Java Lock Contention Antipatterns and Their Detection within Java Code

by

Joseph Robertson

A thesis submitted to the

School of Graduate and Postdoctoral Studies in partial

fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

April 2023

THESIS EXAMINATION INFORMATION

Submitted by: **Joseph Robertson**

MSc in **Computer Science**

Thesis title: Java Lock Contention Antipatterns and Their Detection within Java Code

An oral defense of this thesis took place on Monday, April 3 2023 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Loutfouz Zaman
Research Supervisor	Dr. Ramiro Liscano
Research Co-supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Jeremy Bradbury
Thesis Examiner	Dr. Ken Pu

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Java Lock Contention Antipatterns and Their Detection within Java Code

Joseph Robertson

April 20, 2023

Abstract

Java Based Multithreaded programs are used in a wide variety of applications and consequently many developers are required to create code designed for synchronized environments. However, finding problems in synchronized code can be a time-consuming task, and an inability to properly find and fix all problems results in contention problems and failures. Currently the approach used to find these problems is to run the code and, if problems are found, further investigate related areas. We have created a static analysis program that examines input Java code and checks it against a series of antipatterns to determine possible issues. The program was tested on several programs designed as examples of the antipatterns, and a set of open-source code.

It was found that some of the antipatterns created do appear in open-source java code, and the tool created for their identification was able to reliably locate them in testing and open-source code.

***Keywords*—** Java Concurrency; Lock-Contention; Run-time Faults; Static Analysis; Antipatterns;

Author's Declaration

Joseph Alexander Robertson

Statement of Contributions

A majority of this work was performed by me, some components appear in other previous work that I was a part of:

N. H. Khan, J. Robertson, R. Liscano, A. Azim, V. Sundaresan, and Y.-K. Chang, "Lock contention performance classification for java intrinsic locks," eng, in Runtime Verification, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 274–282, ISBN: 9783031171956.[1]

J. Robertson, A. Ahmed, A. Azim, R. Liscano, V. Sundaresan, and Y.-K. Chang, Java lock contention classification and recommendation through synchronization antipatterns[2]

R. Liscano, A. Azim, J. Robertson, et al., A preliminary investigation into runtime fault identification and localization of java-based cloud-native microservice applications.[3]

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 INTRODUCTION

Multithreading is used in a variety of applications as it makes it possible for a program to organize multiple tasks to while having them avoid obstructing each other, while still reaching the correct result. Due to this there is a demand for software that is able to automate the detection of problems that occur in these systems.

In Java, which is one of the most widely used programming languages and the focus of this thesis, concurrency is handled primarily through the use of synchronized blocks and methods. This allows the programmer to control the interaction behavior of objects through the monitors. Each object in java has an intrinsic lock, also known as monitor lock or monitor, that allows the program to control access to objects and ensure that code is executed in an order that satisfies the dependencies.

This means that a programmer can easily ensure that threads will not interfere with each other's processes, without needing to manually implement that at a system level. However, if concurrent programming is used it will also introduce contention, this is expected and introduces acceptable overhead in an ideal situation, but if the contention reaches a point where it interferes significantly with the execution of the program then it becomes an issue.

One approach used to find potential instances of lock contention problems in source code is static analysis. Static analysis is the process of examining code without its execution. This has the advantages of directly analysing the code, being faster than dynamic analysis, not requiring a full test environment, and not being reliant on runtime conditions. This has the trade off of not being able to directly run code and confirm any conclusions.

A method of testing that could check for lock contention problems would execute the program while measuring performance statistics and if those match with numbers that indicate lock contention issues, then a possible issue would be identified, examples of dynamic analysis tools are async-profiler or perf[4]. Additionally, it can be combined with dynamic analysis to give a more in-depth look at the possible issues with a program.

As the application described in this work targets concurrent programming specifically, it is designed to work with a dynamic analysis tool to first confirm the presence of contention which it cannot be reliably done through static analysis alone.

The program, that will be described in further detail later in this work, will take in code from synchronized regions which show signs of contention and then decide which of a series of patterns that it may belong to. These are Hot1,Hot2,Hot3_1, Hot3_2, Overly Split, Simultaneous, and Unpredictable. The Hot Antipatterns relate to one portion of the code that generates the problem, Overly Split describes a group of critical sections that cause an issue due to increasing the volume of requests, Simultaneous is a incorrectly handled synchronization, and Unpredictable is a problem, usually syntactic, that causes the program to contend or fail unexpected. Of these types the Hot antipatterns are the most important and represent the largest group in actual code. The antipatterns will contain the name, a definition, how it may be found or sings of its presence, example code, and general recommendations.

1.2 Motivations

Many currently used static analysis tools are not designed for detecting contention problems, and antipattern definitions of contention issues are frequently informally defined. Static analysis tools normally focus on finding bugs and making sure that code sticks to coding standards, this work to find a majority of problems with a given program, however it is not effective at finding or solving contention problems. This means that developers need to struggle to find the actual cause of a problem or consult someone with specialist knowledge to find the source of the problem. If an automated system could find the source of the problem and describe what is wrong, then it would make the creation of concurrent programs much easier for many developers.

However, the informal nature of many code patterns makes them unsuitable for an automated approach, some systems resolve this by using other systems to introduce patterns such as a User Interaction Model[5], but that requires a pre-existing dataset and needs the classifier to be able to differentiate between the positive and negative examples, which is not simple with concurrency issues.

Therefore, we propose an automated static analysis system that will use a new set of code concurrency antipatterns to classify problematic contended regions accordingly and propose solutions that are applicable to each.

1.3 Research Questions

As this work focuses on both pattern definitions and classification both must be addressed in this paper. The research questions that this work hopes to answer are:

- Are the proposed antipatterns represented in real world code?
- Can recommendations based on the type of the antipattern be constructed provide unique solutions?
- How reliably can this system differentiate between anti-patterns?

1.4 Contributions

The contributions of this work are:

- The creation of a set of antipatterns describing Java lock contention issues.
- The creation of a dataset of the antipatterns to be used in testing and training.
- A program that identifies the antipatterns in java source code.
- Validation of the antipatterns on open source data.

1.5 Organization

This thesis in split into the following chapters. Chapter two will review the related work around the topics of code patterns and anti patterns, as well as the topic of static code analysis. The review of antipatterns shows formats and uses of antipatterns that occur in other work and how they relate to Java lock contention problems. The review of static analysis covers multiple techniques and tools used to analyze code as well as advantages and disadvantages for each of them.

Chapter three will go over the methodology, the first item covered is the process used by the program to analyze Java source code, including detail on how it reads, matches, and provides recommendations. Additionally, there is an analysis of anti pattern formats with advantages and disadvantages for each, including how well they work in a lock contention context.

Next, there are definitions and example code for each of the lock contention antipatterns in this thesis as well as some background information. It starts by reviewing some other ways that antipatterns are used and stored and evaluates how well they apply to the context of lock contention and automation. Finally, there are detailed explanations of each of the code antipattern, talking about how each works, why it causes problems, how it may be detected in static analysis, and briefly discuss the code example.

In chapter four there is the validation of the work in chapter three, it starts by describing the second dataset and the test setup that is used. The first dataset is the dataset given as an example at the start of chapter 3, it is executed and checked to see if they return the correct patterns as expected by their definitions. For open source the Apache HBase GitHub repository was used, the results of its execution were manually verified to evaluate the correctness of the algorithm.

Chapter five is the conclusion which will give an overview of the results, threats to the validity of this work based on the approach, and what future work is planned in relation to this.

Chapter 2

RELATED WORK

Both lock contention and static analysis are fields that are currently being worked on, there are many tools designed to check code for errors and style violations. However, they are not designed to locate the types of problems that are specified in the list of antipatterns. Some of them are able to detect a subset of the anti-patterns in our list, but they are not the ones related most directly to lock contention.

We will discuss several methods of static and dynamic analysis currently used and give background on anti-patterns with some examples of their use.

2.1 ANTI-PATTERNS

Concurrent Bug Patterns and How to Test Them by Eitan Farchi et al.[6] presents a categorized taxonomy of concurrent bug patterns and then tested their patterns by creating heuristics for Con-Test[19] and evaluating their performance.

The paper initially presents the patterns by category: ”Code Assumed to Be Protected”, ”Interleavings Assumed Never to Occur”, and ”Blocking or Dead Thread Bug Pattern”. ”Code Assumed to Be Protected” is straight forward, the code should use a synchronization method but due to the programmers mistake it does not, the first example type named is ”Nonatomic Operations Assumed to Be Atomic”, it is also present in the antipattern list as Simultaneous Access.

Paper	Important point(s)
Concurrent Bug Patterns and How to Test Them[6]	Pattern categories ConTest tool
SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs[7]	Bug categories with related solutions Example detection tools
DECOR: A Method for the Specification and Detection of Code and Design Smells[8]	Code smell descriptions DECOR tool
Defining a Catalog of Programming Anti-Patterns for Concurrent Java[9]	Listing of antipatterns System of antipattern definitions
Recognizing Antipatterns and Analyzing their Effects on Software Maintainability[10]	Evaluating source code
An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension[11]	Impacts of antipatterns
HaLock: Hardware-assisted lock contention detection in multithreaded applications[12]	Dynamic analysis methods
Profiling and tracing support for Java applications[4]	Evaluation of JVM performance tools
Continuous and Efficient Lock Profiling for Java on Multicore Architectures[13]	”Free Lunch” profiler, critical section pressure Type 1/2 contention
Static analyses for eliminating unnecessary synchronization from java programs[14]	Static analysis tool that removes unnessisary conention
Comparing Four Static Analysis Tools for Java Concurrency Bugs[15]	Bug patterns vs antipatterns How to evaluate static classifiers Discussion of non-determinism
A security pattern detection framework for building more secure software[16]	Static pattern detection method
Detecting Anti-Patterns in Java EE Runtime System Model.[17]	Antipattern template format discussion
Falcon: Fault localization in concurrent programs[18]	pattern frequency statistics runtime behaviors

Table 2.1: Table summarising important points taken from related works.

”Interleavings Assumed Never to Occur” is a similar issue to ”Code Assumed to Be Protected”, however it refers to a different source of issue, it is mostly present as a race condition type issue and the first example type is The sleep() Bug Pattern where a parent thread waits a set amount of time for a child thread to return a value and there are issues with timing. This type of issue is not present in the types we defined types due to them focusing on static analysis of lock contention issues inside the critical section. The final type, ”Dead Thread Bug Pattern”, relates to situations where a thread will hold a lock indefinitely either by attempting to preform an operation that will never terminate or due to an unexpected and improperly handled exception. Again, this is not present in our set of code anti patterns as it has more to do with runtime issues and is not an issue that static analysis is well suited to.
The tool that is used for analysis, ConTest, is a dynamic analysis tool that works by instrumenting the critical sections in code and manipulates the behavior of the code to attempt to discover potential sources of contention. As a dynamic analysis tool its approach does not translate well to a static setting, however its categorization of bug patterns is useful for analysis purposes.

SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs by Mohammad Mejbah ul Alam et al.[7] outlines a set of categories fixes and strategies that relate to explicit synchronization primitives. The types are Improper Primitives, Improper Granularity, Over-Synchronization, Asymmetric Contention, and Load Imbalance. Improper Primitives and Improper Granularity are when the lock is not set up efficiently, Over-synchronization is synchronization of safe operations, Asymmetric Contention is where the parts of a divided lock do not have balanced traffic, and Load Imbalance is where a group of threads are contending more than others similar to Hot Section.
SyncPerf also presents two tools related to these types. The first is a detection tool that is very light and designed to run in a development setting, it is able to detect susceptible callsites and synchronization variables with potential performance issues as well as identify some root causes. The

second tool is used when the first is unable to tell what the root cause of a problem is and collects more data.

DECOR: A Method for the Specification and Detection of Code and Design Smells by N. Moha et al.[8] analyses several approaches to detect code smells and their limitations. To address those problems they propose DEtection & CORrection(DECOR) which contains all the steps necessary for the specification and detection of code and design smells. The correction component was left for future work. The detection component was named DETection EXpert (DETEX).
Descriptions of code smells in related work are discussed in section 2.1, however these mostly use text-based descriptions. Next a series of detection methods were described the first used manual inspection of the code so it does not scale well, next a metric based approach using the IPLASMA tool was described, it extracted metrics that were applied to a series of thresholds to match to smells. There are several other approaches that use their own language, or significantly involve a human in their execution.
DETEX detects code and design smells by this process: Step 1 is to extract a vocabulary from design pattern descriptions, step 2 combines the vocabulary with several other features that describe the code smells, step 3 translates the descriptions into an algorithm form, step 4 is detection that uses algorithms from step 3, and step 5 is validation that uses manual verification to check results.
For the application proposed in this thesis the DETEX process is somewhat applicable, a vocabulary-based approach would not work due to the similarity of many operations in a synchronized setting as well as the importance of the order in which operations occur, however some of the other data that they collect in the second step such as structural and behavioural metrics could be used for this classification.

