Measuring Java Util Concurrent Parking Contention in the IBM J9 Virtual machine

by

Panagiotis Patros

Degree of Informatics and Telecommunications, University of Athens, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master's in Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Eric Aubanel, PhD Computer Science
	David Bremner, PhD Computer Science
Examining Board:	Gerhard Dueck, PhD Computer Science
	Kenneth Kent, PhD Computer Science
	Mary Kaye, MEng, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

February, 2014

Abstract

Java Util Concurrent (JUC) is a widely used library providing support for multithreaded applications. JUC provides a variety of tools such as explicit locks, thread pools, blocking queues etc. Many of these constructs use thread parking as a means of synchronizing threads. Measuring thread parking contention can potentially provide information which can be used to accelerate Java applications through the identification of bottlenecks. To the best of our knowledge, no such tool exists, possessing the ability to run as part of the JVM while not otherwise modifying the JUC base classes.

IBM's J9 Virtual Machine, which is an implementation of a Java Virtual Machine (JVM), has been modified to store parking contention data and output the results to the screen or to Java dumps. We also show that our tool has a negligible effect on the performance of the JVM. Furthermore, we have investigated coding patterns involving JUC locks and have used our tool to identify bottlenecks and consequently accelerate the Java code.

We conclude that our instrumentation to the JUC thread parking implementation provides useful information to developers hoping to optimize applica-

tion performance.

Dedication

To Bertrand Russell.

Acknowledgements

Andrew Somerville, Marcel Dombrowski and Kim Briggs, all colleagues of mine at the CASA IBM project, have been very helpful by introducing me to the necessary background needed for this project. Andrew also read the final draft of the thesis and pointed out a number of typos in the document. Michael Dawson, our IBM contact, has been crucial to all stages of the development of this thesis and also came up with the idea of park monitoring. Without his assistance, this project would not have been the same.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Java [12] is a high level programming language which is compiled to machine-independent instructions, referred to as bytecodes [13]. Bytecode is interpreted at run-time by a program called the Java Virtual Machine (JVM) [2, 5]. J9 is IBM's implementation of the Java Virtual Machine Specification and one of the main JVMs currently on the market [2].

Java provides a great deal of support for multi-threaded programming including thread management, intrinsic locks, thread waiting, thread sleeping and yielding, volatile variables and immutable objects [4]. However, only basic synchronization techniques are covered by the core Java specifications. Instead, more advanced functionality is provided by the Java concurrency library, Java Util Concurrent (JUC), which includes explicit locks, concurrent data structures, advanced task management functionality and a plethora of other useful features [11, 4].

JUC classes use a technique called thread parking as a means of thread synchronization [4]. In this project, we implemented a tool to measure JUC parking contention data for Java applications running on the J9 JVM and instrumented its functions to provide us with the necessary data. This data can be presented to the user via the console or in Java dump files. The provided metrics furnish developers (and optimizers) with the tools they need to measure which JUC objects are most heavily park-contended and consequently, which objects are the most likely candidates for optimization. In order to show a proof of concept for our approach, we measured the parking contention of two Java coding patterns with JUC lock bottlenecks and used these results to identify the bottleneck and to effectively optimize the code. Consequently, the hypothesis we seek to test is if substantial performance gains can be made by analyzing JUC park contention.

In Chapter 2 we introduce the reader to the necessary background for our work which includes a discussion of Java's multithreading support, the difficulties in using it, a brief presentation of the JUC library, a small introduction of other profiling tools for Java, a description of the most relevant JVM performance benchmarks and a brief survey of the relevant literature.

We continue with Chapter 3 where we describe the changes we performed to IBM's J9 JVM to develop our tool. More specifically, we discuss the underlying thread parking support in J9 and the available parameters for starting profiling with our tool. Moreover, we present a general overview of our implementation's architecture, the data structures and algorithms used,

the measured and derived metrics provided and how the user can acquire the results of the profiling.

In Chapter 4 we investigate the effects of our modifications on the J9 JVM by measuring the overhead induced by our tool. Our analysis is broken down into two main categories: throughput and resident memory overhead. Furthermore, for each category, we check the overhead both when our tool is running and when it is not.

We continue in Chapter 5 discussing the importance of our tool in JUC profiling by investigating specific code patterns in which we try to find bottlenecks by using the provided park contention metrics. We compare the results by implementing a second version of the pattern, with Java's built-in multithreading capabilities instead of the JUC functionalities, and profile that on Healthcenter, IBM's monitoring program for the J9 JVM [3]. Finally, we show that the bottlenecks suggested by the profiling tools are correct by optimizing the code patterns accordingly and taking speedup measurements.

We conclude in Chapter 6, where we present possible future extensions on our work alongside of a brief summary of our project and the conclusions drawn from it.

