a constructor that builds a mutable copy from an instance of the corresponding DSObject object. It also defines a DSObject.getMutable() method that must be overridden by its subclasses. This method builds a mutable copy from the immutable attributes of the class instance.

Figure 135 and Figure 136 show the Java code for the DSObject and DSObjectMutable classes respectively.

public class DSObjectMutable {
    public DSObject(DSObject dso){};
    public DSObject getImmutable(DSObjectMutable dsm){
        throw new Error("Stored object must override getImmutable method");
    }
}

Figure 136 - Java code for DSObjectMutable
VII.6. Example of practical use

Figure 137 provides an example of the way in which the DataStore class might be used within a larger program.

The class CA extends DataStore. The class CB must extend DSObject and there must be a mutable variant CBW that extends DSObjectMutable. The attributes of CB must be final, the attributes of CBW must not be final. The attribute list of CB must match the attribute list of CBW. The constructors must correctly initialise the attributes. The getMutable() and getImmutable() methods must organise a deep-copy of the class hierarchy headed by CB, if one exists. All this would be set up as part of the project infrastructure.

```java
public class CA extends DataStore {
    DataStore ds = new DataStore(new CB());
    class CB extends DataStore.DSObject {
        final int a;
        public CB() {
            super();
            a = 0;
        }
        public CB(CBW cbw) {
            super();
            a = cbw.a;
        }
        CBW getMutable() {
            return new CBW(this);
        }
    }
    class CBW extends DataStore.DSObjectMutable {
        int a = 0;
        public CBW(CB cb) {
            super();
            a = cb.a;
        }
        CB getImmutable() {
            return new CB(this);
        }
    }
    public class DataStoreDemo {
        CA ca = new CA(new CA().new CB());
        public DataStoreDemo(){}
        void body() {
```
CA.CB copy = (CA.CB)ca.getData();
// process using copy.a
boolean success = true;
// loop that re-considers the new value
// and submits until it succeeds
while (success) {
    // more process that delivers a new value for "a"
    // get latest copy of data
    copy = (CA.CB)ca.getData();
    int temp = copy.a;
    temp++;
    final int b = temp;
    success = ca.putData(dsm ->
    {
        CA.CBW cb = (CA.CBW)dsm; cb.a = b;
    });
}

Figure 137 - Usage of DataStore

Other members of the project would code using the pattern shown as the DataStoreDemo class. Any threaded code may safely obtain a copy of the datastore data by using the getData() method. The obtained data is immutable, but, of course, the user can copy it freely. It is the user's responsibility to recognise that any copied data is stale. Any threaded code may update the data by using the putData() method. The lambda expression may make any desired changes to any attribute of any object in the data hierarchy. It may assume that the object reference it receives points to a hierarchy of mutable objects. The changes will be applied in a uniprocessor execution circumstance and will succeed or fail collectively as if they had been performed atomically. If an invocation of putdata fails, then the invoking code knows that the stored data hierarchy has been changed by another thread. Accordingly, it should re-consider what changes it now wishes to make in the light of this new circumstance. This explains the use of the while(success) loop.

VII.7. Mechanising the DataStore class

The DataStore class, shown in Figure 138, stores an object of type DSObject and has two non-private methods: getData() and putData(). The DSObject getData() method simply returns a copy of the object reference to the stored
data hierarchy as held within the DataStore class. The VarHandle method invocation is implemented by the Java Virtual Machine as a compiler intrinsic that is mapped into an appropriate fence instruction.

In Figure 138, the putData() method takes a lambda expression that accepts an object reference to a mutable copy of the stored data hierarchy. The DSHelper class, which also forms part of our package, implements a static VarHandle to the data attribute. It follows the coding example given in JEP 193 (Lea and Sandoz 2015). This means that the data attribute can generally be accessed without memory fences, but, when needed, can be manipulated with a CompareAndSet operation.

```java
class DataStore {
    private DSObject data;
    final DSObject getData() {
        DSObject result = data;
        VarHandle.acquireFence();
        return result;
    }
    final boolean putData(Consumer<DSObjectMutable> lambda) {
        boolean result = false;
        DSObject olddata = data;
        // No fence needed because of address dependencies
        DSObject newdata = olddata.modify(olddata, lambda);
        if (data.equals(olddata)){
            result = DSHelper.DS_data.compareAndSet(this, olddata, newdata);
            // No fence required because of atomic compareAndSet
        }
        return result;
    }
}
```

The logic for the replacement of the data object is as follows:

- if several threads concurrently attempt a putData operation, they will each create and update their own immutable copy;
- The conditional fails if the current value of data is not the same as that originally read by this thread. Inter-leaving semantics still allows a subsequent change, but this is trapped by the atomic CompareAndSet operation. The conditional reduces the chance of incurring the cost of a CompareAndSet operation in those cases where the data object has
already been updated by another thread. The method returns false if its changes have not succeeded. The caller of the putData method is then free to take whatever action it deems necessary.

The putData(Consumer<DSObjectMutable> lambda) method performs the following steps:

- Create a mutable copy of the stored data hierarchy;
- Apply the lambda expression to the mutable copy;
- Create an immutable copy of the modified data hierarchy;
- Replace the current object reference stored in the DataStore class with a reference to this new immutable copy.

There is no possibility of an ABA-type error because:

- The getMutable and getImmutable methods always create fresh objects;
- The original immutable object, the mutable copy and the new updated immutable object all exist at the same time and must, therefore, have different addresses. The original immutable object and its mutable copy do not become eligible for garbage collection until after the overwriting of the data object reference.