Defining a Catalog of Programming Anti-Patterns for Concurrent Java by Jeremy S. Bradbury et al.[9] organizes and defines a set of low-level antipatterns for use with concurrent Java. It first provides the format for how the antipatterns will be laid out: the pattern name, the problem, the

context, and the solution. Pattern name, problem, and solution are straightforward definitions, context describes how the problem occurs from a programmer perspective.
The set of antipatterns that this paper provides are: Nonatomic operations assumed to be atomic, Two-state access bug(synchronized operations are not necessarily in the right order), Wrong lock or no lock bug, Doublechecked lock, Sleep()(manually delaying a thread to try to help with synchronization and race conditions), Missing or nonexistent signals, Notify instead of notify all, A “blocking” critical section(a lock is never released), Interference, Deadlock, Starvation, Resource exhaustion, And Incorrect count initialization.
These antipatterns a good summary of types of contention issues, however they mainly focus on functional issues, rather than performance problems.//

Recognizing Antipatterns and Analyzing their Effects on Software Maintainability by Denes Ban et al.[10] works to evaluate the relationships between antipatterns and software maintainability, number of bugs, and future faults. It starts by defining patterns as recommended solutions to common design problems according to their situation and antipatterns as common solutions to a problem that produce negative outcomes. Additionally, it states that an antipattern must have a solution that fixes the problem while maintaining the function of the code.
Their approach involved using a machine learning system to use a set of metrics generated from a LIM model (Language Independent Model) to test connections between the presence of their listed antipatterns and software maintainability, number of bugs, and future faults. The nine antipatterns they examined are: Feature Envy, Lazy Class, Large Class Code, Large Class Data, Long Function, Long Parameter List, Refused Bequest, Shotgun Surgery, and Temporary Field. They found a positive correlation between the number of bugs and the number of antipatterns, a negative correlation between the number of antipatterns and maintainability, and it was able to perform almost as well as several other methods.
The patterns presented in this paper are mostly not relevant to the field of concurrent programming, however the method used to evaluate the source code is related to the approach we used.

An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension by Marwen Abbes et al.[11] seeks to detect the correlation between program readability and the presence of antipatterns. They looked at two antipatterns "Blob" a single large class the holds a significant portion of a system's functionality and only uses other classes as data holders, and Spaghetti Code where classes are not properly structured, methods are badly named and use few parameters, and global variables are used heavily.
The testing was done on Java based systems with combinations of Blob, Spaghetti Code, and no antipatterns by a group of graduate students and professional developers. It found that the presence of the a single antipattern was not a major problem for the participants, however both antipatterns resulted in a significant increase in difficulty.
While this paper does not connect directly to the problems addressed in this thesis, it does help to illustrate what types of problems other antipatterns can cause to developers.

2.2 Dynamic Analysis

HaLock: Hardware-assisted lock contention detection in multithreaded applications bs Yongbing Huang et al.[12] presents a hardware-based lock profiling tool that works by observing memory. Due to its use of a memory tracing tool, it has minimal overhead.
This paper establishes two parts of dynamic analysis, the first is observing a system without interfering in its operations, and the second is recording the data so that it can later be analyzed. The described ways of observing a system are instrumentation, performance counting, and sampling. Instrumentation is the act of modifying the original code to produce outputs that can be monitored to evaluate the performance of the system. Performance counting is a system like JLM that monitors the actions of the program to produce certain values, such as operation types and time information. Sampling is an alternative to these two methods that attempts to reduce overhead by only collecting

data at intervals, this has the advantage of less execution time and less data written, but also has the possibility of missing short critical sections.
Once this data is gathered it can be analyzed through manual or automatic systems, however they would all be examined outside of the operation of the original test environment in order not to interfere.
HaLock has good accuracy and scalability, it also quickly reads read performance counters and eliminates clock inconsistencies due to the additionally hardware.

Profiling and tracing support for Java applications by Andy Nisbet et al.[4] tests and evaluates JVM with different tools for performance evaluation. This paper focuses on sampling-based tools, and the limitations they have due to the system they use to acquire sample points. Sampling profilers work by recording the call stack of functions and methods that should represent the state of the program at that point in time.
Thread execution time for methods is calculated by the number of times the method appears in stack traces, this relies on the sampling points being fully random and being collected fairly. The problem is that many approaches are reliant on restricting sampling to safe points that are inserted by the JIT compiler to support GC which introduces bias.

Continuous and Efficient Lock Profiling for Java on Multicore Architectures by Florian David et al.[13] describes a profiler called free lunch that uses a metric called critical section pressure using a modified Java Virtual Machine Tool Interface (JVMTI) and thread data. This paper also established a distinction between lock contention types at runtime: type one where the contention is a result of a large slowdown, and type two where the contention is a result of multiple smaller threads accessing a locked region too quickly. This work applies the types to static analysis as well: type one is where the problem originates from a single lock object, and type two is where the problem lies outside of the critical section. Due to this distinction, the code examples provided in this paper are primarily type one.

2.3 Static Analysis

Static analyses for eliminating unnecessary synchronization from java programs by Jonathan Aldrich et al.[14] presents and evaluates static analyses that help to reduce synchronization overhead by detecting and removing unnecessary synchronization. The situations that it looks for are where there is no contention possible during the task, or there are relationships between monitors that mean some may not be required.
The optimizations that this paper discusses are split into three parts reentrant monitors, enclosed monitors, and thread local monitors. Reentrant monitors are where a thread attempts to acquire a lock that it already possesses, the optimization is to create multiple versions of the same code but only one is explicitly synchronized, while the other is only able to be accessed from the first.
An enclosed monitor is a dependency between monitors that allows one to rely on the other's synchronization, this is only possible if it can be proven that the dependency is always true, if there is another situation where the second can be accessed or another monitor that can take the place of the first in the dependency order then the optimization is not possible.
Thread local monitors are where a thread creates a class that can only be accessed by itself, in this situation the synchronization does nothing, as the is only one possible thread that can enter.
The enclosed monitor optimization is a possible element that could be added to the recommender created for this thesis in the future as it focuses only on critical sections and their structure.

Comparing Four Static Analysis Tools for Java Concurrency Bugs by A Mamun et al.[15] is an evaluation of 2 open source and 2 commercial static analysis tools with regards to how well they can detect bugs in multithreaded code. The goal is to determine whether commercial or open source is better and how well static analysis performs in general.
During the paper they discuss non-determinism and why it makes it static analysis particularly difficult for multithreaded code due to interactions between threads that are decided at runtime.

Additionally multithreaded programs have an additional set of problems on top of what other programs can. This paper splits the problems by two properties: safety and liveness, however states that they are not helpful for classification. The paper also makes a distinction between bug patterns and antipatterns. Bug patterns are common errors that can occur in the program, while anti patterns are mistakes that are made while designing the program. However, in the context of concurrent software testing, bug patterns and antipatterns are used interchangeably.
The data used for testing came from two sources: a set representing concurrency bugs and the second set representing concurrency bug patterns. The first came from a concurrency bug benchmark suite, and the second set is a collection of Java concurrency bug patterns and antipatterns which they collected or wrote programs for, each one is 10 to 30 lines long. They identified 87 unique bug patterns from 141 bug patterns.
The end result was that the commercial classifier Jlint was most effective with a 0.47 detection ratio, however the other commercial classifier used was ranked last with a ratio of 0.13 so there is no conclusion on commercial vs open-source classifiers. Additionally, the average performance was 0.25 and each one produced a false positive rate approximately equal to the number of true positives, so they state that static analysis alone is not sufficient to properly detect concurrency bugs

A security pattern detection framework for building more secure software Aleem Khalid Alvi et al.[16] presents a framework for detecting security patterns. Security patterns are the solution to security design problems or anti-patterns. The framework consists of three parts: class relationship discovery, security pattern matching, semantic analysis, and detection report.
The class relationship discovery takes, a software system model and an empty class relationship matrix as input and uses the software model to detect and record the interactions between the classes in the relationship matrix. The relationship matrix contains values of one or zero showing whether or not the classes are connected.
Security pattern matching uses the filled class relationship from the first step and a set of security

pattern matrices that correspond to the set of known patterns, it will then return a list of patterns which match with the original relationship matrix.

Semantic analysis uses dictionary data extracted from documentation and system requirements to reduce the number of possible matches to make them more useful.

The detection report returns the information for the previous step in a human readable format and includes information about the identified pattern(s) and important locations in the code.

For testing three open-source Java-based systems were used: Simple Android Instant Messaging (SAIM) client (Mermerkaya, 2013), the Automated Teller Machine Simulator (ATM sim.) (Bjork, 2019), and the Electronic Voting System (VoteBox).

Security patterns are always present in any security application and there are catalogs available that provide details of security patterns using standard security pattern templates.

Detecting Anti-Patterns in Java EE Runtime System Model Lei Zhang et al.[17] proposes an approach to classification of java antipatterns based on a Java EE meta-model that uses QVT (Query/View/Transformation) for the detection process.

In this work antipatterns are represented as a template that contains the name of the antipattern, what runtime elements are needed to detect it, a text description of the pattern, and the QVT scripts to identify and locate the elements in code.

The approach creates a runtime model based on the actual code using SM@RT, then classifies that abstract model based on the execution of the QVT script.

This work also points out that although there is significant work and documentation on anti-patterns they are usually represented graphically or textually using informal language and, in the case of Java EE anti-patterns, they are usually designed with one specific system in mind. This makes classification difficult to fully automate, due to the lack of an appropriate dataset. This issue is shared by out setup as well, as readily available datasets are not available for all types of antipatterns.

This model is limited by the QVT language and is unable to detect any pattern that cannot be represented in it. These patterns include those that are “highly tangled in source code”, and ones that are not well suited to the language, which includes some contention issues such as Excessive Dynamic Allocation and Spin Wait.

Falcon: Fault localization in concurrent programs by Sangmin Park et al.[18] presents a dynamic fault localization technique that locates problematic data access patterns in concurrent programs by examining data gathered from code instrumentation. The actual process used is not relevant as it is dynamic. However, the background information provided is quite helpful.
The first point they make is that concurrency faults are hard to find because they usually only appear under specific situations and are dependant on runtime behavior and thread parameters which are somewhat random. Because of this many fault detection tools will detect patterns without being able to distinguish problematic patterns from benign ones, this is also true in the case of lock contention, where the degree of the convention is dependant on runtime factors and therefore problematic patterns found in code are not necessarily bad.
Additionally, the paper provides some statistics on concurrency usage from 2007 from Microsoft, one of the statistics they report is that nearly two-thirds of the developers and testers reported that they had to deal with concurrency issues and over half of those had to work on debugging concurrency issues at least once a month.
Finally, the paper cites that these types of faults are most commonly data races(multiple threads accessing improperly protected resources), atomicity violations(operations treated as atomic that are not), and order violations(order dependant tasks improperly handled), and that they are also ranked as the most common and difficult source of concurrency faults

Chapter 3

METHODOLOGY

The beginning of this work was a list of lock contention anti-patterns. Once that was constructed and examples were created, then the classifier was made to locate instances of those anti-patterns in Java code and which was then tested against real world examples to verify the anti-pattern list. The static classifier is designed to work together with a larger system that will include dynamic analysis to provide more context for the recommendation.

Limits of Static Analysis

Static analysis is effective at determining the source of a problem in code without running it and it is generally faster and more transferable than dynamic analysis, however it does have limitations. The first is that without running code it is impossible to tell the actual result of that code. Additionally, in the case of concurrent programming, more complex interactions between critical sections are hard to spot and external factors cannot be accurately accounted for.

For this reason, it is impossible to say without runtime data whether or not a contention anti pattern will result in an actual performance problem.

3.1 Anti-pattern Background Information

Definition Format

The antipatterns defined will have several components, the first is a text description of the problem, where indicators of the antipattern may be located in code, and what might fix the problem, next there is a code example of the antipattern that will help to illustrate the problem and how it occurs. Additionally, there is a highlighted Abstract Syntax Tree to help show what signs might point to specific antipatterns and how they manifest in an AST.