Chapter 2

Background

2.1 Java

Java [12] is a relatively high level programming language based on C/C++ but also incorporating aspects from other languages. It is a general-purpose, concurrent, class-based and object oriented language with features that make it very attractive to industry.

2.2 Java Virtual Machine (JVM)

Java is compiled to machine-independent instructions referred to as bytecode [13]. Bytecode is interpreted/compiled at run-time by a program called a Java Virtual Machine (JVM) [2, 5]. Furthermore, memory deallocation takes place automatically by a component of the JVM named a Garbage Collector

(GC) [12].

One of the main JVMs currently available in the market is IBM's J9 [2]. Another important one is Oracle's HotSpot [5].

2.3 Built-in Thread Support in Java

Programming in a multi-threaded environment poses challenges unlike those of the standard single-threaded model. We investigate some of these issues and elaborate on the ways of dealing with them. Java provides a number of built-in concurrency related features which include thread management, intrinsic locks, thread waiting, thread sleeping and yielding, volatile variables and immutable objects [4]. In this section we present these constructs and explain their functionality.

2.3.1 Hazards in Multithreaded Java Programming

A number of hazards can be introduced by badly written concurrency code in Java. These mistakes are hard to spot and have the tendency to appear under significant stress which is usually the case when the application is used in production. Therefore, special care must be taken by the developer to avoid any such cases which include race conditions, memory visibility, object escaping and unsafe object publication [12].

2.3.1.1 Race Conditions

Race conditions can occur when two or more threads try to manipulate shared memory without using proper synchronization. This causes instructions which are supposed to be executed atomically to interleave amongst threads' execution. Two types of race conditions can be distinguished [12]:
The first one is called **Read-Modify-Write**. In that case the threads read the same value for a variable from the main memory, modify it, being unaware of the other threads doing so as well, and finally store it back in memory. Only the last store will practically be successful, overwriting any changes made by previously unsynchronized writes.
An example of code that can cause this behavior is given in Listing 2.1. When the increment command is interpreted, it is split into three operations which read the variable `counter` from the main memory, increment it by one and store the result back. If two threads executing concurrently run this code, they can potentially read the same initial value for the variable causing one of the increment operations to be nullified. Therefore, two threads both executing the function `ReadModifyWrite.incrementCounter()` can potentially increment the variable `counter` only once and not twice as it is expected to happen.

Listing 2.1: Read-Modify-Write Race Condition Code Example

```
public class ReadModifyWrite{
        volatile int counter = 0;
        public void incrementCounter(){
                /* The increment statement is not
```

```
                * atomic and is broken down to
                * read, increment and store*/
                counter++;
        }
}
```

The second case of race condition is named **Check-Then-Act**. In that case the threads check for a specific condition, the validity of which can be changed by the branching code. Potentially more than one thread could pass or fail that condition concurrently; therefore, all these could execute the same branching code that alters the condition's validity.

An example of code that can cause this behavior is given in Listing 2.2. Suppose two threads run concurrently the function `CheckThenAct.decideResult()` and both check the value of the variable `flag` before the other thread has the chance to change it. We can see that in this unfortunately timed scenario, the two threads will follow the same branching path and will both set the `result` to 0; something that would not happen, were that function run twice in a single threaded environment.

Listing 2.2: Check-Then-Act Race Condition Code Example

```
public class CheckThenAct{
        volatile int result;
        volatile boolean flag = true;
        public void decideResult(){
                /* The code inside the if blocks
                * is not executed atomically*/
                if ( true == flag ){
                        result = 0;
                        flag = false;
                }
                else{
                        result = 1;
```

```
                        flag = true;
                }
        }
}
```

2.3.1.2 Statement Reordering

The Java compiler and the JVM are allowed to reorder statements and even optimize some of them out [4]. These optimizations might make sense under a single threaded execution but could be devastating under a multithreaded environment.

Listing 2.3: Statement Optimizing Code Example

```
public class Optimizing{
        boolean done = false;
        public void start(){
                /* If the variable done is not
                * changing inside the loop, it
                * can be optimized out of it*/
                while(!done){
                        execute();
                }
        }
        public void stop(){
                done = true;
        }
}
```

Listing 2.3 displays an example of code which can potentially never end due to the checking of the condition being optimized out of the loop and converted to a single branching at the beginning of the execution of the function `Optimizing.start()`. This is allowed, if the value of the variable `done` does not change within the loop. Therefore, even if another thread calls

`Optimizing.stop()`, the thread running in `Optimizing.start()` could be caught in an endless loop.