Most of this work is encapsulated within the modify method of DSObject as shown in Figure 139.

```java
final DSObject modify(DSObject dso, 
Consumer<DataStore.DSObjectMutable> lambda){
    // take a mutable copy of the existing data
    DSObjectMutable newdata = dso.getMutable(dso);
    // modify it using the lambda expression
    lambda.accept(newdata);
    // return a new immutable variant
    return newdata.getImmutable(newdata);
}
```

Figure 139 - DSObject modify

VII.8. AEG analysis of DataStore class

Because the code involved in the DataStore class is not extensive, it was easy to reason about the correct selection and placement of fences using the advice presented in the JSR 133 Cookbook (Lea 2008). Because the JEP 193 methods were available, it was not difficult to avoid the inherent overheads
of the implementation of the synchronized and volatile constructs. We did not use the automated algorithm developed in Chapter V because we wished to examine and demonstrate transparently the detail of the interactions between competing getData and putData invocations and between competing putData invocations. As discussed in section V.2.2 of Chapter V, code like that in the putData method has an abstract event graph (AEG) with many cycles over a small number of events. Accordingly, it is sufficient to consider breaking every event-to-event edge with an appropriate fence and then optimise those fences.

The process we followed was:

- Declare the variable that would, conventionally, be declared volatile without using that construct.
- Analyse the accesses to that variable to determine the memory fences that should optimally be used, taking into account the succeeding, and where appropriate, preceding events. This analysis is greatly simplified by the use of an atomic compareAndSet action. This action has the effect of applying full fences round the atomic Read/Write action, which avoids the need for an explicit invocation of a full fence after the Write action.

```java
//Initialisation
DataStore ds = new DataStore(new CB());
...
//Mutator thread
ds.putData(cb -> {cb.a = 42;});
...
//Observer thread
CB cb = ds.getData();
int x = cb.a;
...
```

Figure 140 - Typical use of DataStore

The Java code shown in Figure 140 contains the relevant fragments of a typical use of the DataStore class. The CB class extends DSObject and has a single int variable a. The variables within the stored data hierarchy are always changed privately so there can be no question of a loss of sequential consistency. The only variable where a change is publicly visible is data.
Accordingly, we need only consider this variable in seeking a loss of sequential consistency.

**VII.8.1. Theorem concerning sequential consistency**

**Theorem 4**
The *DataStore* class is sequentially consistent.

We first prove that concurrent executions of the *putData* and *getData* methods are sequentially consistent. Then, we prove that concurrent executions of the *putData* method are sequentially consistent.

**Lemma 5**
Concurrent executions of the *getData* and *putData* methods are sequentially consistent.

**Proof**
Suppose that an Observer thread makes a number of successive invocations of *getData* concurrently with the execution of a Mutator thread.

The corresponding Abstract Event Graph (AEG) for these fragments is shown in Figure 141. The blue arrows indicate program order edges. The pink edges indicate competing-pairs relationships.

![Figure 141 - AEG for Observer versus Mutator threads](image)

On an implementation of the x86 architecture, the *CompareAndSet* action has the effect of *full* fences before and after the *CompareAndSet* action. On implementations of the ARM architecture, the LL/SC combination provides only a weak*CompareAndSet* so that the LL/SC combination must be followed by a *full* fence, implemented by a *dmb* instruction.
We can find the following cycles that we have annotated with the fences provided by the CompareAndSet actions:

1. (a) (b) (e) (a)  
   Full fence after (e)
2. (a) (b) (h) (a)  
   Full fence after (h)
3. (a) (e) (f) (g) (h) (a)  
   Full fences after (e) before (g) and after (h)
4. (a) (h) (b) (e) (a)  
   Full fences after (h) and after (e)

As all these cycles are broken by fences, this use of the DataStore class is sequentially consistent.

This proves lemma 5. ■

**Lemma 6**

*Concurrent executions of the putData method are sequentially consistent.*

**Proof**

We now consider the other possible interaction, which is between two instances of threads that execute the *putData* method. The corresponding AEG is shown in Figure 142.

![AEG for two mutators](image)

**Figure 142 - AEG for two mutators**

In this case the cycles are:

1. (a) (b) (c) (f) (a)  
   Full fences before (b) effectively between (b) (c), after (c) and after (f);
2. (a) (b) (c) (e) (f) (a)  
   As for 1 plus fences in and around (e) (f);
3. (a) (b) (c) (d) (e) (f) (a)  
   As for 2.
4. (a) (b) (f) (a)

Fences before (b) and after (f).

This AEG is symmetrical so that there are corresponding cycles starting from (d) with corresponding fences that break them, so that in this case, also, the use of the Data Store class is sequentially consistent.

This proves lemma 6. ■

Proof of Theorem 4

By lemma 5, interactions between concurrent executions of the getData and putData methods are sequentially consistent.

By lemma 6, interactions between concurrent executions of the putData method are sequentially consistent.

There are no other cases of usage interaction.

This proves theorem 4. ■

VII.8.2. Theorems concerning freedom from data races

Given the sequentially consistent behaviour established in section VII.8.1, this section begins by providing proofs of a set of theorems that together establish the correctness of our design and implementation. We prove that the DataStore package of classes does, indeed, deliver freedom from data races.

In theorem 7 we prove that threads may freely read variables that form part of the stored data hierarchy retrieved from the DataStore through the use of the getData method. Then we prove theorem 13, which states that using the putData method to change the stored data cannot cause data races either on the stored data or on the data reference. Theorem 17 combines these two to give a general assurance against data races.

Theorem 7

Threads that acquire access to a stored data hierarchy through the getData() method cannot cause data races by accessing that stored data.