Comparison With Other Antipattern Formats

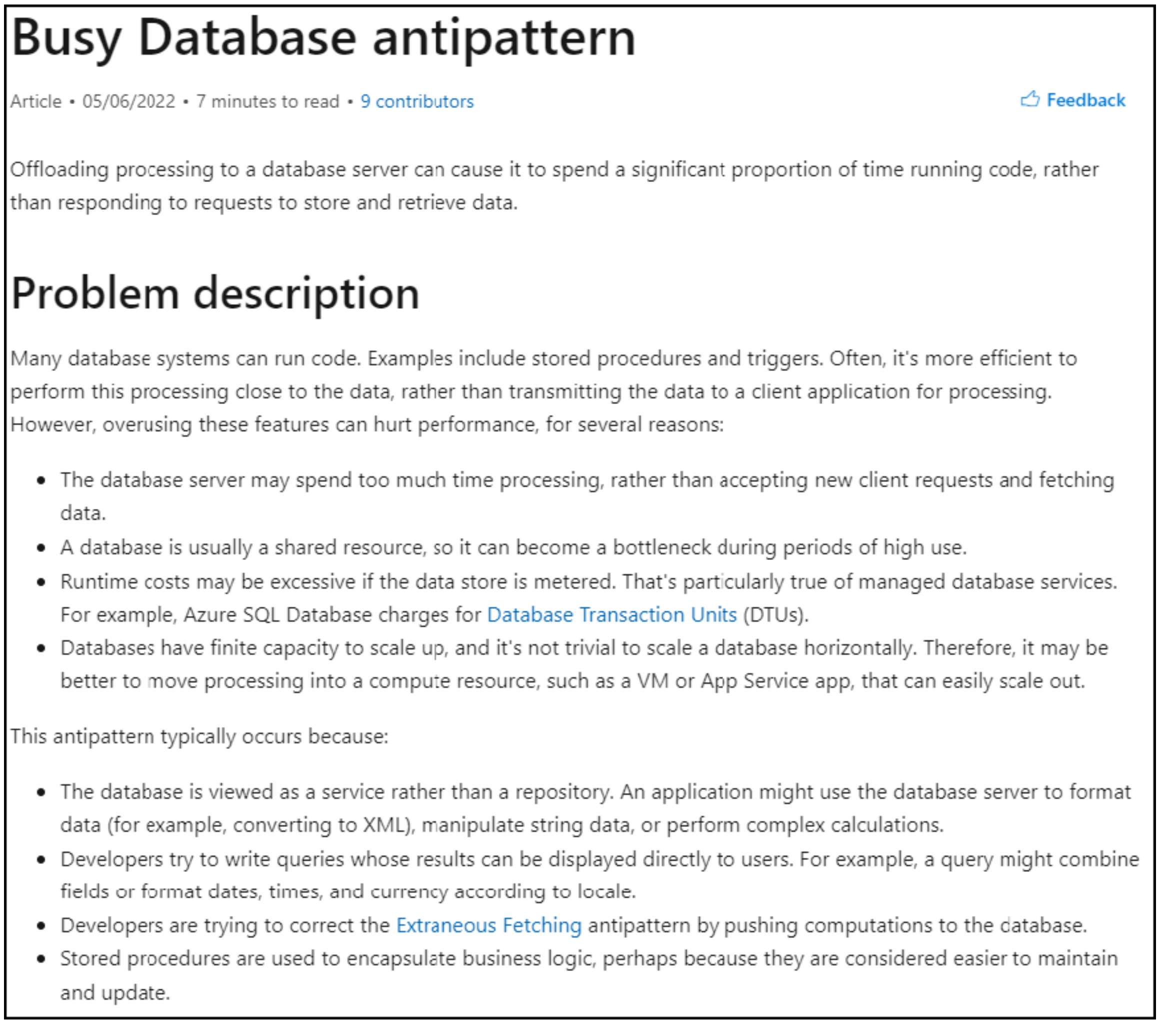
To decide how to represent antipatterns we went through several possible options, to find the most appropriate form to record and explain them.

Plain Text and Examples

The most classic and widely used form is to use plaintext descriptions and possibly code examples, this has been used for patterns and antipatterns since they were first formally created for use in computer engineering. This description could be formal or informal, and software design textbooks will usually use this format to explain in detail what a pattern is and how it may be used.

The text component of the description should cover what the antipattern is, how it is created, what effects it may have, and the antipattern's name. In figure 3.1 is included an example from Microsoft's Azure performance antipattern page "Busy Database antipattern"[20] to show what one of these might look like The code example may be included to illustrate how a pattern is used or what situation it may be used in in code, these can be as actual code or pseudocode. The example from f igure 3.1 also contains a code example on its web page to support their definition.

This format is used because it allows a human to clearly understand the definition, however it is not suitable for an automated application, even if the pattern is represented formally. Therefore, it



Busy Database antipattern

Article • 05/06/2022 • 7 minutes to read • 9 contributors

Feedback

Offloading processing to a database server can cause it to spend a significant proportion of time running code, rather than responding to requests to store and retrieve data.

Problem description

Many database systems can run code. Examples include stored procedures and triggers. Often, it's more efficient to perform this processing close to the data, rather than transmitting the data to a client application for processing. However, overusing these features can hurt performance, for several reasons:

- The database server may spend too much time processing, rather than accepting new client requests and fetching data.
- A database is usually a shared resource, so it can become a bottleneck during periods of high use.
- Runtime costs may be excessive if the data store is metered. That's particularly true of managed database services. For example, Azure SQL Database charges for Database Transaction Units (DTUs).
- Databases have finite capacity to scale up, and it's not trivial to scale a database horizontally. Therefore, it may be better to move processing into a compute resource, such as a VM or App Service app, that can easily scale out.

This antipattern typically occurs because:

- The database is viewed as a service rather than a repository. An application might use the database server to format data (for example, converting to XML), manipulate string data, or perform complex calculations.
- Developers try to write queries whose results can be displayed directly to users. For example, a query might combine fields or format dates, times, and currency according to locale.
- Developers are trying to correct the Extraneous Fetching antipattern by pushing computations to the database.
- Stored procedures are used to encapsulate business logic, perhaps because they are considered easier to maintain and update.

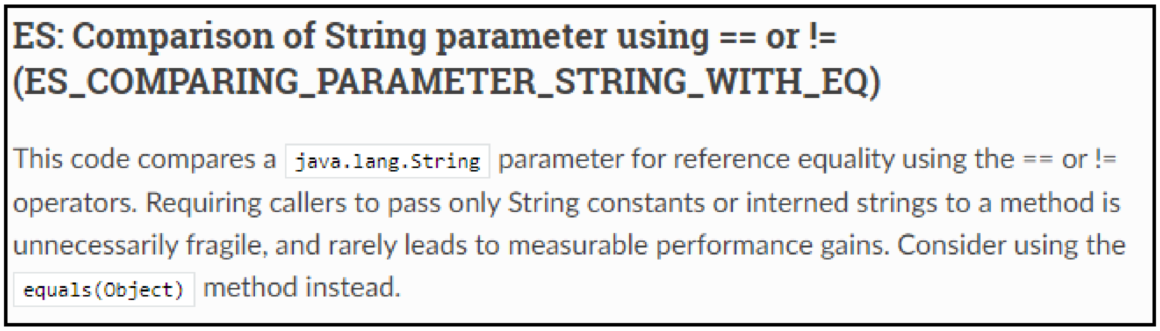
Figure 3.1: Antipattern definition ”Busy Database antipattern” from Microsoft Azure documentation[20]

is not the only format that is needed.

User Interaction Model

A user interaction model is a way of representing antipatterns that is only readable by a computer, it works by providing an initial description of the antipattern, and having a user provide feedback to classify code examples as positive or negative. the feedback is then fed into the algorithm along with the other data from the rest of the patterns in the dataset and to match to each of the starting set of antipatterns.

An example of this is SpotBugs, a Java static analysis tool for finding bugs, that uses a set of over 400 bug patterns. Each pattern has a name associated with it and a short description. An example description is in figure 3.2. This is the only description of the antipatterns provided. Additionally, it is possible to add new detectors to an instance of SpotBugs, however as this is a User Interaction model based tool adding patterns and antipatterns to it is done by adding good and bad examples to the system and letting the classifier create the detector.


ES: Comparison of String parameter using == or != (ES_COMPARING_PARAMETER_STRING_WITH_EQ)

This code compares a `java.lang.String` parameter for reference equality using the == or != operators. Requiring callers to pass only String constants or interned strings to a method is unnecessarily fragile, and rarely leads to measurable performance gains. Consider using the `equals(Object)` method instead.

Figure 3.2: Example antipattern (bug pattern) description from SpotBugs documentation.[5]

The detector did not work well for the examples tested from the example dataset, this may be due to static analysis not being able to fully differentiate between contention scenarios by itself.

This model was not used due to this limitation.

Structure based AST

An Abstract Syntax Tree(AST) is a way of representing code that eliminates some inconsistency in code and allows an observer to capture the structure and meaning of the code.

An example AST of one of the sample codes is shown if figure 3.3. This AST does not capture a full program, but rather a single critical section extracted from a program, this was done to focus on the relevant parts.

Analysis of this would then work by comparing the structure of the tree and values of nodes to determine similarity to other trees, however the problem that arose during investigation was that many ASTs generated from java code have a very similar structure, especially in the case where they are all generated from synchronized regions. A second AST example is shown in figure 3.4. Each tree shares the features of mostly extending to the left of the root node and sharing a similar number of nodes at each level. This problem is only increased by the analysis of open source code that shows that many critical sections are mostly short, meaning there is not much of a structure to analyse.

To fix this issue the primary focus was shifted to the values of nodes, and only on some structural elements that may exist in code samples.

3.2 Anti-pattern definitions

Hot Section 1 Antipattern

Description

A Hot Section is a situation where one critical section is the cause of a significant portion of the contention issues in a system, there are multiple ways it can be created. The first way that a Hot Section can be created is that one critical section is causing the contention issues by itself. This can be due to either the process in the critical section being very large and taking the thread a significant amount of time, or a large amount of traffic through the lock which will create a problem regardless of the length of the process.

Detection/solution notes

As this is a problem that relies on the entire critical section there is no one sign that there is a problem, making Hot Section type 1 a general category rather than a situation with a specific solution. Despite there being no specific solution to this problem there are some solutions that can be applied to this type of problem, they are to reduce the time needed for the task inside the lock to complete