Listing 2.4: Statement Reordering Code Example

```
public class Reordering{
        int a[];
        int b[];
        public Reordering(){
                a = new int[4];
                b = new int[4];
                for(int i = 0; i < 4; i++){
                        a[i] = 0;
                        b[i] = 1;
                }
        }
        public void f(){
                a[0] = b[0];
                a[2] = b[2]; // These two statements
                a[1] = b[1]; // could be reordered
                a[3] = b[3];
        }
        public int g(){
                if(a[1] == b[1]){
                        /* If the assignment
                        * statements of a[2] and
                        * a[1] are reordered, a[2],
                        * a divisor, could be zero*/
                        return b[2]/a[2];
                }
                return 0;
        }
}
```

In Listing 2.4 an example of a problematic situation arising from statement reordering is displayed. In function `Reordering.f()`, the Java compiler is allowed to inverse the execution of the assignments to the variables `a[2]` and `a[1]`. This is an optimization that would make perfect sense for the compiler to perform because consecutive positions in arrays are stored closely enough

in memory they probably fit in the same cache line. Therefore, when `b[0]` is fetched from the main memory, the probability of `b[1]` being fetched as well is very high, which means that a costly cache miss can potentially be avoided by reordering these commands. However, in function `Reordering.g()` the branching condition could be passed without the assignment to the variable `a[2]` having been performed yet, which would lead to division by zero.

2.3.1.3 Visibility

Another common issue that occurs in multi-threaded Java programming is that of memory visibility. Each Java thread is allowed to maintain its own copy of the objects it is using; therefore, updating one variable does not necessarily mean that the local copies of the other threads will also be updated [4].

Listing 2.5: Visibility Code Example

```
public class Visibility{
        int a = 0;
        public void f1(){
                a = 1;
        }
        public int f2(){
                /* No guarantee exists that a thread
                * will see the updated value if f1
                * is called by another thread. Each
                * thread can have a local copy of
                * the object and its variables  */
                return a;
        }
}
```

An example of visibility problems is displayed in Listing 2.5. Suppose a

thread calls `Visibility.f1()` which would set the variable `a` to 1. However, only its local copy could be updated; thus, when another thread calls `Visibility.f2()` it could only read its local stale value and return `0` instead of `1`.

2.3.1.4 Non-atomic 64-bit Operations

In Java, 64-bit variables can be stored into two consecutive 32-bit memory locations. However, there is no guarantee that these two parts will be updated atomically; therefore, a thread can get a value with only half its bits updated and the other half stale [4].

Listing 2.6: 64-bit Non-atomicity Code Example

```
public class NonAtomic64{
        long X = 0x0123456789ABCDEF;
        public void setX(long X){
                /* Operations on 64-bit variables
                * can break into two 32-bit parts:
                * [this.X]_low = [X]_low
                * [this.X]_high = [X]_high*/
                this.X = X;
        }
        public long getX(){
                return X;
        }
}
```

Listing 2.6 shows an example of code which can potentially lead to problems. A thread calling `NonAtomic64.setX(0x0)` could be context switched just after changing the lower part of the variable `X` but without having changed the upper. Thus, another thread calling `NonAtomic64.getX()` at that point

can receive the improper value `0x0123456700000000` even though the variable only seems to have switched from the value `0x0123456789ABCDEF` to `0x0000000000000000`.

2.3.2 Thread Management

A new thread in Java is created when a new object of the class `Thread` is allocated [4]. Afterwards, the new thread can be started by calling its `start` function. A thread can wait for another thread to finish execution by invoking the latter's `join` function. Furthermore, a thread can be interrupted when its `interrupt` function is called. This action can be checked by the interrupted thread's `isInterrupted` function and can cause an exception to be thrown when specific blocking functions are interrupted.

2.3.3 Intrinsic Locks

Java provides a built-in lock for every object on which threads can lock on [12] using the Java keyword `synchronized` [11]. This intrinsic lock provides a means for only one thread at a time to operate within a code segment called the critical section. In the beginning of that code segment the keyword `synchronized` is used with either an explicit reference to the locking object or, if omitted, to the object of the current scope (`this`) by default.

Intrinsic locks provide guarantees on both atomicity, i.e. the code being executed by only one thread at a time, and visibility, i.e. a thread accesses

the most recent values of the shared variables; therefore, dealing with the issues discussed in Sections 2.3.1.1 and 2.3.1.3.

2.3.4 Thread Waiting

Java provides a basic thread synchronization mechanism which causes threads to wait on an object until notified [4]. This is achieved by using the functions `wait`, `notify` and `notifyAll` which are part of the `Object` class.
In particular, when a thread calls the `wait` function of an object, the former is removed from scheduling causing it to wait on the latter's monitor. The `notify` function wakes up one thread waiting on the object's monitor, whereas `notifyAll` wakes all of them up. Finally, it is important to keep in mind that in order to be able to call these functions successfully, the object's intrinsic lock must be already acquired; therefore, their invocations must be included in a `synchronized` code-block or function of that object.