Lemma 8

In between invocations of the putData method, all threads that invoke the getData method will receive a reference to the same stored data hierarchy as each other.
Proof
The reference variable data is private to the DataStore class instance and can be changed only by the DataStore class itself. Accordingly, in between invocations of the putData() method, the data variable will always point to the same stored data hierarchy which we here denote by \(dh_n\). Hence all threads that invoke the getData method in between executions of the putData method will receive a reference to the same data hierarchy identified by \(dh_n\).
This proves lemma 8. ■

Lemma 9
Threads that obtain access to a stored data hierarchy under the conditions of lemma 8 obtain access to variables that are read-only.

Proof
The data variable is typed as a reference to a DSObject class instance. Correctly constructed sub-classes of DSObject must have all their public variables declared final which ensures that they are read-only.
Let \(V_{dso}\) be the subset of all variables that are declared within DSObject and its sub-classes.
By the construction of DSObject, all variables that are members of \(V_{dso}\) must be read-only. The variable data is private and so threads can access it only through the getData method. All elements within the stored data hierarchy can only be accessed through the variable data. Hence, accessible elements must be members of \(V_{dso}\) and are therefore read-only.
This proves lemma 9. ■

Lemma 10
Threads that access the stored data hierarchy through the getData method in between invocations of the putData method cannot cause data races
Data races require at least one write event. By lemma 8, the threads can only access a stored data hierarchy of which all variables are read-only.
This proves that lemma 10 is true. ■
Lemma 11
Two different threads that access the stored data hierarchy through the getData method respectively before and after an invocation of the putData method cannot cause data races.
Proof
By the construction of the putData method, those threads that invoke the getData method prior to the invocation of the putData method will receive an instance of a stored data hierarchy that we denote by $dh_0$ while those that invoke getData after the invocation of putData will receive a different instance that we denote by $dh_1$. Formally,

$$dh_0 \neq dh_1 \tag{121}$$

Let $V_{d_{h_0}}, V_{d_{h_1}}$ denote the sets of variables in the hierarchies $dh_0, dh_1$ respectively so that, for a data race to exist on a variable $v$.

$$v \in V_{d_{h_0}} \land v \in V_{d_{h_1}} \tag{122}$$

so that

$$v \in V_{d_{h_0}} \cap V_{d_{h_1}} \tag{123}$$

But, by the construction of the putData, modify and getImmutable methods,

$$V_{d_{h_0}} \cap V_{d_{h_1}} = \emptyset \tag{124}$$

and so

$$v \in \emptyset \tag{125}$$

So, there can be no data race variable.

This proves lemma 11. □

Lemma 12
There is no interval between the "before" and "after" conditions quoted by lemma 11.
Proof
By the construction of the putData method, the substitution of a new stored data hierarchy in replacement of the current data reference is accomplished by an atomic memory access. By the definition of atomic, no other access to the same variable can occur until all the components of the atomic access are complete. Hence, the access to the data reference within the getData method must occur either before or after the change made by the putData method.

This proves lemma 12. □
**Proof of Theorem 7**

By lemma 10, threads that access the stored data hierarchy through the `getData` method in between invocations of the `putData` method cannot cause data races by accessing the variables in the hierarchy.

By lemma 11, if a `putData` method invocation intervenes between invocations of the `getData` method the threads cannot cause data races by accessing the variables in stored data hierarchy.

By lemma 12, there are no other cases.

Accordingly, the theorem 5 is true.

**Theorem 13**

*Concurrent invocations of the DataStore.putData method cannot cause data races.*

**Lemma 14**

*The mutable copies created by concurrent invocations of the putData method are identical.*

**Proof**

It is trivially obvious that if the change to the `data` reference made by one thread occurs before the reading of `data` by another thread then the threads are being executed in a sequential rather than a concurrent manner.

By the construction of the `putData` method, the `data` reference is only changed by an atomic write action at the end of the method. Hence, one and only one of the concurrent threads can succeed in changing the `data` reference. All read actions that occur prior to that event must receive the same unchanged value for `data`. Again, by the design and construction of the DataStore class and the sub-classes of DSObject and DSObjectMutable, all the attributes of all the objects are copied. As each copy action makes a copy of the same object, the mutable copies must hold the same values for every attribute within the data hierarchies.

This proves lemma 14.

**Lemma 15**

*The concurrent execution of the lambda expressions associated with different invocations of the putData method cannot cause data races.*

238
Proof
By the construction of the \textit{putData} method, each thread that invokes the method creates its own distinct mutable copy of the stored data hierarchy. By lemma 14, the attributes in the copies are distinct but hold copies of the same value.

As before we consider two threads that create distinct data hierarchies that we denote by $dh_0$ and $dh_1$. A data race on a variable $v$ can exist only if it is a member of both data hierarchies, which, as before, means that

\[ v \in V_{dh_0} \cap V_{dh_1} \]  

But, by the construction of the \textit{putData} method,

\[ V_{dh_0} \cap V_{dh_1} = \emptyset \]

therefore,

\[ v \in \emptyset \]

which means that, in this case, there can be no data races.

This proves lemma 15. ■

Lemma 16
\textit{Concurrent invocations of the putData method cannot cause data races on the data reference.}

Proof
When the putData method has applied the lambda expression to its mutable copy of the data, it creates a new immutable copy from the mutable copy and then changes the data reference to refer to this new immutable copy.

By the construction of the putData class, the data reference is changed only if its current value matches the value read at the start of the method. This is achieved by the use of \textit{atomic CompareAndSet} functionality. If this action fails, the method is aborted because a concurrently executing method has already applied its lambda expression and updated the data reference. The failure of the second and subsequent concurrent putData method invocations is reported to the caller(s) so that they can become aware that their changes have not been applied. The use of this logic ensures that only one concurrently executing putData invocation can succeed. Data races require more than one interfering write event.