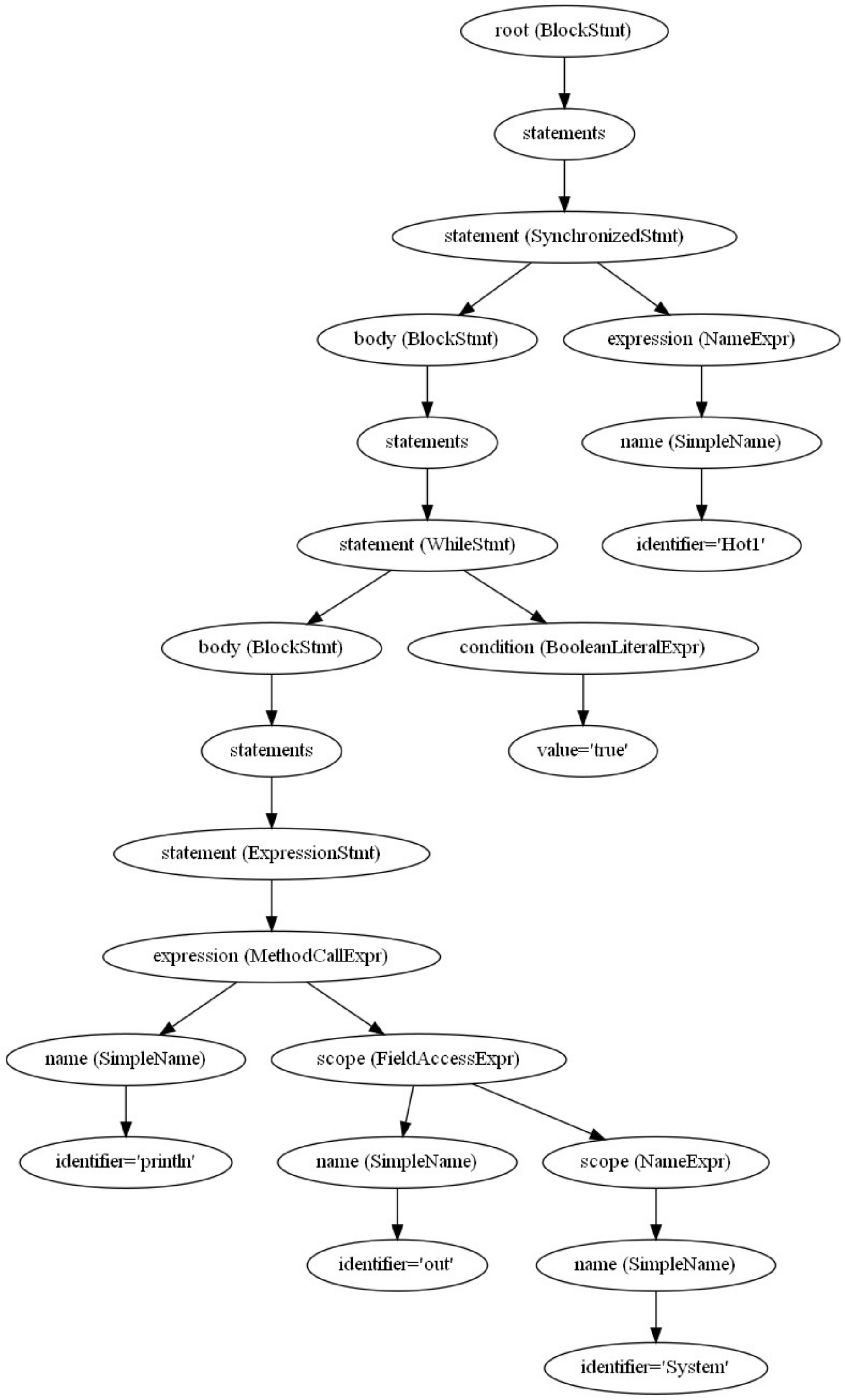


Figure 3.3: AST of antipattern generated from Hot1

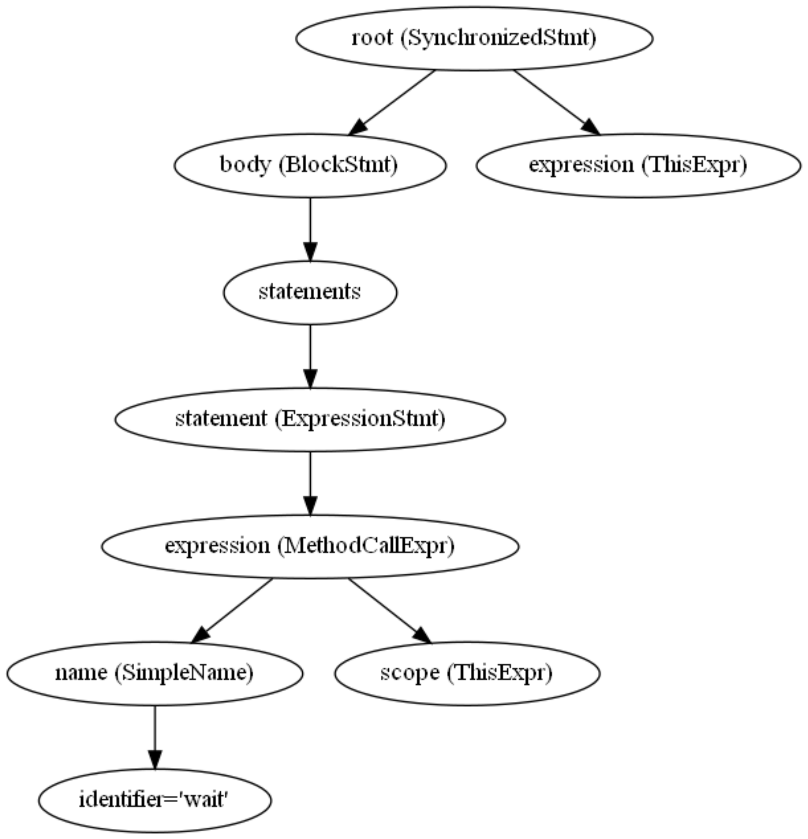


Figure 3.4: AST generated from open source code taken from Apache Hbase

or to break apart the critical section where possible and non synchronized operations.

If the problem comes from too many attempts to access the critical section, a frequent access problem, then the solution would need to change the code around the critical section and other components because the source of the problem is outside of the critical section, the first solution would be to switch the synchronized object to a different style to allow it to have multiple users at once and split the required throughput if possible. If that is not an option, then you need to either significantly speed up the critical section or reduce the number of calls made to it.

Due to these lock contention antipatterns being designed for a static context, they are not directly tied to performance metrics. The same code could manifest different runtime behavior depending

Type	Problem Signs	General Solutions
Hot Section 1	Significant slow from one critical section Function calls/network calls inside critical section Long tasks	Lock splitting Lock stripping
Hot Section 2	Significant slow from one lock monitor. Many critical sections use the same lock object.	Change in synchronization method Remove/speed up bottleneck Critical section
Hot Section 3_1	Frequent acquires to one critical section Loop immediately outside relevant critical section	Lock stripping
Hot Section 3_2	Significant slow from critical section Loop inside relevant critical section	Reduce lock granularity
Unpredictable	Unreliable/unexpected performance data	Correct improper definitions
Simultaneous	Unexpected contention or data modification	Fix synchronization
Overly Split	Significant slow from one lock monitor primarily from one method Short synchronized regions	Reduce lock granularity

Table 3.1: Table Summary of Antipattern types.

on what the rest of the program is doing. The antipattern only describes a possible source of contention and it is only problematic once the frequency or duration of the task becomes too much, the exact amount that describes “too much” depends on the user.

Hot Section 2 Antipattern

Description

The second type of Hot Section is where the one synchronized object is being accessed throughout the program. Similar to the frequent access portion of Hot Section 1, it is harder to pinpoint the issue here due to the problem not occurring inside an individual critical section as it has to do with the behavior of the program, or with a lock variable that is used in multiple places that combine to create the problem.

Listing 1 Hot Section One Code Example

```
class Hot1{
  Object Hot1;
  public void doSomething(){
    synchronized(Hot1){
      while(true){
        System.out.println(Hot1);
      }
    }
  }
}
```

Listing 2 Hot Section One Code Example With Solution

```
class Hot1{
  Object Hot1;
  public void doSomething(){
    while(true){
      Object A;
      synchronized(Hot1){
        A=Hot1
      }
      System.out.println(A);
    }
  }
}
```

Detection/solution notes

Due to the described behavior this can be detected either through repeated use of the lock object, or by examining the structure directly around the synchronized region.

However, this may not be possible, if it is not then the solutions that remain are what was used in the previous section of modifying the critical section(s) it takes less time or reducing the overall number of calls made to significant critical sections.

As this is not a problem due to one section, the solution also works differently, the most straightforward option is to split the lock into multiple smaller ones, either by changing to logic or by using a data type that allows concurrent access.

This situation is not necessarily a problem, as many locks using the same object may be part of the system design. However, if there are enough locks spread across the program then they can cause a overall slowdown, additionally if one critical section holds the lock for a long time, then it can affect the rest of the program.

In Listing 3 there is an example of Hot Section 2 where the object “Hot2” is used in both critical sections. In the example solution, Listing 4, one of the critical sections is removed due to the object type being changed to an atomic type to reduce the time spent in critical sections across the program.

Listing 3 Hot Section Two Code Example

```java
public class Hot2{
  int Hot2;
  public void task1(){
    synchronized(Hot2){
      System.out.println(Hot2.toString());
    }
  }
  public void task2(){
    synchronized(Hot2){
      Hot2+=1;
    }
  }
}
```

Hot Section 3 Antipattern

Description

The third type of contention is caused by loops in or around critical sections.

This type of problem will look very similar to Hot Section-Type 1 at runtime depending on the positioning and functions of the loops. Determining if this would be a problem with purely static

Listing 4 Hot Section Two Code Example Fixed

```
public class Hot2{
  AtomicInteger Hot2;
  public void task1(){
    System.out.println(Hot2.get());
  }
  public void task2(){
    synchronized(Hot2){
      Hot2.set(Hot2.get()+1);
    }
  }
}
```

analysis may be impossible due to variables only set during runtime. However, given that this assumes that the system already knows that there is contention occurring, static analysis should be sufficient to find the problem. To fix the problem would be done in two general methods depending on how the loops are constructed and placed, the solutions are similar to Hot Section-Type 1.

Detection/solution notes

Listing 5, class Hot3_1, has the loop outside of the critical section which causes the thread to attempt to enter the critical section quickly after exiting, this should result in a frequent access problem due to not changing the length of the task but increasing the number of acquires and percentage of time spent with the lock held. Therefore, detecting it would involve looking for loops or equivalent behavior close to the beginning of the critical section, if there is a critical section inside of a loop it does not necessarily mean that it is this type of problem as the other tasks inside the loop may reduce the effect to a manageable level.

In Hot3_2 there is a loop inside the critical section which multiplies the execution time of the operation inside the critical section, in that case the solutions from Hot Section-Type 1 should apply, as this problem is a subtype of Hot Section Type 1. In Hot1 The code example is a loop inside a critical section that extends the time spent inside the critical section, detecting this would involve looking for loops or recursive structures inside critical section and moving as much as is possible outside of

Listing 5 Hot Section Three One Code Example

```
public class Hot3_1{
  Object Hot3_1;
  public void doSomething(){
    Boolean x=true;
    while(x){
      synchronized(Hot3_1){
        x=System.currentTimeMillis()==123456789;
      }
    }
  }
}
```

Listing 6 Hot Section Three One Code Example fixed

```
public class Hot3_1{
  Object Hot3_1;
  public void doSomething(){
    int time;
    do{
      synchronized(Hot3_1){
        time=System.currentTimeMillis();
      }
    }while(time==123456789);
  }
}
```

the critical section to create the non problematic state of Hot Section Type 3_2, additionally Hot3_1 becomes a problem depending on how much of the loop is spent inside the critical section as it significantly increases the hold time of the lock.

Listing 7 Hot Section Three Two Code Example

```
public class Hot3_2{
  Object Hot3_2;
  public int doSomething(ArrayList A){
    synchronized(Hot3_2){
      int x=0;
      while(A.get(x)!=15){
        x++;
      }
      return x;
    }
  }
}
```

Listing 8 Hot Section Three Two Code Example fixed

```
public class Hot3_2{
  Object Hot3_2;
  public int doSomething(ArrayList A){
    int x=0;
    synchronized(Hot3_2){
      value =Hot3_2.value();
    }
    while(A.get(x)!=value){
      x++;
      synchronized(Hot3_2){
        value =Hot3_2.value();
      }
    }
    return x;
  }
}
```

Unpredictable Outcome/Bad Form Antipattern

Description

Unpredictable outcome or bad form describes a situation where the performance is negatively affected by the code not properly handling synchronization in a way that causes locks and makes synchronized data interfere with other data unintentionally. This type of issue will usually result in strange unexplained behavior for both functional and performance issues.

The specific case used here is based on improper definitions for synchronized objects, in the example the monitor object is defined as an object that exists as part of the Boolean class, this can cause issues with contention and is dependant on the compiler.

Detection/solution notes

Identifying this type of antipattern involves examining monitor definitions, or points in the code where the monitor value is changed, which should mean examining the critical sections. Those operations could then be compared against a list of known problematic definitions to identify potential issues.

The actual runtime behavior of this antipattern could be anything, as there are many possible forms of this issue. Additionally, each problem will only appear occasionally in a runtime environment, and the location of the problem in code is far from where problems would appear.

Simultaneous Access Antipattern

Description

Simultaneous access is where the code accidentally allows a variable to be accessed without the correct protection or permission, such as outside of its critical section. This can be done either by having the object accessed outside of a critical section by accident, or by having two objects interact with each other in an unexpected way.

Detection/solution notes

Listing 9 Bad Form Code Example

```
public class Unpredictable{
  private final Boolean Unpredictable = Boolean.FALSE;
  public void doSomething(){
    int x=0;
    synchronized(Unpredictable){
      if(Unpredictable){
        x+=1;
      }
    }
  }
}
```

Listing 10 Bad Form Code Example fixed

```
public class Unpredictable{
  private final Boolean Unpredictable = false;
  public void doSomething(){
    int x=0;
    synchronized(Unpredictable){
      if(Unpredictable){
        x+=1;
      }
    }
  }
}
```

This problem is likely to happen occasionally on accident and checking for the pattern is time consuming for a human. The first step of identifying the problem is to construct a list of all know synchronized objects, this can be done by checking each critical section and extracting the synchronized objects used. Next the project would need to be checked for any violation of the normal synchronization which is not necessarily straightforward. In an ideal case each objects synchronized block starts within the method it is used in, however as a method can be called from within a critical section it is possible to have regions synchronized in a way that is not immediately visible, this means that you must trace where in the program each instance of the object can come from, up

Listing 11 Improper Access Code Example

```
public class Simultaneous{
  Object SimultaneousA,SimultaneousB,SimultaneousC;
  public void doSomething(){
    synchronized(SimultaneousA){
      synchronized(SimultaneousB){
        SimultaneousC=SimultaneousA+SimultaneousB;
      }
    }
  }
}
```

Listing 12 Improper Access Code Example

```
public class Simultaneous{
  Object SimultaneousA,SimultaneousB,SimultaneousC;
  public void doSomething(){
    synchronized(SimultaneousA){
      synchronized(SimultaneousB){
        synchronized(SimultaneousC ){
          SimultaneousC=SimultaneousA+SimultaneousB;
        }
      }
    }
  }
}
```

to entering the relevant critical section, to determine if it is properly synchronized. Also, due to how this issue is reached it does not generate contention, but is a related type that can be reached while attempting to fix contention issues. If an instance of the antipattern is found it is simple to fix by adding a critical section around the relevant lines.