2.3.5 Thread Sleeping and Yielding

Another mechanism for multithread-oriented programming provided by Java are the sleeping and yielding functions, both located in the `Thread` class [4]. The function `sleep` instructs the JVM to block the invoking thread for a certain amount of time expressed in milliseconds and/or nanoseconds.
The function `yield` hints the scheduler that the current thread is willing to relinquish the CPU which can be handled accordingly by the JVM and,

eventually, the Operating System (OS).

2.3.6 Volatile Variables

The `volatile` keyword is used to denote a special status for Java variables and resolves three concurrency related issues for multithreaded programming [11]. Firstly, it instructs the Java compiler not to reorder any statements dealing with volatile variables. Secondly, it prevents the caching of the value of a volatile variable, which solves memory visibility issues. Thirdly, it forces operations on 64-bit variables to be executed atomically; therefore, resolving the 64-bit non atomicity hazard.
Consequently, `volatile` should be considered as a light-weight synchronization mechanism with a subset of the capabilities of intrinsic locking provided by the `synchronized` statement.

2.3.7 Immutable Objects

Java provides a mechanism for declaring the value of a variable as non changing after initialization. This is performed by declaring that variable as `final` [11, 4]. If a variable cannot be changed, then there is no case that race conditions caused by it might interfere with the proper execution of the application. Furthermore, by explicitly stating this, the `final` variable can be accessed thread-safely without the need of any synchronization mechanism. In general, immutable objects are always thread-safe [11].

2.4 Java Util Concurrent (JUC)

Java Util Concurrent (JUC) [4] is a Java library which provides a variety of concurrency related functionalities. Those include concurrent data structures, management of thread pools, asynchronous task execution, synchronization mechanisms, atomics and locks [11]. They are inherently thread-safe and easy to use; therefore, JUC is the tool of choice for multithreaded Java development.

2.4.1 JUC LockSupport Class

The JUC `LockSupport` class [4] provides an underlying thread synchronization mechanism which is used by the JUC locks and other classes. This class is static which means that it cannot be instantiated and its methods are called directly using the name of the class instead of an object's [12]. The provided functionality includes the methods of Table 2.1.

In particular, a single permit is associated with each thread. When a thread is parked, if the permit is unset, the thread is blocked. If the permit is set, then the thread is not blocked but the permit is unset (Figure 2.1). When a thread is unparked, it is unblocked and its permit is set. Furthermore, a thread is also unblocked if it is interrupted or a specific deadline, associated with its parking, expires, however, its permit is not set in that case (Figure 2.2).

Table 2.1: LockSupport Class API

Object getBlocker(Thread t)	Returns the blocker object supplied to the most recent invocation of a park method that has not yet unblocked, or null if not blocked
void park()	Disables the current thread for thread scheduling purposes unless the permit is available
void park(Object blocker)	Disables the current thread for thread scheduling purposes unless the permit is available
void parkNanos(long nanos)	Disables the current thread for thread scheduling purposes, for up to the specified waiting time, unless the permit is available
void parkNanos(Object blocker, long nanos)	Disables the current thread for thread scheduling purposes, for up to the specified waiting time, unless the permit is available
void parkUntil(long deadline)	Disables the current thread for thread scheduling purposes, until the specified deadline, unless the permit is available
void parkUntil(Object blocker, long deadline)	Disables the current thread for thread scheduling purposes, until the specified deadline, unless the permit is available
void unpark(Thread thread)	Makes available the permit for the given thread, if it was not already available

2.4.1.1 Thread Parking in J9

IBM's J9 JVM implements the parking functionality by handling thread parks and unparks as described in the JUC `LockSupport` class [4].