This proves that lemma 16 is true. ■
Proof of Theorem 13
By lemma 14, all concurrent invocations of the putData method must act on identical copies of the stored data hierarchy as currently visible to threads that invoke the getData method.
By lemma 15, the application of the lambda expressions to these mutable copies cannot cause data races because the copies are distinct.
The remaining possibility is a data race for the data reference, which is excluded by lemma 16.
Accordingly theorem 13 is true. ■

Theorem 17
The DataStore class and its associated classes prevent data races on the variables in the stored data hierarchy.

Proof of Theorem 17
By theorem 7, threads that obtain access to the stored data hierarchy through the getData method cannot cause data races on variables in that hierarchy.
By theorem 13, threads that change the values of variables in the stored data hierarchy using the putData method cannot cause data races on variables in that hierarchy.
The data reference is private so that it and the data hierarchy to which it refers cannot be accessed other than through the getData and putData methods.
There being no other possibilities, theorem 17 is true. ■

We have shown that all possible usages of the DataStore methods give rise to sequentially consistent behaviour. Relying on this result, we have shown that all possible usages of the methods cannot cause data races so that the class is proven to be thread-safe. These proofs also show that the mutator method, putData, satisfies the atomicity, consistency and isolation properties of the ACID definition.

VII.9. Effects of erroneously constructed sub-classes
The proofs in the preceding section rely on the correct construction of the sub-classes of DSObject and DSObjectMutable. We continue by considering
the various ways in which the assurance of freedom from data races provided by the DataStore class might be impaired by the incorrect construction of these sub-classes. We have identified a number of possible errors that might occur in the construction of the sub-classes:

- **Non-final attributes.**
  Some attributes are not declared as `final`;
- **Data structure mis-matches.**
  Attributes are omitted from the either the mutable or the immutable variant;
- **Object references within sub-classes are not copied by creating a new object.**
  - Failure to invoke `getMutable`;
  - Failure to invoke `getImmutable`;
- **Collections are not immutable.**
  Readers must not be able to `add` or `remove` elements of collections within the sub-classes;
- **static variables.**
  Some attributes are declared as `static`;
- **Over-riding the equals method.**
  Object references stored in the `data` attribute must return `false` from the `equals` method even if all the values of attributes of objects in the stored data hierarchy are the same.

Each of the following sub-sections deals with a particular counter example from this list.

**VII.9.1. Violations of read-only access**

The `data` object reference points to a hierarchy of objects. Every class in that hierarchy is a sub-class of DSObject.

Suppose that a class that uses the DataStore erroneously attempts to write to an attribute of one of these classes, for example by using code similar to that shown in Figure 143.
DataStore ds = new DataStore(new MyClass(42));
...
MyClass mc = ds.getData();
mc.a++;

Figure 143 - Erroneous write to attribute

If MyClass has been correctly constructed, the attribute a will have been declared final and the compiler will report an error. If the attribute a has not been declared final, then the write action will be legal and a data race situation will exist. There are a number of different ways to close this loophole:

- Manually inspect the source code to ensure that this pre-requisite condition is satisfied;
- Use a source code pre-processor or plug-in to automate the generation of DSObject sub-classes from DSObjectMutable sub-classes;
- Enhance the DSObject constructor to use Reflection to examine the sub-classes and ensure that all the attributes have the final property.

VII.9.2. Data structure mis-matches

Suppose that there is some attribute p that exists in the DSObject sub-class, but not in the corresponding DSObjectMutable sub-class. This attribute will be permanently immutable. This is undesirable, but does not cause data races. If the DSObject constructor and the getImmutable() method are built to the best standards, the compiler will report at least a warning because the attribute p will not be initialised.

Suppose, conversely, that there is some attribute p that exists in the DSObjectMutable sub-class, but not in the corresponding DSObject sub-class. The attribute will be mutable but will be absent from the object returned by getData(). Once again, this is undesirable, but does not cause data races. If the DSObjectMutable constructor and the getMutable() method are built to the best standards, the compiler will report at least a warning because the attribute p will not be initialised.

There are similar undesirable but safe situations where the code of the constructors and/or the getMutable() and getImmutable() methods erroneously omits the copy of particular attributes.
VII.9.3. Copying object references

We have established in VII.9.2 that the effect of failing to copy attributes is undesirable, but safe. In this section, we consider the effects of dealing incorrectly with object references within the sub-classes.

If an attribute of a sub-class of DSObject is not primitive, but is a class instance, the building of an immutable object must be invoked successively for all the objects at every level in the hierarchy. If, at any level, the object reference is not to a sub-class of DSObject then the attempt to invoke getMutable() will cause a compiler-time error. If, at any level, the getMutable() code erroneously treats the object reference as a primitive and performs a simple copy, the supposedly mutable copy will contain immutable attributes so that the execution of the lambda function will fail at run-time. This is undesirable, but safe.

Similarly, if object references are not sub-classes of DSObjectMutable then attempts to invoke getImmutable() will fail at compile-time. If the getImmutable() simply copies the reference, the supposedly immutable copy will contain references to mutable objects. This will cause data races if the code that uses the DataStore class also contains erroneous attempts to write to the attributes of objects retrieved by using getData(). As noted earlier in section VII.9.1, these loopholes may be closed by inspection, by using a source code pre-processor or by automating the copy process using Reflection.

Sections VII.9.1, VII.9.2 and VII.9.3 together establish the conditions and developmental processes that must be applied to ensure that errors in the construction of the DSObject and DSObjectMutable classes cannot impair the integrity of the use of the DataStore class.