Overly Split Locks Antipattern

Description

Overly split locks means that the locks have been significantly divided, perhaps in an attempt to resolve a hot section, to the extent that they begin to cause a problem for performance. The delay could be due to the overhead of entering and exiting a lock rather than the waiting or executing time of the lock itself. This is an instance of a frequent access problem, it was included due to its strong relation to the Hot Section antipatterns.

Detection/solution notes

Detecting this antipattern involves looking for multiple instances of critical sections with the same lock object in the same code block, additionally if there is little or no code in between the blocks then it might be beneficial to combine adjacent blocks to reduce the number of acquires to the lock. Therefore, this antipattern, in general, may be resolved by lock merging or switching the synchronization method if that is not possible.

The code example given is where the critical section is split as far as it can be, an ideal situation is a balance between one large Hot Section and many small Overly Split sections.

While attempting to find that balance it is very similar in behavior to Hot Section 2, there is a point where each task is small enough to function well, but not too split that it causes other issues, however in this case it can also produce functional issues.

3.3 APPROACH

The first stop was to decide what types of anti-patterns it should use. This work started from a dynamic analysis approach similar to Brian Goetz[21], among others, and stated that lock contention fits into two classes: type 1 where the lock is held by a thread that spend too much time inside a critical section and type 2 where a lock is held by many smaller lock over a short period of time, and a problem arises from the frequency of the lock acquisition.

When comparing this to static analysis a type one contention problem is most likely related to a problem that occurs inside the critical section, due to it being based on the time the lock is held for, and a type two problem is related to a problem that occurs outside of the critical section, due to it being based on the behavior of the program, and not reliant on execution time.

Listing 13 Significantly Overly Split Code Example

```
public class OverlySplit_1{
  Object OverlySplit_1A,OverlySplit_1B,OverlySplit_1C;
  public void doSomething(){
    int x=0;
    String name;
    synchronized(OverlySplit_1A){
      OverlySplit_1=new Scanner(System.in);
    }
    System.out.println("Enter username");

    synchronized(OverlySplit_1A){
      name = myObj.nextLine();
    }
    synchronized(OverlySplit_1A){
      name=null;
    }
  }
}
```

It is also worth noting that type one contention is dominant as the frequency of acquisitions is limited by the duration a lock is held for, as a new acquire cannot happen until the previous one has released the lock.

For these reasons this work focuses primarily on type 1 contention, contention that originates inside the critical section, and problems that can occur together with those. Due to this all of the antipatterns have sources inside the critical section and are able to be represented simply.

Patterns need to be defined in such a way that they can be properly sorted and split into types. That lets helps to create a system to categorize inputs and use categories for recommendations.

Recommendations work based on an analysis of each of the antipatterns. Each antipattern has a set of recommendations in table 3.1 that forms the basis for the final recommendations. Each of the recommendations comes from recommended solutions from resources about reducing or fixing lock contention issues. Once the program has created the list of what antipatterns a section of code matches to, it will be able to first output a list of recommendations in order, according to what antipattern is most narrow up to the most general.

3.4 Reading and Classification Process

This section will review the process used by the program to classify critical sections into the antipattern classes. The program initially reads through the input files and extracts all of the critical sections, if a class and method is input as the location of contention, then it will proceed only with synchronized objects that appear inside of that class and method, this may include synchronized regions outside of the requested method if the objects are shared. The process is shown in figure 3.5.

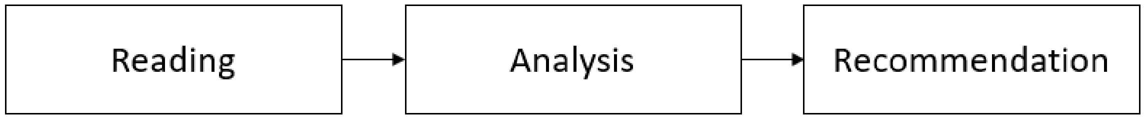


Figure 3.5: Process used for classification

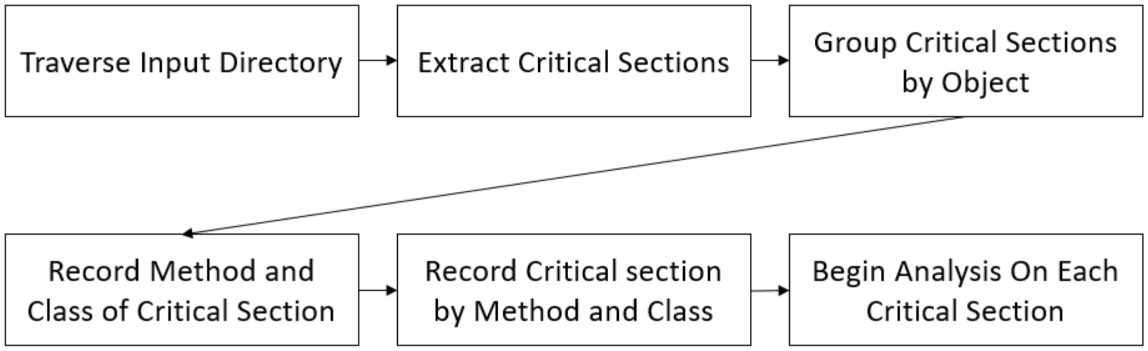

Traverse Input Directory
Extract Critical Sections
Group Critical Sections by Object
Record Method and Class of Critical Section
Record Critical section by Method and Class
Begin Analysis On Each Critical Section

Figure 3.6: Detailed reading process

3.4.1 Reading Files/Parsing

In the first step, the program reads through the input directory and its sub directories it finds the java files, shown in figure 3.6 and shown as pseudo code in listing 14.

Listing 14 Algorithm for reading Java code.

```
def read()
    readFiles(inputDirectory)
    locate synchronized regions in returned structured list
        record name and location as strings
        record critical section as reference to structured
         ↪  list
def readFiles(directory):
    instantiate structured list
    for each folder in directory:
        readFiles(folder)
        add returned structured list to this structured list
         ↪  in subfolder
    for each javaFile in directory:
        add java file and file name to structured list
            java file is stored as AST
    return structured list
```

It then parses each file into an AST using the JavaParser library[22] that will be stored for the duration of the program and used later. Next the program will search through the ASTs to find each critical section these will also be stored and grouped according to the lock object, and the method and class where each critical section occurs will be recorded.

3.4.2 Analysis/Pattern Matching

Next for each of the lock objects the list of critical section ASTs is fed into the analysis step, shown in figure 3.7 and displayed as pseudo code in listing 15, the first step is to examine each of the ASTs to find method calls, each of those will be replaced with the contents of their method where possible, this allows the program to get a better idea of what the program may do at runtime, if the method cannot be found it may be from a source that does not have java files inside the input directory such as a library, in this case it is left as a function call. Modified ASTs are stored as a copy of the original, the original is not changed.

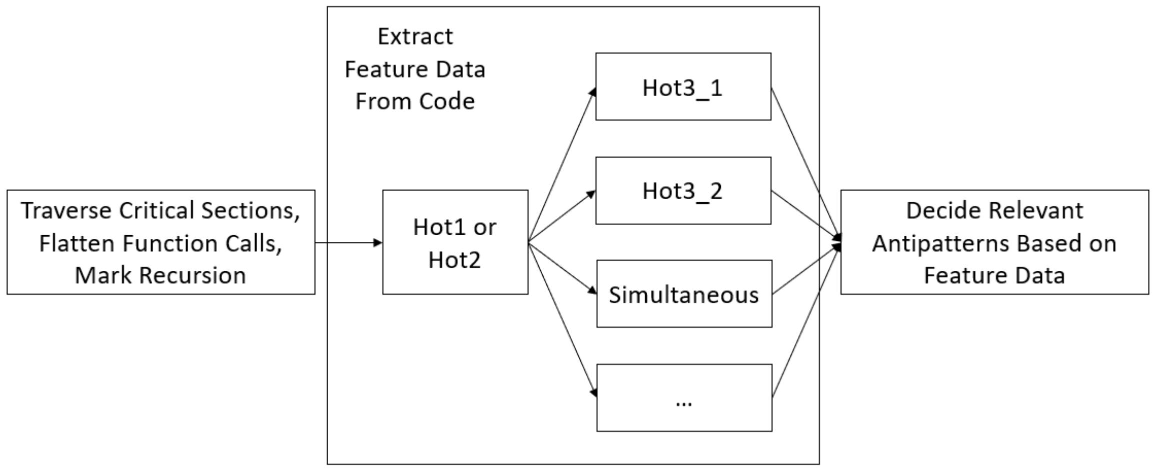

Extract Feature Data From Code
Traverse Critical Sections, Flatten Function Calls, Mark Recursion
Hot1 or Hot2
Hot3_1
Hot3_2
Simultaneous
...
Decide Relevant Antipatterns Based on Feature Data

Figure 3.7: Detailed analysis process

Then the program will search through each modified critical section for signs of each antipattern. The initial classification is between Hot1 and Hot2, it is a straightforward classification depending on the number of critical sections that use that sane lock object as the current one, if the critical section being examined is the only one using that object, then it is assigned Hot1, otherwise it is Hot2. Next Hot3_1 is examined, it is different as the examination is of the outside of the critical section while the rest focus primarily on the inside, therefore it uses the original non modified copy of the code. To check the program will look at the code near to the start of the critical section and look for loops that contain the critical section and not much else, a larger distance becomes less helpful and more complex due to function calls. Hot3_2 is checked the opposite way, the inside of the loop is checked for loops, and recursion that has been marked earlier, then the maximum depth of the loop is returned and recorded. In the case of Unpredictable and Simultaneous the unmodified critical section is used because it holds a reference to the entire program's AST and can be used to examine the code outside of the critical section. Both of them search for specific statements, Unpredictable for the lock object being assigned one of the invalid values, and Simultaneous for uses of the lock object outside of a valid synchronized setting. Additionally, the output can be read as priority, the rightmost entries refer to more specific issues, while the leftmost, such as Hot1, refer to more general problems.

Additional information is given in section 4.3.1 with a walkthrough of the logic for example outputs.

These features are found primarily according to the contents of AST nodes, and their immediate relations. As was previously stated, the structure of the ASTs was found to be of little use distinguishing the antipattern examples, because of this the literal layout of the AST is not a focus of the analysis. However, there are examples where the AST is used to analyse the code structure such as Hot3_2 as previously mentioned, and Hot2, which requires knowledge of other critical sections in the code.

The AST node analysis is done to extract information about the java code with all of the variation from the programmer removed. This means that the tool can accurately identify the code contents. Those are then compared to known signs of the listed antipatterns and recorded for later use.

The Program will then use that information to output a binary vector that represents the features of various antipatterns.That binary vector can then be used to determine what the problem most likely i. The categories are arranged in order of ascending precision, so the rightmost entry is the most specific.

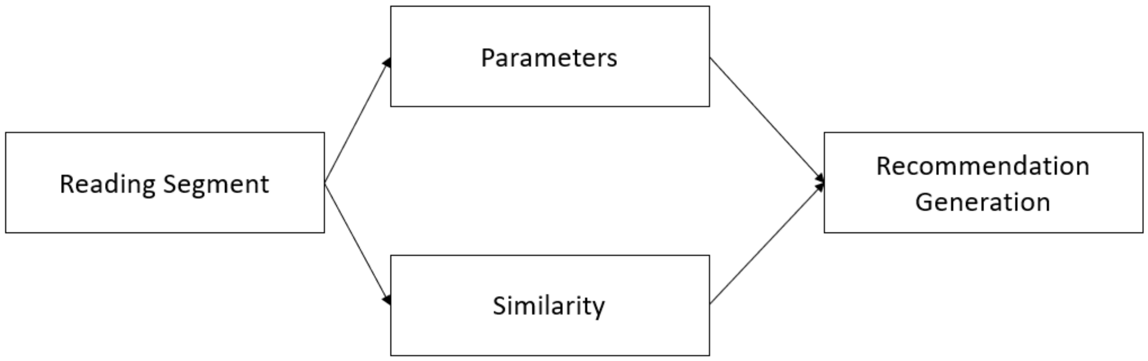


Figure 3.8: Detailed recommendation process

Listing 15 Algorithm for analysing Java project.

```
def analyze(read, target):
    for each critical section in keyList:
        if is target OR target==nul:
            AST_Copy=replace(critical section AST reference)
            search for features based on modified critical
             ↪  section AST and original
                reference AST is able to access upper levels
                 ↪  of tree and get information on parent
                 ↪  class
            output classification results if specified
            return binary vector to main class
def replace(AST):
    create AST copy
    locate function calls inside critical section
    search project directory for function
    if found append function contents to section
    else add function information to list of functions not
     ↪  found
    return modified critical section
```

3.4.3 Recommendations

The output of the analysis will be fed into the recommender system, figure 3.8, which will generate potential fixes based on the antipattern definitions in table 3.1. The recommender also uses information from the ASTs to make the recommendations more relevant to the program.