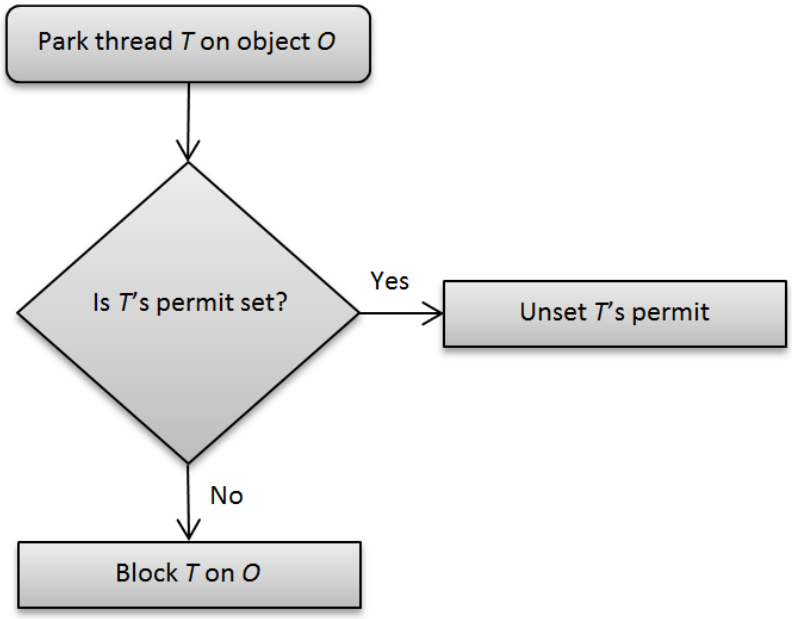


Figure 2.1: LockSupport Thread Parking

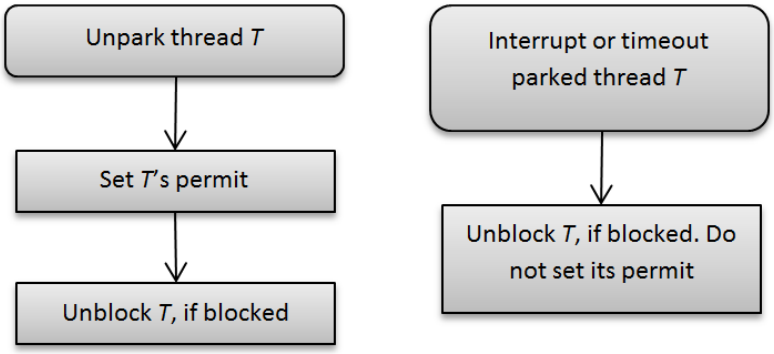


Figure 2.2: LockSupport Thread Unparking and Interrupting

2.4.2 JUC Synchronizers

A number of thread synchronization mechanisms are provided by the JUC library [11, 4]. These programming constructs enable proper communication

and timing between threads while avoiding the multi-thread hazards discussed in Section 2.3.1. Besides the popular Locks, a number of other JUC synchronizers are available which include Barriers, Latches, Semaphores, Exchangers, Phasers etc. The suspension of threads on these synchronizers takes place by using the `LockSupport` thread parking class.

2.4.2.1 Locks

Locks are programming tools used for synchronization purposes. They provide a means for only one thread at a time to operate within a code segment called the critical section and marked by a locking function call at the beginning and an unlocking one at the end [11]. Threads that are blocked on JUC locks are parked using the `LockSupport` thread parking class.

JUC provides a locking interface named `Lock` [4] as well as two classes which implement this interface called `ReetrantLock` and `ReentrantReadWriteLock` [11].

The `ReentrantLock` class provides an implementation of these locking functionalities and the `ReentrantReadWriteLock` expands it by providing a pair of locks used for read-write purposes, allowing many readers at a time inside the critical section but only one writer. Both classes handle locking in a reentrant manner which means that an acquire performed by a thread already owning that lock always immediately succeeds. Furthermore, a try-lock locking functionality is also provided which tries to acquire a lock and returns false, instead of blocking, if the lock is unavailable. Finally, the try-

lock functionality can be timed giving a certain time-frame to the thread to acquire the lock before returning false.

Consequently, JUC locks not only provide the standard mutual exclusion and reentrant locking also offered by the intrinsic Java locks, they also provide polled, timed and non-block formatted acquisitions, as well as read-write locks [11].

2.4.3 JUC Data Structures

A number of multithreaded data structures are provided by the JUC library [11, 4]. The emphasis is placed on providing tools which allow threads to work concurrently on the same data structure or block while waiting for data transferring; therefore, constructs like Concurrent Hash Maps and Blocking Queues are available. The blocking of threads on these data structures is performed using the `LockSupport` thread parking class.

2.4.4 JUC Task Management

The JUC library provides a set of tools for handling tasks, their execution and dispatching amongst available threads [11, 4]. Some of the available constructs for these tasks are Thread Pools, Fork-Join Tasks, Executors and Future Tasks.