VII.9.4. Collections within the stored data hierarchy

The present implementation implicitly assumes that none of the objects in the stored data hierarchy is a Collection. Adding elements to a Collection or removing them, if permitted, would violate the principle of immutability that is at the heart of the design of the DataStore class. This must be prevented by providing a wrapper class for each Collection that makes the mutation
methods invalid except where the hierarchy is mutable. Once again, the correctness of the code of the DSObject and DSObjectMutable sub-classes must be assured by inspection or through the use of a source code pre-processor.

**VII.9.5. Use of static variables**

The use of static attributes within the stored data hierarchy will cause data races during the update process. This must be prevented as noted earlier by inspection, by using a source code pre-processor or by automating the copy process using Reflection.

**VII.9.6. Immutable equals method**

It would be possible for the code of the putData method to be invalidated by sub-classes of DSObject that over-ride the equals method. We have eliminated this source of error by declaring the method as final within the DSObject class.

**VII.9.7. Summary of error prevention**

We have analysed all the ways in which errors in the construction of DSObject and DSObjectMutable sub-classes might impair the use of the DataStore class. We have shown that all these errors may be eliminated by inspection of the code, by use of an appropriate source code pre-processor or, perhaps, by the use of Reflection techniques.

In the next section we provide an evaluation of the performance of our DataStore class against functionally equivalent code that uses more conventional synchronisation techniques.

**VII.10. Performance**

This section deals with the evaluation of the performance of our DataStore class. To form a baseline against which we might compare its performance, we built two other variants that used less advanced synchronisation techniques. In particular, we built a variant that relied on the synchronized construct. We describe the three variants and follow this description with a presentation of the results of our evaluation of the performance of our DataStore class against the other variants.
VII.10.1. Class variants

For evaluation purposes, we built three variants of the DataStore class.

The DataStore class represents our recommended most efficient implementation. The essential difference between the variants is contained within the `putData` method. The `putData` method for the DataStore class is shown in Figure 144. In this variant, there is no restriction on readers. Multiple mutators can execute concurrently, but the use of the `compareAndSet` operation ensures that only one will succeed and all others will fail.

```java
public boolean putData(Consumer<DSObjectMutable> lambda){
    boolean result = false;
    DSObject olddata = (DSObject)data.get();
    // No fence needed because of address dependencies
    DSObject newdata = olddata.modify(olddata, lambda);
    if (olddata.equals((DSObject)data.get())){
        result = data.compareAndSet(olddata, newdata);
        // No fence required because of atomic compareAndSet
    }
    return result;
}
```

Figure 144 - DataStore putData method

The VDataStore variant relies on the properties of `volatile` variables to propagate the change to the data reference and uses the `synchronized` construct to ensure that only one mutator can execute at a time. The Java code for this variant is shown in Figure 145.

The SDataStore variant simply uses the `synchronized` construct on both the `getData` and `putData` methods. This provides a baseline that reflects a conventional implementation of the class.
public class VDataStore extends DataStore {
    private static volatile DSObject data;
    public VDataStore(DSObject d){
        super();
        data = d;
    }
    public final DSObject getData(){
        return data;
    }
    public final synchronized boolean putData(Consumer<DSObjectMutable> lambda){
        data = data.modify(data, lambda);
        return true;
    }
}

Figure 145 - Java code for VDataStore class

We experimented with using the VarHandle compareAndSet method so that the data object could be a simple reference rather than an AtomicReference. This gave a small, but noticeable improvement in performance. However, technical difficulties with the Java 9-ea builds made it hard to automate the testing of this variant so we chose to defer further work on that variant until these difficulties are resolved.

VII.10.2. Results

In section III.1.2 of Chapter III, we described our extensions to the Synchrobench (Gramoli 2015) benchmark. We further modified these extensions to automate the testing of all three DataStore variants using a variety of different write percentages. We obtained output that we represent graphically in Figure 146, Figure 147, Figure 148 and Figure 149. These graphs show how the performance, measured as operations per millisecond, varies as the number of threads is increased.

This set of graphs shows that the performance of our DataStore implementation is always superior to a conventional synchronized implementation even where there is a high percentage of write operations compared to read operations. This is gratifying, but unexpected. It appears that there is an enormous benefit from the lock-free read access.
Figure 146 - Comparative performance 20% write

Figure 147 - Comparative performance 40% write

Figure 148 - Comparative performance 60% write

Figure 149 - Comparative performance 80% write
This seems to out-weigh the obvious costs incurred in copying the entire stored hierarchy twice during the update process. We summarise the benefit afforded by the DataStore class by normalising its performance against that of the SDataStore class. We do this by taking the ratio of the number of operations per millisecond achieved by the DataStore class against the number of operations per millisecond achieved by the SDataStore class. There are four data series for different percentages of write actions. We show this comparison in Figure 150.

![Figure 150 - Normalised DataStore performance](image)

This shows that where the percentage of write actions is low, the DataStore class provides about eight times better performance. Where the percentage of write actions is above 50%, this reduces to about twice. This is still a very significant gain.

In the next section, we consider a possible explanation for the, otherwise, anomalous behaviour of the SDataStore variant exhibited in Figure 146 and Figure 147 when compared to Figure 148 and Figure 149.

### VII.10.3. Impact of C2 optimisations

The results show that as the percentage of write actions rises above 50% the performance of the SDataStore class doubles from about 5000 ops per millisecond to about 10 000 ops per millisecond. In this section we present an analysis that provides a possible explanation for this behaviour.