3.4.4 Antipattern Sources

Hot section is the most common source of contention, because of this it has multiple forms which were divided into 1, 2, 3_1, and 3_2, however they are all contention caused by a single lock object or synchronized region being significantly more contended than the rest of the program.

When looking for sources it is important to note that many causes of contention are not language specific, this means that, in addition to java specific resources, resources from other languages can be used, as long as it is confirmed that they do not rely on things only present in that language.

The Microsoft support page "How to reduce lock contention in SQL Server" it states that the main way is to avoid having multiple processes update or insert the same data page, this is somewhat helpful, but mainly means to reduce contention by not contending. It later goes into more detail, with the most relevant to this work points being: avoid transactions that contain user interaction, keep modification transactions short, and keep transactions in one batch. As these all relate to keeping the duration of the critical section low they directly support this paper's definition of Hot1, and Hot3_2.

The IBM documentation page "Resolving lock contention" states that "There are two mechanisms for reducing the rate of lock contention", they are to reduce the time the lock is held for and to reduce the scope of the lock. In this scope is used to mean the proportion of the data controlled by the lock, for example locking rows of a table individually rather than as a whole.

These also support Hot1 and Hot3_2 as the length of a synchronized region is important for them, however it also supports Hot2, Overly Split, and Hot3_2 as splitting a critical section should result in a reduced rate at which it is acquired.

It is also worth noting that this page supports the recommended fixes to lock contention, Lock Splitting and Lock Stripping.

Reducing lock contention on multi-core platforms by Haimiao Ding et al. states that they evaluate lock contention in two ways. The problem of large critical sections, and frequent lock request. These are the two divisions made in between runtime contention issues, and they connect to this

work in the same way.
Simultaneous and Unpredictable both appear in the SpotBugs Bug list as IS: Inconsistent synchronization (IS2_INCONSISTENT_SYNC) and DL: Synchronization on Boolean (DL_SYNCHRONIZATION_ON_BOOLEAN) respectively.
All types or related solutions also appear in many other locations defined in the form of both problems and solutions. The Spotbugs[5] documentation contains many references to possible sources of contention or issues with synchronized regions, it mostly focuses on problems such as Simultaneous and Unpredictable. The article ”Improving Lock Performance in Java”[23] covers several tips for improving lock performance, which can be turned around to determine what problems they are related to. The article recommends reducing the lock scope, removing unnecessary code from the synchronized region, splitting the lock, and switching to synchronized data types, these mostly relate to the Hot1 antipattern, but are related to most of the ”Hot” antipatterns. The IBM Documentation about resolving lock contention[24] suggests two methods: reducing lock duration and reducing lock scope. Scope of a lock refers to how much data is controlled by a single lock, the problems related to these types would be Hot1 for duration and Hot2 for scope. The Oracle ATG Repository Guide page ”Resolving Lock Contention”[25] suggests information on how to resolve contention on a database, the recommended solution is to split the database to allow synchronous access, this would help to resolve a Hot2 issue. Sonar Source[26] is a code analysis tool, they have a documentation page that lists the rules that are checked for in Java by filtering it to the types of rules related to synchronization and locks, it shows the types of problems it looks for. They are issues like those in Spotbugs where many are functional rather than performance issues, but of the performance issues shown they relate most to Hot2.

3.5 classification output explanation

The classification process prior to the recommendations will be shown with several parts shown in listing 16, each part is separated by a line to make reading easier. The parts are the name of the

Listing 16 Example output in file HFileCleaner.cancel_0.txt.

```
looking:HFileCleaner.cancel
HFileCleaner        cancel
-----------------------
@Override
public synchronized void cancel(boolean mayInterruptIfRunning) {
    super.cancel(mayInterruptIfRunning);
    for (Thread t : this.threads) {
        t.interrupt();
    }
}
-----------------------
@Override
public synchronized void cancel(boolean mayInterruptIfRunning) {
    super.cancel(mayInterruptIfRunning);
    for (Thread t : this.threads) {
        t.interrupt();
    }
    {
        super.cancel(mayInterruptIfRunning);
        for (Thread t : this.threads) {
            t.interrupt();
        }
    }
}-----------------------
possible Hot1
possible type Hot3_2
1
```

lock object and location of the critical section, next the critical section as it exists in the source code, followed by the critical section as it has been modified to include contents of function calls, finally the classification results are shown on separate rows.

The name of the critical section is either the object that is used to synchronize, or based on the location of the critical section in the case synchronizing on ”this” or a synchronized method.

In situations where a critical section contains a function call the code from inside function will be added to the end of that section, provided that it can be found in the provided source code.

Each antipattern found in each critical section is output individually, additionally Hot3_2 will output the largest loop depth in the critical section. The order the antipatterns are printed in is the same used in the vector that represents the output: Hot1, Hot2, Hot3_1, Hot3_2, overlySplit, Simultaneous, and Unpredictable.

Chapter 4

VALIDATION

This section will describe the effectiveness of the algorithm on the test and real datasets with regards to how accurately it can determine what antipatterns are present. The test data is the set of patterns that are presented in section 3.2, and they are included to determine that it can distinguish between the antipattern types. The more important test data is pulled from the Apache HBase GitHub, it is an open-source project written primarily in java.

The output will be given for each critical section examined in two parts: a text file that contains the object that the critical section locks on, the location of the critical section in the code(class and method) , two variants of the method(The original unmodified method and another containing the method with function calls expanded where possible), and the results of the classification, the second part is a binary vector representing the presence of each antipattern to be used by the recommender. An example is shown in listing 17 for reference the binary vector would be [1 0 0 0 0 0 0].

When determining the output it will return multiple possible results, as shown above, this is because there is overlap between patterns, especially in the static space and many differences only appear at runtime. An example of overlap is between Hot section 1 shown in Listing1 and Hot Section 3 Type 2 show in Listing7. This is partially because the anti patterns were created to describe problematic behavior and were then represented as code, and also due to the somewhat hierarchical layout of the antipatterns.

Listing 17 Example output in file AbstractWALRoller.this_0.txt, this is a file selected from the HBase output to demonstrate the replacement of a line(controller.requestRoll() is swapped for the method's contents.)
This object is classified as possible Hot1.

```
looking:AbstractWALRoller.this
AbstractWALRoller        logRollRequested
-----------------------
// TODO logs will contend with each other here, replace with
 ↪  e.g. DelayedQueue
synchronized (AbstractWALRoller.this) {
    RollController controller = wals.computeIfAbsent(wal, rc
     ↪  -> new RollController(wal));
    controller.requestRoll();
    AbstractWALRoller.this.notifyAll();
}
-----------------------
// TODO logs will contend with each other here, replace with
 ↪  e.g. DelayedQueue
synchronized (AbstractWALRoller.this) {
    RollController controller = wals.computeIfAbsent(wal, rc
     ↪  -> new RollController(wal));
    controller.requestRoll();
    AbstractWALRoller.this.notifyAll();
    {
        this.rollRequest.set(true);
    }
}-----------------------
possible Hot1
```

4.1 Sample Dataset

The classification of the sample antipattern code presented in section 3.2 performed as expected, each of them properly classified the with their intended classes, however as expected many of the entries were classified as additional antipatterns. Table 4.1 lists the results from each of the sample codes with this classifier. As you can see the classifier worked, but all of them also classified as either Hot1 or Hot2, this is because they function as a general category which can be further

	Hot1	Hot2	Hot3_1	Hot3_2	OverlySplit	Simultaneous	Unpredictable
Hot1	1						
Hot2		1					
Hot3_1	1		1				
Hot3_2	1			1			
OverlySplit		1			1		
Simultaneous	1					1	
Unpredictable	1						1

Table 4.1: Results of classification(horizontal) vs expected result(vertical)
1's mark antipattern identification result.

dividend to provide more information.

Hot1 is a situation where the critical section is causing the contention due to single long processes, and Hot2 is where the critical section is where a synchronized object is slowed down by a large number of requests. Therefore, the classifier works for the sample data.

4.2 Validation Test Setup for Open Source Data Testing

To test and validate the program it was given an entire project from which it would pull data. The program would then perform the analysis on the critical sections that it extracted from the code. Once it returned the critical sections and their matching anti-patterns, it would be manually reviewed to confirm that the critical sections that were examined for instances of anti-patterns were matched correctly, and that the features that indicate their presence were correctly identified. Once the code review was complete, it could then be compared to the generated classification to determine how well it functioned.

4.3 Test Dataset

The test data as stated was taken from the Apache HBase GitHub page, the entire repository was searched for critical sections which were used in the classification, no all of the proposed antipatterns occur in this dataset, however that may be expected as this is a actively maintained project which functions properly. The breakdown of classes in shown in figure 4.1, the categories are given as the binary vectors of the antipatterns as described above. The exact numbers for each section are Hot1 only 318 (49.9%), Hot2 only 173 (27.2%), Hot2 and Hot3_2 67 (10.5%), and Hot1 and Hot3_2 79 (12.4%). There are 637 total samples.

Hot1 is present in 62.3% of the dataset, Hot2 is present in 37.7% of the dataset, and Hot3_2 is present in 22% of the dataset. As you can see Hot1 is the larges category by far, at almost half of the

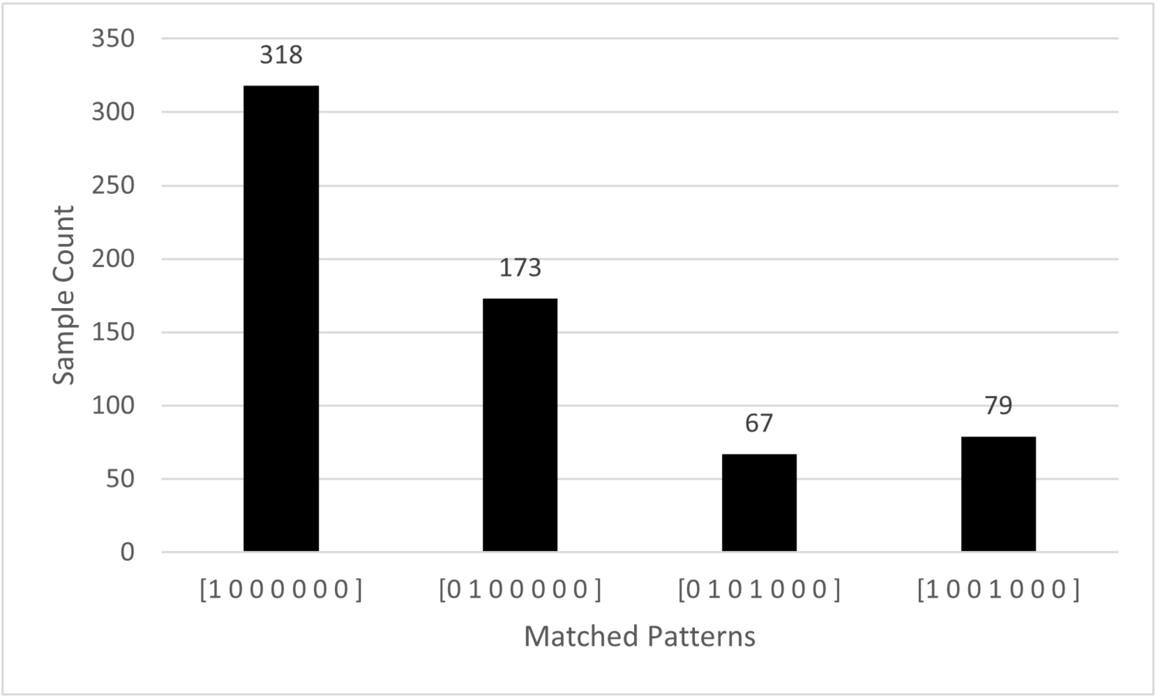


Figure 4.1: Output categories of Apache HBase. The vectors represent [Hot1 Hot2 Hot3_1 Hot3_2 OverlySplit Simultaneous Unpredictable]

dataset, and including partial matches to Hot1 it is 62% of the dataset, this is to be expected because

Hot1 is the most simple case, as it involves as little as one line of code inside of a critical section and due to the constraints of this static analysis approach it will include every critical section that is only a single minor operation, Hot2 is large for the same reason, however it is less so because it is requires the synchronized object to be used in many places.