2.5 Monitoring Tools

2.5.1 HealthCenter

HealthCenter [3] is an IBM monitoring program for the J9 JVM for several application areas which include performance, memory usage, system environment, Java class loading, file input/output (I/O), real time monitoring and object allocations. In particular, the performance aspect of HealthCenter profiles Java methods, analyses Java intrinsic locks, measures the impact of the GC on the system and monitors thread contention. It is important to note that HealthCenter does not provide monitoring information for JUC locks or any other parking related metric of JUC classes.
Concerning intrinsic lock profiling in particular, Healthcenter provides the following metrics for inflated, i.e. contended, locks [3]:

- **Gets**, the total number of times the lock has been taken while it was inflated.
- **%miss**, the percentage of the total Gets, or acquires, for which the thread trying to enter the lock on the synchronized code had to block until it could take the lock.
- **Recursive**, the total number of recursive acquires. A recursive acquire occurs when the requesting thread already owns the monitor.
- **Slow**, the total number of non-recursive lock acquires for which the

requesting thread had to wait for the lock because it was already owned by another thread.

- **%util**, the amount of time the lock was held, divided by the amount of time the output was taken over.
- **Average hold time**, the average amount of time the lock was held, or owned, by a thread. For example, the amount of time spent in the synchronized block, measured in processor clock ticks.
- **Name**, the monitor's name.

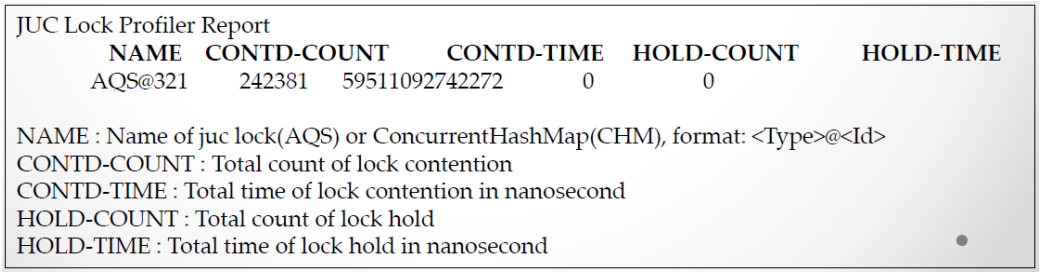
2.5.2 Multicore Software Development Kit (MSDK)

Multicore Software Development Kit (MSDK) for Java is a set of tools that developers can use to test, debug and analyze applications targeted for multicore hardware systems. Besides its more general features, MSDK contains tools for lock analysis and synchronization coverage analysis [6].

MSDK provides limited metrics for explicitly used JUC locks or when part of the JUC data structure `ConcurrentHashMap`. However, their approach to the situation was to modify the JUC library itself so, since it has not been updated since November 2011, it no longer functions properly with newer versions of Java.

An example of the metrics produced by MSDK is displayed in Figure 2.3 while trying to profile a very simple application with a JUC lock. The provided metrics are not enough to fully evaluate the impact that a JUC lock has on the

code. Furthermore, two of the metrics are unexpectedly zero, which should be attributed to the non compatibility of MSDK with the tested version of Java.


JUC Lock Profiler Report

NAME	CONTD-COUNT	CONTD-TIME	HOLD-COUNT	HOLD-TIME
AQS@321	242381	59511092742272	0	0

NAME : Name of juc lock(AQS) or ConcurrentHashMap(CHM), format: <Type>@<Id>
CONTD-COUNT : Total count of lock contention
CONTD-TIME : Total time of lock contention in nanosecond
HOLD-COUNT : Total count of lock hold
HOLD-TIME : Total time of lock hold in nanosecond

Figure 2.3: Example of the Printout of MSDK for JUC Lock Profiling.

2.6 Benchmarking Tools

Benchmarks are applications which stress the software and the hardware of a system and then produce a score which can be used to compare amongst different packages. Different JVMs have varying performances; therefore, a benchmarking tool is needed to rank the different JVMs efficiency or even to find out if changes in a particular JVM have positive, negative or neutral effects.

2.6.1 SPECjbb2013

SPECjbb2013 is a benchmark that has been developed to measure performance based on the latest Java application features. It is relevant to all audiences who are interested in Java server performance, including JVM

vendors, hardware developers, Java application developers, researchers and members of the academic community [7].

The benchmark has two metrics. The first one, called `max-jOPS`, aims to measure the maximum throughput of the system; the second, called `crit-JOPS`, aims to measure throughput under a time response constraint. Comparisons can be done on any of the two but results from both are needed for acceptable publications [8].

2.6.2 DaCapo Benchmark suite

This benchmark suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory loads [1]. The metric used for this benchmark is the execution time of the specific application of the suite used.

2.7 Context of Research

Interest in profiling is high in the literature and in this section we investigate relevant work in the area.

2.7.1 Lock Contention Analysis

A lock analysis method has been introduced in [14]. The authors propose three gradually improving strategies to accurately profile locks. Each strat-

egy is an improvement of the previous one, specifically targeting its deficiencies. The proposed approaches start by blaming the victims, i.e. the threads that get blocked on a lock, continue by placing the blame on the suspects, i.e. all the threads running on a critical section, and conclude by attributing idleness and contention to the actual perpetrators, i.e. threads running on the critical section of a particular lock.