The C2 compiler transforms the code of a method so that there is a "straight through" path with out of line branches. It uses the profile collected during the execution of C1 generated code to determine which branch should be made "straight through". We consider here the case where a method tries
to acquire a lock. There are two possibilities: the attempt may succeed or fail. If there is a low level of contention for the lock then the attempt will generally succeed. Where there is a high level of contention, the attempt will most frequently fail. At either of these two extremes of the execution envelope the choice is clear and the compiler will generate machine code that is well tuned for execution on a pipe-lined CPU. However, as the degree of contention changes from one of these extremes to the other it passes through states in which the decision is marginal. Suppose the profile shows that attempt succeeds 51% of the time. Unless the compiler specifically recognises and deals with this finely balanced decision, it will assign the "straight through" path to the successful attempt. When this code is executed, the stochastic variability of the environment ensures that about half the time, the "straight through" path will be taken and about half the time the execution will take the out-of-line branch. We would expect that this would cause an observable drop in the measured performance.

If the compiler recognises the situations where the decision is marginal, it has two options: it can make a decision in favour of one branch; or it can treat both options equally and make both of them into out-of-line branches.

Reverse-engineering the intent of the compiler from the generated machine code is difficult and very time-consuming. We have not succeeded in setting up a controlled environment that facilitates the systematic investigation of these effects. Consequently, we cannot definitively say which of the options discussed above has been implemented. However, this discussion may provide an explanation for the observed behaviour of the SDataStore class.

VII.11. Limitations and potential for improvement
This section discusses the practical impact of the known limitations of our design and suggests avenues for improvement.

VII.11.1. Limitations
The copying of attributes between the mutable and immutable classes relies on the correctness of the constructor code of these classes. Where classes use extensive sub-classing it is good practice that the super-classes should
not be aware of any additional structure created by sub-classes. Unless this principle is violated or special coding measures are taken, the presence of an extensive hierarchy of sub-classes will cause difficulties in the use of the DataStore class.

**VII.11.2. Our current implementation**

Our implementation of the algorithm involves the following list of actions to implement the *putData* method:

1. Make a mutable copy of the current DSObject. This involves physically copying every field and creating a new instance of every object in the stored data hierarchy. The JVM will force a full memory fence at the end of the constructor method of every created object.

2. When the changes have been applied, make an immutable DSObject from the changed mutable copy. This involves another copy of every field and the creation of a new instance of every object. A full memory fence will be executed for every created object.

3. The reference to the old data is overwritten with the reference to the new data. This means that the old data becomes eligible for garbage collection.

4. When the *putData* method returns, the reference to the mutable copy of the changed data is lost so that it becomes eligible for garbage collection.

In summary, the process requires:

- two complete copy actions of all the data in the stored data hierarchy;
- garbage collection of two copies of the data;
- execution of two full fences for every object in the stored data hierarchy.

Some of this overhead may be avoided by keeping a pool of mutable class instances that are continually re-used by the *putData* method. This avoids one of each pair of the full fence executions and the cost of the garbage collection of each used mutable class instance. A similar technique cannot be applied to the immutable classes because a reader may be holding a reference to an old immutable class instance.
VII.11.3. Our proposal for a Java language enhancement

Daloze, Marr et al. (2016) describe the incorporation of a thread-safe object into their design for the addition of support for multi-threaded execution to dynamically typed languages such as JRuby (Nutter, Enebo et al. 2011). Their thread-safe objects are similar to our DataStore class in providing lock-free read access and controlled write access. This suggests that there may be scope for enhancements to the Java language specification that would include our DataStore mechanism but hide it behind a more aesthetically pleasing syntactic façade. In this sub-section, we outline some of the directions that such an enhancement proposal might consider.

The programmer must be able to indicate that instances of a class should be treated as a shared data store. We suggest that this might be achieved by extending the implications of the `synchronized` keyword. We propose that the use of the `synchronized` keyword in the definition of a class would cause that class to be treated as if it were a sub-class of DataStore. We suggest the use of syntax similar to that shown in Figure 151.

```java
synchronized class MyShared {
    Shareable data;
    ...
}
```

*Figure 151 - Proposed syntax for synchronized class*

The javac compiler would enforce the rule that objects declared as references within a `synchronized` class must implement the `Shareable` interface. The bytecode definition would be extended to accommodate `synchronized` as a valid attribute of a class.

At runtime, code would be able to read `data` and any variable of any object within stored data hierarchy in a lock-free manner. However, write access would be permitted only within a `synchronized` block in a manner similar to that shown in Figure 152.
class Example {
    MyShared ds = new MyShared();
    ...
    void body() {
        int r1 = ds.data.p.q.r;
        ...
        synchronized (ds) {
            ds.data.p.q.r = 42;
            ds.data.p.q.s = "Amazing grace";
            ...
        }
        ...
    }
}

Figure 152 - Controlled write access

The Java Virtual Machine would recognise the use of a synchronized class as the operand of MONITORENTER and MONITOREXIT instructions and treat it in a manner functionally equivalent to the invocation of the DataStore putData method. The contents of the synchronized block would be treated as the lambda expression parameter.

The javac compiler would report a compilation error for any write statement that referred to any variable within the stored data hierarchy that was syntactically located outside a synchronized block that used the corresponding synchronized class as its lock. When presented with a class that declared Shareable as one of its implemented interfaces, the javac compiler would generate mutable and immutable variants of the defined class as described in section VII.5.3.

It is a moot question what syntax should be used to handle the case where an attempted change to the stored hierarchy fails. One solution would be to allow the synchronized block to throw a ConcurrentUpdateException. The caller could trap this with the standard try/catch syntax. The disadvantage of this approach is that concurrent update is a rare, but expected condition. The Java Exception mechanism is relatively heavy-weight, reflecting the expectation that exceptions most usually reflect an error condition. Although aesthetically less pleasing, a pragmatic solution might be to report nothing and rely on the caller to detect the absence of the desired changes by reading a suitable flag variable within the data hierarchy.
This is safe because the actions in the `synchronized` block are guaranteed to be atomic, either they all occur or they all do not occur. This approach might be facilitated by the provision of a conventionally named variable in the `synchronized` class that is time-stamped during the execution of the `synchronized` block.