4.3.1 Reviewing Samples From Test Data

This section will review how accurate the classification of the test was, this will be done by selecting examples from the output randomly until one from each category if found, and then comparing that to the output result, all 636 from the HBase dataset were manually examined and found to be correct, but only one of each category are shown here. The first sample output is shown in listing 18. The result given is just Hot1. This makes sense because, by examining the list of outputs we can see that that is the only critical section where the key "AbstractWALRoller.this" was used therefore it can't belong to Hot2, and it belongs to Hot1 because it acts as a default state. It has no looping, split critical sections, improper synchronization, or variable definition, so it can't be any of the other categories.

The second sample output is shown in listing 19, from entries_4.txt. The result is only Hot2, similarly to the first example this example has no looping, split critical sections, improper synchronization, or variable definition so most of the categories are eliminated. The difference is that in the output listing the synchronized object shows up 4 times rather than just once, this means that it is more likely to be a Hot2 antipattern instance than Hot1, therefore the classification results are correct.

The third example is from HFileCleaner.cancel_0.txt it is shown in listing 20. The result shows Hot1 and Hot3_2. There is only one place this critical section is used therefore it cannot be Hot2, it has no looping outside of the critical section, split critical sections, and no improper synchronization or variable definition so it can't be any of them. There are two loops in the critical section, however they are separate, and the depth is only one, but that is enough to be Hot3_2.

Listing 18 Example output in file AbstractWALRoller.this_0.txt, an example output from the HBase project that is classified as Hot1.

```
looking:AbstractWALRoller.this
AbstractWALRoller        logRollRequested
-----------------------
// TODO logs will contend with each other here, replace with
 ↪  e.g. DelayedQueue
synchronized (AbstractWALRoller.this) {
    RollController controller = wals.computeIfAbsent(wal, rc
     ↪  -> new RollController(wal));
    controller.requestRoll();
    AbstractWALRoller.this.notifyAll();
}
-----------------------
// TODO logs will contend with each other here, replace with
 ↪  e.g. DelayedQueue
synchronized (AbstractWALRoller.this) {
    RollController controller = wals.computeIfAbsent(wal, rc
     ↪  -> new RollController(wal));
    controller.requestRoll();
    AbstractWALRoller.this.notifyAll();
    {
        this.rollRequest.set(true);
    }
}-----------------------
possible Hot1
```

The final example is shown in listing 21, file storefiles_1.txt. This is very similar to the fourth example, it has no looping outside of the critical section, split critical sections, and no improper synchronization or variable definition and it contains 3 loops with a maximum depth of 1, the difference is that there are multiple places that the storefiles object is used. Therefore, it matches with Hot2 and Hot3_2.

Overall the classifier is accurate on the types that were present in the dataset, however OverlySplit and Hot3_1 did not appear which are the patterns that were expected to appear more, however that

Listing 19 Example output in file entries_4.txt, an example output from the HBase project that is classified as Hot2.

```
looking:entries
RegionReplicationSink        getFailedReplicas
-----------------------
synchronized (entries) {
    return this.failedReplicas;
}
-----------------------
synchronized (entries) {
    return this.failedReplicas;
}-----------------------
possible Hot2
```

was not the case, the other patterns not appearing in the dataset was expected Simultaneous and Unpredictable are both very specific problems that would have a significant effect and be fixed while doing testing, the majority of this code is not currently being written, therefore these errors would be strange to see.

Other Data

The program was also run on several other open source programs Glide and EventBus. They also show similar behavior of antipatterns as HBase, they all have the same set of labels, however they do not all have the same proportions shown in figure 4.2 each of the bars represents the percentage of each repository that matched with those antipatterns.

They all follow the same pattern of Hot1 and Hot2 being the most common, with Hot3_2 being third. No other antipattern types were found in the three repositories. The other two repositories had more of Hot1 than Hot2 but that is most likely due to differences in the function of the program rather than anything meaningful.

Listing 20 Example output in file HFileCleaner.cancel_0.txt, an example output from the HBase project that is classified as Hot3_2 and Hot1.

```
looking:HFileCleaner.cancel
HFileCleaner        cancel
-----------------------
@Override
public synchronized void cancel(boolean mayInterruptIfRunning) {
    super.cancel(mayInterruptIfRunning);
    for (Thread t : this.threads) {
        t.interrupt();
    }
}
-----------------------
@Override
public synchronized void cancel(boolean mayInterruptIfRunning) {
    super.cancel(mayInterruptIfRunning);
    for (Thread t : this.threads) {
        t.interrupt();
    }
    {
        super.cancel(mayInterruptIfRunning);
        for (Thread t : this.threads) {
            t.interrupt();
        }
    }
}-----------------------
possible Hot1
possible type Hot3_2
1
```

Listing 21 Example output in file storefiles_1.txt, an example output from the HBase project that is classified as Hot3_2 and Hot2.

```
looking:storefiles
FileBasedStoreFileTracker        doAddNewStoreFiles
-----------------------
synchronized (storefiles) {
    StoreFileList.Builder builder = StoreFileList.newBuilder();
    for (StoreFileInfo info : storefiles.values()) {
        builder.addStoreFile(toStoreFileEntry(info));
    }
    for (StoreFileInfo info : newFiles) {
        builder.addStoreFile(toStoreFileEntry(info));
    }
    backedFile.update(builder);
    for (StoreFileInfo info : newFiles) {
        storefiles.put(info.getPath().getName(), info);
    }
}
-----------------------
synchronized (storefiles) {
    StoreFileList.Builder builder = StoreFileList.newBuilder();
    for (StoreFileInfo info : storefiles.values()) {
        builder.addStoreFile(toStoreFileEntry(info));
    }
    for (StoreFileInfo info : newFiles) {
        builder.addStoreFile(toStoreFileEntry(info));
    }
    backedFile.update(builder);
    for (StoreFileInfo info : newFiles) {
        storefiles.put(info.getPath().getName(), info);
    }
}-----------------------
possible Hot2
possible type Hot3_2
1
```

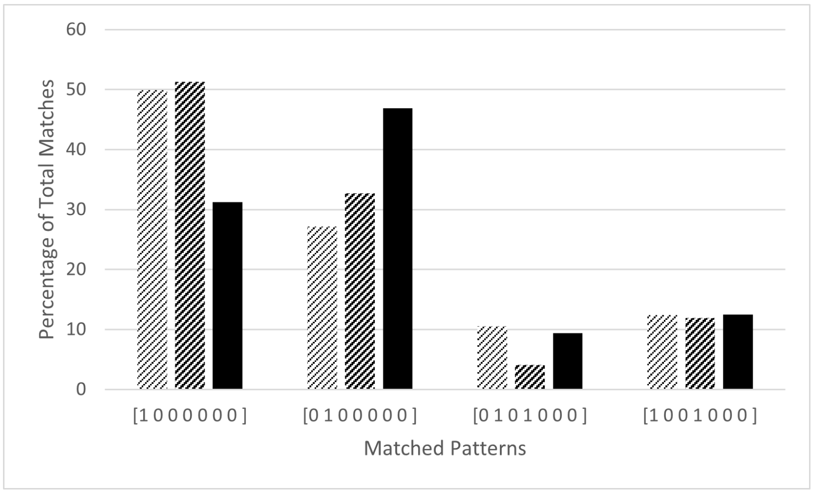


Figure 4.2: Apache Hbase, Glide, and EventBus open source projects distribution of antipatterns found. The vectors represent [Hot1 Hot2 Hot3 1 Hot3 2 OverlySplit Simultaneous Unpredictable]

Chapter 5

CONCLUSION

In this work we set out to create a set of antipatterns that represent possible and frequent types of lock contention that appear in Java code and then create a tool that will classify java code according to those antipatterns.

Many of the antipatterns established do appear in the open-source code that was examined (Apache HBase, glide, eventBus). Out of the seven types listed 3 appeared in that project, and 3 appeared in the list of types used by other classifiers such as SpotBugs[5]. This is what was expected as Hot1 is the most common type of contention, and Unpredictable, Simultaneous, and Overly Split are unlikely to appear in a finished project due to their ability in introduce errors in the codes execution and would likely be fixed early on in development.

For recommendations details are mostly left for future work, however general solutions that apply to each antipattern type are provided in table 3.2. When the user is provided with the list of antipatterns that match with the given lock objects critical sections, they will have a set of recommendations that may fix the problem.

The classifier is able to reliably differentiate between the antipatterns, however many critical sections will match with multiple antipatterns, this is due to overlap between critical sections. As an example the code example for Hot11 matches with the definition and example for 7, supported by the runtime example in table 4.1. This also means that each of the antipatterns have overlapping solutions to the lock contention, as seen in table 3.1. This is not a problem and means the classifier is functioning as expected. If more antipatterns are added it will be possible to organise the antipatterns into a hierarchy that should be able to provide more detailed information about the critical section and provide more precise recommendations.

5.1 THREATS TO VALIDITY

The primary threat to the validity of this work is that it is the source of all of the antipatterns, the classifier, and the verification of the classification. This means that the classification may be correct, but without an additional party to confirm it, both the classification and the confirmation are based on the same logic.

5.2 FUTURE WORK

In the future we plan to add more categories to the antipattern database to provide more detail on issues and cover a wider area. Additionally, we plan to create the recommendation system to make the antipattern identification more helpful. Once the components are created, we plan to integrate these into a tool that will fully cover the full analysis process including the dynamic analysis. The static analysis component will also be tested on a larger data sample.

5.3 LIMITATIONS

This work is limited mostly by the situation it is designed for, as this tool is not designed to run independently and is a static analysis tool, it is not able to determine the presence of lock contention or other problems, only to find possible sources of those problems, this means that it won't provide helpful information by itself. Additionally, recommendations are very simple in this work and need further detail.

Bibliography

[1] N. H. Khan, J. Robertson, R. Liscano, A. Azim, V. Sundaresan, and Y.-K. Chang, "Lock contention performance classification for java intrinsic locks," eng, in *Runtime Verification*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 274–282, ISBN: 9783031171956.

[2] J. Robertson, A. Ahmed, A. Azim, R. Liscano, V. Sundaresan, and Y.-K. Chang, "Java lock contention classification and recommendation through synchronization antipatterns," in *WeaveSphere 2022*.

[3] R. Liscano, A. Azim, J. Robertson, *et al.*, "A preliminary investigation into runtime fault identification and localization of java-based cloud-native microservice applications," in *Cascon 2020*.

[4] A. Nisbet, N. M. Nobre, G. Riley, and M. Luján, "Profiling and tracing support for java applications," *ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pp. 119–126, 2019. DOI: `10.1145/3297663.3309677`.

[5] Spotbugs, *Spotbugs/spotbugs at fa9e53ac7d969e8f945ee8a90365173c31ce6f8a*. [Online]. Available: `https://github.com/spotbugs/spotbugs/tree/fa9e53ac7d969e8f945ee8a90365173c31ce6f8a`.

[6] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," 2003, 7–pp.

[7] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 298–313, ISBN: 9781450349383. DOI: `10.1145/3064176.3064186`. [Online]. Available: `https://doi.org/10.1145/3064176.3064186`.

[8] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010. DOI: `10.1109/TSE.2009.50`.

[9] J. Bradbury and K. Jalbert, "Defining a catalog of programming anti-patterns for concurrent java," Oct. 2009.

[10] D. Bán and R. Ferenc, "Recognizing antipatterns and analyzing their effects on software maintainability," Jun. 2014, pp. 337–352, ISBN: 978-3-319-09155-6. DOI: `10.1007/978-3-319-09156-3_25`.

[11] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 181–190. DOI: `10.1109/CSMR.2011.24`.

[12] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen, "Halock: Hardware-assisted lock contention detection in multithreaded applications," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 253–262, 2012, ISSN: 1089795X. DOI: `10.1145/2370816.2370854`.

[13] F. David, "Continuous and efficient lock profiling for java on multicore architectures. (profilage continu et efficient de verrous pour java pour les architectures multicœurs)," Ph.D. dissertation, Pierre and Marie Curie University, Paris, France, 2016.

[14] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers, "Static analyses for eliminating unnecessary synchronization from java programs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1694, pp. 19–38, 1999, ISSN: 16113349. DOI: `10.1007/3-540-48294-6_2`.

[15] M. A. Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for java concurrency bugs," Sep. 2010.

[16] A. K. Alvi and M. Zulkernine, "A security pattern detection framework for building more secure software," *Journal of Systems and Software*, vol. 171, p. 110 838, 2021, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2020.110838`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121220302296`.

[17] L. Zhang, Y. Sun, H. Song, W. Wang, and G. Huang, "Detecting anti-patterns in java ee runtime system model," in *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, ser. Internetware '12, Qingdao, China: Association for Computing Machinery, 2012, ISBN: 9781450318884. DOI: `10.1145/2430475.2430496`. [Online]. Available: `https://doi.org/10.1145/2430475.2430496`.

[18] S. Park, R. W. Vuduc, and M. J. Harrold, “Falcon: Fault localization in concurrent programs,” *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 245–254, 2010, ISSN: 02705257. DOI: 10.1145/1806799.1806838.