All of those approaches rely on the periodic sampling of the application to determine in which function it runs at a given point in time. In the third and most relevant approach, counters are incremented per lock for each spinning/blocking sample and the total amount is attributed to the thread releasing the lock.

2.7.2 Critical Lock Analysis

A critical lock analysis method has been proposed in [9]. The authors initially show how traditional lock profiling techniques are inefficient as they can point to locks whose critical sections are not a significant part of the critical path of the application and optimizing them yields low or no speedups.

To this end, a new method is suggested which only monitors hot locks, i.e., locks that have critical sections on the critical path. Two metrics are proposed to measure the impact hot locks have on the performance of the application, contention probability and size of hot critical section.

2.7.3 Scalability Bottleneck Analysis

A contention analysis method has been proposed in [10]. The authors suggest that the main reasons which prevent applications from scaling linearly when the number of threads rises are locking, memory bandwidth and cache contention. In order to better guide the developer/optimizer to which of these factors has the greatest impact, percentages are provided for each one and the results are displayed on speedup stacks. The speedup stacks provide, in a graphical way, the impact each bottlenecking factor has on the application's performance. It is also important to note that this method is only suitable for symmetric, fork-join and data-parallel applications due to the way it operates when gathering results.

In order to perform this analysis, the application is repeatedly executed by initially eliminating all bottlenecking factors and then gradually adding them again one at a time.

2.7.4 Relevance of Research Context

The papers presented in the current section explore new ideas and methods on profiling concurrent applications. The emphasis is mostly placed on locks although other bottlenecking factors, such as cache misses and memory contention, are also explored. We are investigating the idea of providing a thread-parking profiling tool for JUC locks and other JUC constructs, which places our work in the same research area of multi-threaded profiling.

2.8 Summary

So far we have introduced the reader to Java and its underlying running mechanism implemented by a program referred to as the Java Virtual Machine (JVM). Java is a language with embedded multi-threaded support but working with it directly can lead to programming errors when used carelessly. However, the Java Util Concurrent (JUC) library is provided as the standard extension for multi-threaded features; a library which not only provides the user solutions for a number of problems but also enforces higher levels of thread safety, avoiding a number of concurrency hazards.

One feature of concurrency is the need for blocking threads for some period of time. This is performed in JUC by using thread parking. We propose a profiling technique for capturing thread parking contention for JUC classes, implement it in IBM's J9 JVM and investigate the potential benefits provided by this new tool to multi-threaded Java development and optimization.

Chapter 3

Capturing Parking Contention Data

3.1 Introduction

In this chapter we present our park contention tool and the necessary modifications that had to be performed in the J9 JVM. More specifically, we discuss the underlying thread parking functionality in the J9 JVM, the parameters required to activate our monitoring tool, the high-level architectural design of our modifications, the algorithms and data structures used, the provided thread-park related metrics of our tool and finally how the output of our tool is presented to the user.

3.2 Thread parking in the J9

The J9 JVM contains functions which perform the parking functionality of the `LockSupport` Java class. In particular there is one function which implements thread parking and a second which performs thread unpark. We have modified the function which supports thread park to measure parking contention statistics at runtime.

3.3 JVM parameters

In order to have our tool optionally run at the JVM's startup, we modified J9 to check for the parameter `-XXgetparkcontention`. When this parameter is set, our measuring tool initiates JUC parking profiling and reports all the collected results in the standard error file at the end of the application's execution or whenever a Java dump is requested. The general format of our parameter is the following:

$$-XXgetparkcontention[: \textbf{OptionList}]$$

With:

$$\textbf{OptionList} \rightarrow \textbf{Option}[, \textbf{OptionList}]$$

$$\textbf{Option} \rightarrow threshold = value_{threshold}$$

$$\textbf{Option} \rightarrow printThreshold = value_{printThreshold}$$

$$\textbf{Option} \rightarrow freeOnPrint$$

For example, the following could be a valid invocation of the J9 JVM with our tool activated:

```
java -XXgetparkcontention:freeOnPrint,printThreshold=100 ...
```

All values for the options are non-negative integers and are stored in the main data structure of the tool. The effect these options have to the execution of our profiling tool is explained in the remainder of this section.

3.3.1 Collection Threshold

The collection threshold provides an efficient way to minimize overhead when large numbers of short-lived park blockers are used. The $value_{threshold}$ decides how many parking events need to take place on a park blocker before we start collecting park contention data for it. When omitted, the default value is zero; therefore, parking contention data is collected from the first park for all park blockers.