Depending on the detailed implementation of the `final` construct, it might be possible to avoid one of the two copy actions presently needed as part of the update process. The last copy could be avoided if it were possible to dynamically change the `final` attribute of every variable in the stored data hierarchy.

In our "built-out" implementation of the DataStore class, it is not possible interfere with the full fence actions taken as each of the new immutable objects is initialised. A "built-in" solution would have the possibility of inhibiting these fence actions. We do not wish to publish the immutable copy until it is complete so the final automatic fence action implied by the `CompareAndSet` action on the `data` variable would suffice.

**VII.12. Summary**

The popular `acquire/release` paradigm is crucially dependent on the correctness of the code executed within the critical sections. Data races occur if that code does not respect the implicit access protocol. A static analysis to detect these errors is, at best, computationally expensive. Though our algorithm for finding data races is sound for a significant selection of commonly used coding patterns it is not sound in all cases and it is not complete. Trying to reduce the computational complexity increases the probability of false positive reports.

On the other hand, our DataStore class offers superior performance over a wide range of operating conditions while providing a robust framework that enforces the access protocol and prevents the occurrence of data races. All the complication is concentrated and encapsulated within the definition of the stored classes. Conversely, the run-time usage is simple and incurs low overheads. As demonstrated in the analysis of our prototype, the guarantees provided are significantly superior to those provided by the
*acquire/release* paradigm. We have indicated how the remaining loopholes may be closed by external actions taken either as part of the compilation process or during initialisation of the class. We have, further, provided some indications of how the concept of the DataStore class might be integrated more closely within the structure of the Java language. This would improve the ease-of-use and would offer the possibility of further gains in efficiency.

We believe that the definition of this class is an important contribution to work in this area.
Chapter VIII Conclusions

"Just the place for a Snark! I have said it thrice. What I tell you three times is true."
"The Hunting of the Snark"
Lewis Carroll

We divide this chapter into three parts. In the first part we provide a summary of our contributions. Then, we provide some indications of directions for future research. We conclude with some final thoughts.

VIII.1. Our Contributions

In Chapter I, section 1.2, we posed the questions:

• What can be done to facilitate the writing of multi-threaded Java programs that are free from data races? and
• How best to minimise the consequential overheads?

We have responded to these questions with a number of different but inter-related contributions. This section summarises our contributions as they have been presented in the earlier chapters.

VIII.1.1. Benchmarking de-limiter patterns

We have extended an open-source benchmark to support the systematic investigation of the performance of different de-limiter patterns when used in conjunction the standard Java Collection classes. Using this framework we have conducted an extensive sequence of tests to investigate this performance across a wide range of operating conditions. The important conclusion from these results is that the use of a custom de-limiter pattern provides improved efficiency particularly in the crucial cases where the level of contention for the lock is high.

VIII.1.2. Finding data races

We have provided experimental evidence that it is possible to find the critical sections and memory events within a multi-threaded Java program and
detect data races caused by the failure to respect the implicit access protocol of critical sections. We achieved this by implementing our algorithm within a static analysis program. Preliminary experimentation confirmed the anecdotal evidence that it might be beneficial to use more efficient critical section de-limiters. Accordingly, our implementation made provision for detecting a wide range of de-limiter instruction patterns.

The abstract event concept that we use makes it impossible to distinguish between accesses to different elements of a Collection. We have devised the Summarised Abstract Event Graph (SAEG) extension to the AEG notation so that the actions of Java streams can be summarised and incorporated within an AEG. This overcomes the limitation for those programs that use the Java streams paradigm.

Experimental measurements of the performance of this algorithm showed that the use of approximations and summarisations at a method level, together with the judicious use of multi-threading techniques, provide the ability to process programs of a good size within an acceptable time. However, we fully acknowledge the limitation that there is a balance between execution time for this process and the extent to which it delivers false positive reports.

These approximations arise from pre-requisite conditions that are not easily satisfied within a static analysis. However, within the JVM, the conditions created by the JIT compilers are those needed to satisfy the pre-requisites. Although a detailed knowledge of the internal workings of the JVM was not seen as within the original scope of our work, we have been able to provide a theoretical examination that shows how the techniques developed and tested within a static analysis environment might be beneficially injected into the environment of the JIT compilers.

VIII.1.3. Restoring sequential consistency

Previous work (Nimal 2014) showed that, for C programs, it is possible to perform a static analysis to select and place an optimally minimal set of memory fences. This ensures sequential consistency. We have re-implemented this technique for the Java environment with appropriate
modifications. In particular, we have divorced the selection of fence types from the instruction patterns needed to implement them in particular architectures.

Nimal’s algorithm relies on the use of Abstract Event Graph (AEG) analysis. As a static analysis, the AEG analysis technique suffers from limitations of scalability. There are challenges in handling the explosion of cycles that occurs with intensely interacting threads. Consequentially, the technique cannot be usefully deployed in the analysis of whole programs and must be restricted to the analysis of de-limiter patterns. However, we believe that, coupled with the enhanced implementation of memory fences described in section VIII.1.4, this static analysis can make a worthwhile contribution to the development of efficient multi-threaded Java programs.

VIII.1.4. Optimal implementation of memory fences

Weak memory architectures, such as ARM and POWER, provide a number of different instruction sequences that may be used to ensure cache consistency and, thence, the property of sequential consistency across a set of concurrently executing threads. Earlier work suggested that changes to the instruction sequences used to implement fences might be made on the basis of the target architecture. Our research has found evidence that the different implementations of these architectures give different execution costs for the same instructions. This means that optimising the implementation of fences must recognise the target architecture implementation and not just the architecture. We have provided a proof-of-concept that it is possible to use the features of the Graal compiler to recognise that, where the original instructions include an address dependency, the specified memory fence may be unnecessary. Conversely, it may be possible to generate alternative instructions sequences for a particular example of a fence instruction.