[19] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, “Framework for testing multi-threaded java programs,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 485–499, 2003. DOI: https://doi.org/10.1002/cpe.654. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.654. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.654.

[20] Martinekuan, *Busy database antipattern - performance antipatterns for cloud apps*. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/antipatterns/busy-database/.

[21] B. Göetz and A. W. Professional, “Java concurrency in practice,” *Building*, vol. 39, p. 384, 11 2006, ISSN: 03008495. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Java+Concurrency+in+Practice#0.

[22] *Tools for your java code*. [Online]. Available: https://javaparser.org/.

[23] *Improving lock performance in java*, Jan. 2015. [Online]. Available: https://dzone.com/articles/improving-lock-performance.

[24] *Resolving lock contention*. [Online]. Available: https://www.ibm.com/docs/en/mon-diag-tools?topic=perspective-resolving-lock-contention.

[25] “Resolving lock contention,” *ATG Repository Guide*, [Online]. Available: `https://docs.oracle.com/cd/E23095_01/Platform.93/RepositoryGuide/html/s1005resolvinglockcontention01.html`.

[26] *Java static code analysis*. [Online]. Available: `https://rules.sonarsource.com/java/`.

Appendices

Appendix A

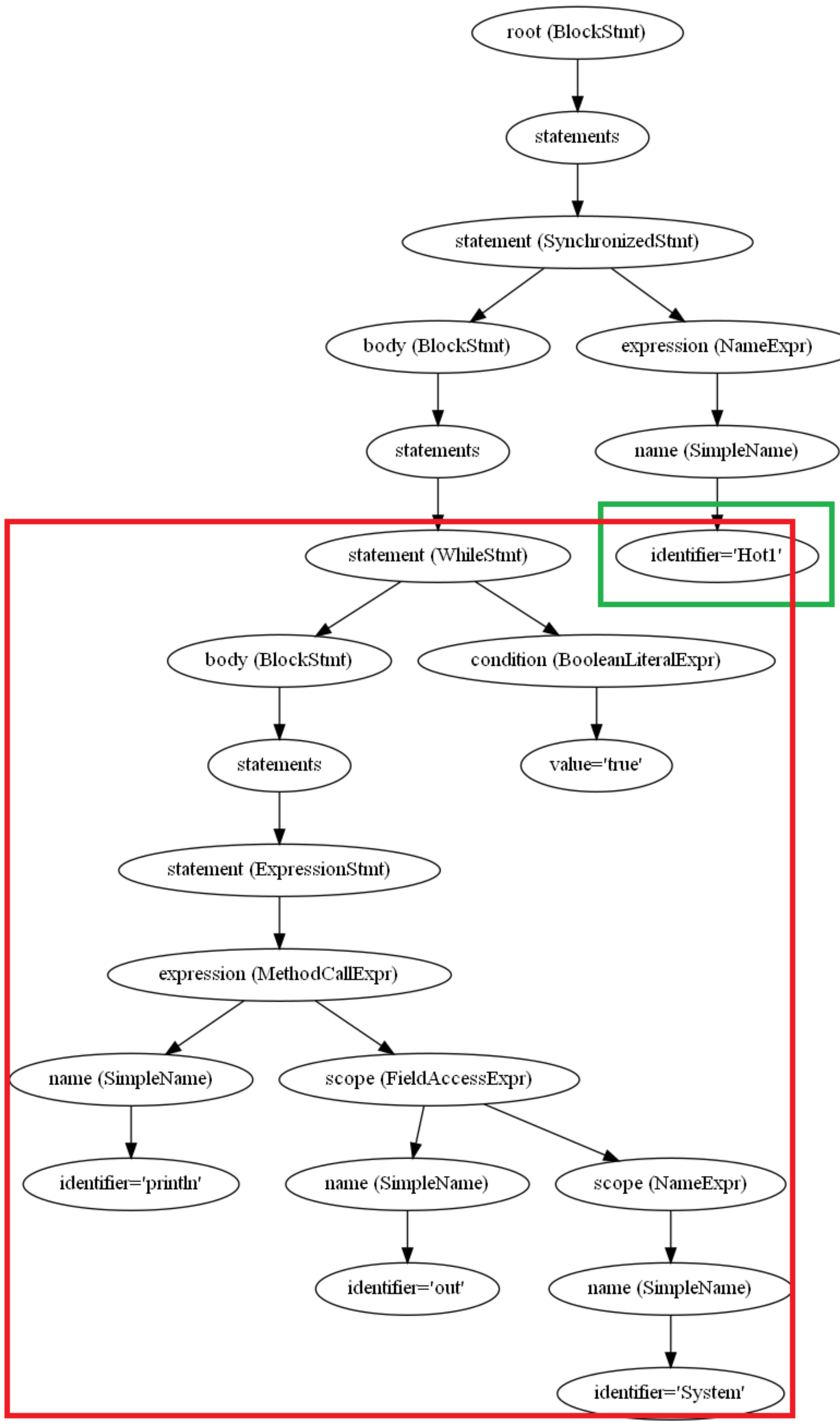


Figure A.1: Hot1 AST annotated with signs of contention

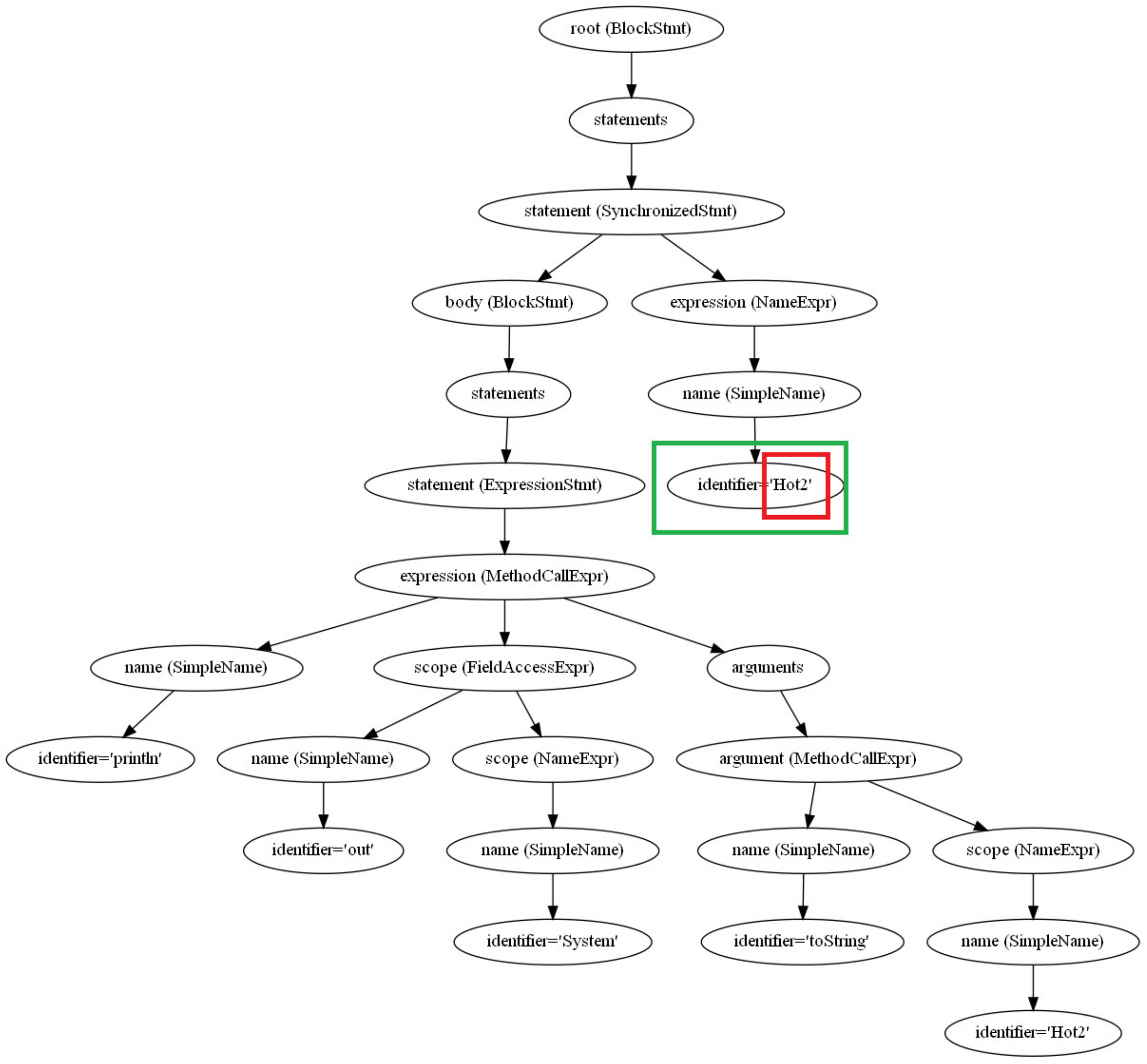


Figure A.2: Hot2 AST annotated with signs of contention part 1

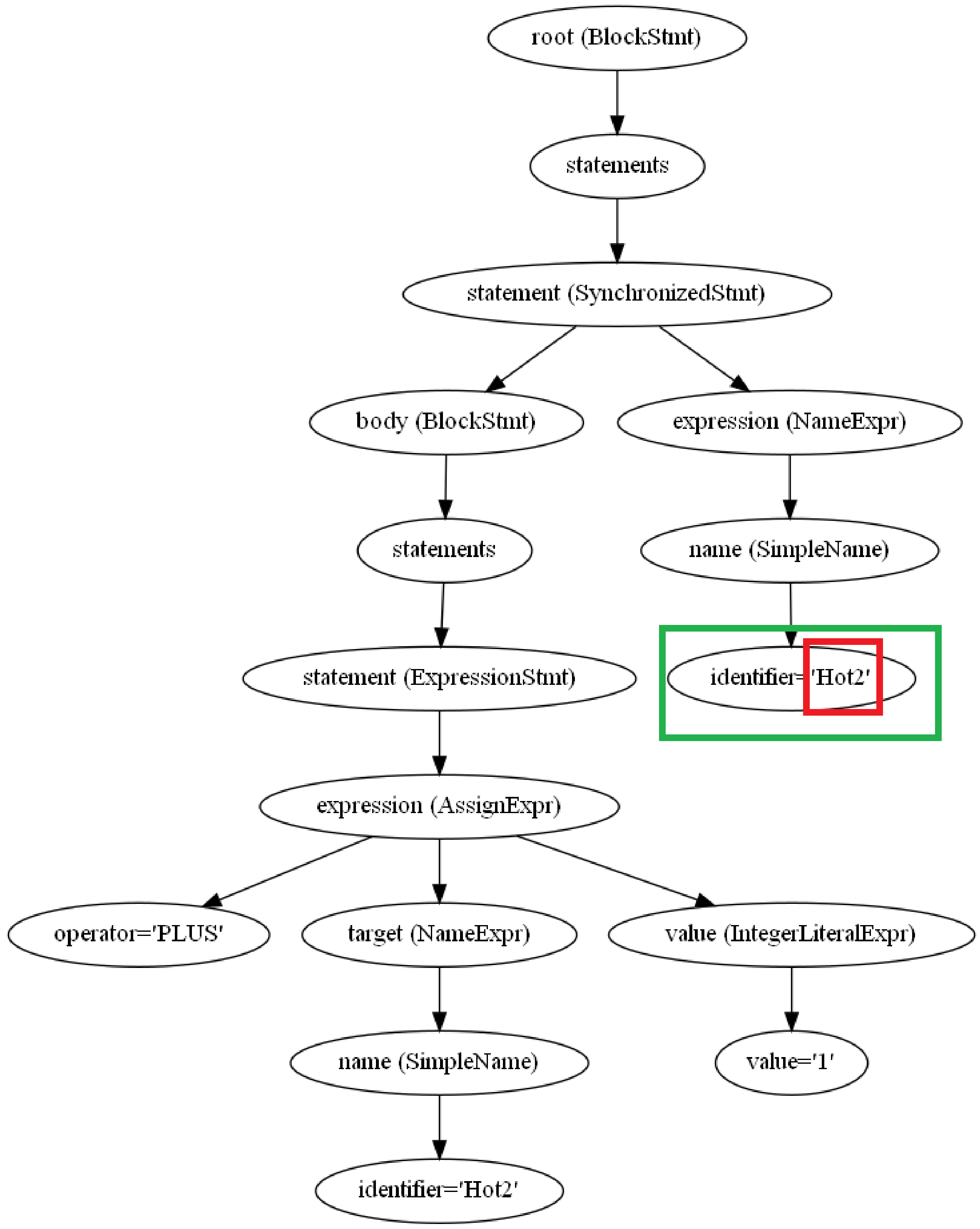


Figure A.3: Hot2 AST annotated with signs of contention part 2