VIII.1.5. DataStore class

The Disruptor class (Thompson, Farley et al. 2011) showed that it was possible to build a substantially lock-free class for efficient message passing. Our novel DataStore class provides similarly efficient access to a hierarchy of stored data objects by using an extension of the CopyOnWrite principle. Our
reported benchmark testing of the prototype implementation provides experimental evidence that the technique is efficient over a wide range of operating conditions, while giving a superior guarantee of freedom from data races.

VIII.2. Future directions
This section considers the way in which various aspects of our research might be carried forward.

VIII.2.1. Benchmarking synchronisation techniques
The results described in Chapter III were obtained by executing the benchmark on an x86 target environment. However, the benchmark framework described is pure Java so that it should work equally well on a Java Virtual Machine (JVM) that targets a weak-memory architecture, such as ARM. This would allow a comparison of the relative efficiencies of the implementation of synchronized, volatile, etc. by JVMs that target the different architectures. This piece of research must wait for the general availability of JVMs that target weak-memory architectures.

VIII.2.2. Finding data races
As well as the features described earlier, the Graal project has developed an Ahead-Of-Time (AOT) mode of operation, which will be supported for experimental purposes from the release of Java 9. In this mode, the program that is being compiled is subjected to an initial execution to generate execution profiles that may then be used to direct a Graal compilation. This compilation takes place Ahead-Of-Time rather than at runtime and can, therefore, use optimisation techniques that might be too costly to use in a production circumstance. We can envisage the possibility of generating the IR graphs for a number of different threads. These threads might be the same method with different execution profiles or different methods, as in a message-passing scenario. In an AOT mode, it would be possible to generate a composite control-flow graph from these IR graphs that uses the "straight-through" path from each thread. Such a composite control-flow graph would still have the resolved conditionals and resolved addresses needed to satisfy
the pre-requisites for efficient extraction of critical sections and memory events. Our data race detection algorithm might then be applied to search the most probable execution paths for data races. This would be a useful result.

**VIII.2.3. Optimal selection and placement of fences**

The AOT mode of the Graal compiler admits the possibility of considering the interaction between different threads that may have very different "straight through" execution paths. Using this mode, it would be possible to build a composite AEG from the various IR graphs of the threads that represented the actual interactions rather than those that might occur. This, in turn, opens the possibility of eliminating some fences and replacing others by less expensive options. This would involve a knowledge of the relative costs of different fences that is beyond the scope of the present research.

**VIII.2.4. Implementation of memory fences**

We have shown that it is possible to change the definition of some of the Graal compiler's IR graph nodes to provide different implementations of memory fences. However, there is evidence that different hardware implementations of the weak memory architectures, such as ARM and POWER, may require using different instruction sequences to achieve the best optimisation. In some cases, no fence is actually required because of characteristics such as the respect for address dependencies. Designing such optimisations would require knowledge of the hardware implementations of particular architectures that is beyond the scope of the present research.

**VIII.2.5. Avoiding data races**

The success of our DataStore class relies on the correct construction of the stored objects in both their mutable and immutable variants. We have identified the minority cases where errors in construction may cause data races. We suggest that for a production implementation, these errors might be avoided through the provision of a generator utility. We note that the use of a pool of instances of the mutable classes offers the possibility of a further reduction in the incurred overheads.
The object of this aspect of our research was to show that the creation of a thread-safe data store that used only the standard features of the Java Language Specification was possible and that its performance would be acceptable in industrial circumstances. This was achieved. However, this "built-out" solution exposes the developer to significant complication in the construction of the classes that form the stored data hierarchy. To improve the usability, we have described the essential content of a Java Enhancement Proposal that would provide a more aesthetically pleasing user interface. It would also provide the opportunity for further improvements in performance.

**VIII.3. Final thoughts**

Without multi-threaded execution it is not possible to exploit fully the power of contemporary CPU chips. This is important because the laws of physics have curtailed the previous annual exponential improvement in their execution speeds.

Writing multi-threaded programs is difficult and error-prone. In this thesis we have investigated many of the problems associated with the writing of error-free multi-threaded Java programs. We have shown that there is substantial benefit to be gained from using the most sophisticated techniques to organise the co-operation between the threads of a program. Such co-operation can be effectively encapsulated so that the developer can rely on well-established efficient techniques without a deep knowledge of their detail.

We have shown that the JIT compilers within the JVM are well placed to support further optimisations and have shown how techniques that we have tested in the static analysis environment may be usefully injected into the JVM environment.

We believe that, taken together, these various contributions significantly improve the ability of developers to build efficient error-free multi-threaded Java programs.
Bibliography


http://dacapobench.org/ Dacapo Benchmarks.


http://openjdk.java.net/projects/graal/ "Graal Project."

http://raja.sourceforge.net/ Raja raytracer program.

http://www.epcc.ed.ac.uk/research/java-grande/ Java grande benchmark.


http://www.w3.org/Jigsaw/ W3C web server jigsaw.


https://github.com/graalvm/ Graal repository.


javailp.sourceforge.net "Java Integer Linear Program interface."


sat4j.org "SAT4J Integer Linear Program solver."


Author/s: Clarke, David Anthony Winscom

Title: Analyses of Java programs over weak memory

Date: 2018

Persistent Link: http://hdl.handle.net/11343/213892

File Description: Analyses of Java programs over weak memory Complete thesis

Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.