3.3.2 Printing Threshold

The printing threshold provides a way of shortening the reports generated by our tool by filtering all the records with number of parks less than $value_{printThreshold}$. When omitted, the default value is zero; therefore, parking contention data is reported for all park blockers regardless of the number of recorded parking events.

3.3.3 Free-on-print Mode

When the option `freeOnPrint` is set, parking records of dead objects, i.e. objects that have been cleaned by the GC, are deleted after a report has been produced. This feature helps lower the overhead when many short-lived park blockers are used. When omitted, no deletion of park records takes place; thus, this mode is off by default and all parking records are kept throughout the execution of the JVM.

3.4 Overview of Modifications

To capture park contention data we need to associate each Java park-blocking object with a relevant parking record. The parking records are connected in a linked-list fashion for fast traversal. Furthermore, to reduce overhead, we use an extended parking record for each parking record which is allocated and updated after the collection threshold has been reached. Moreover, because each park blocker might have a different size depending on its class, the exact place where the pointer to a parking record will be stored varies. To accommodate that variance, we store park blocker Java classes and their required offset for each case in a linked list accessible from the central data structure of the tool.

In Figure 3.1 we present an illustration of the high level architecture of our modifications to the JVM.

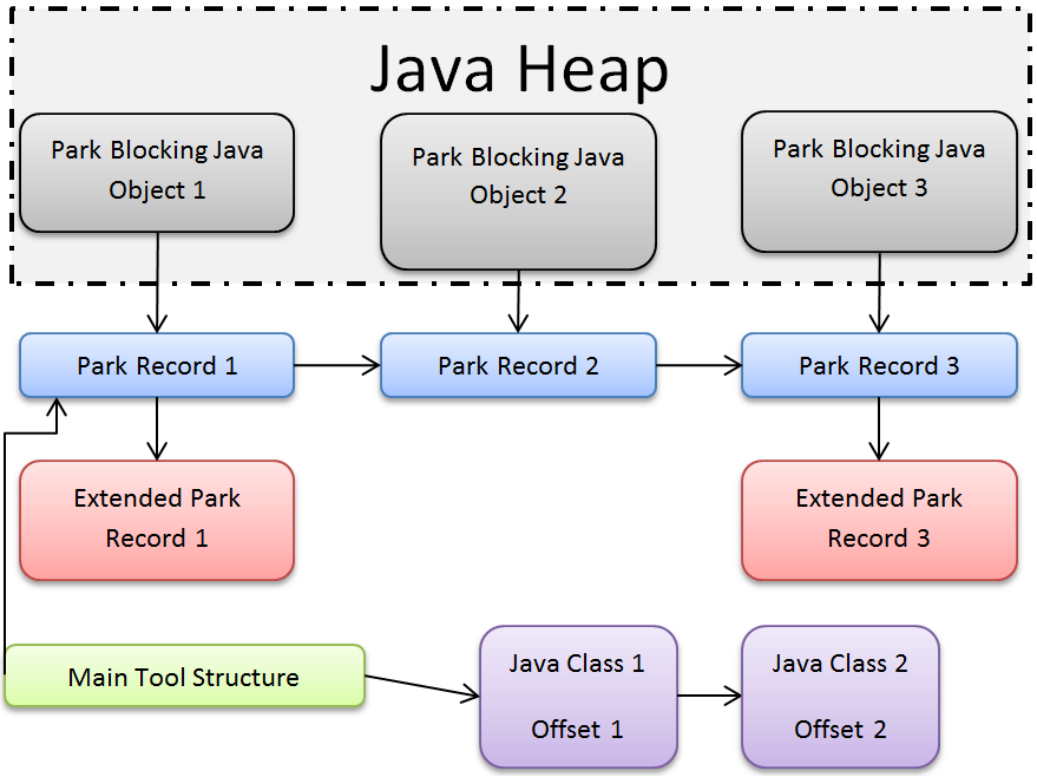


Figure 3.1: High Level Tool Architecture

3.5 Data Structures

In order to store and update parking contention data, we attach one data structure per monitored park-blocking object, also referred to as a park blocker, which contains the following:

- A pointer to the next parking record.
- The number of thread parks that this object has handled.
- The number of threads currently parked on that object.

- A flag indicating whether this park blocker is dead. The flag is updated by the GC.
- A lock to prevent race conditions arising from concurrent updates.
- A pointer to the extended parking record.

Furthermore, each parking record points to an extended parking record storing the following:

- The name of the thread of the first monitored parking event.
- The stack trace of the thread of the first monitored parking event.
- The class name of the park blocker.
- The hash code of the park blocker.
- The real time at least one thread was parked on that object. Also referred to as parking real-time.
- The current maximum number of threads parked concurrently on that object.
- The total thread time threads have spent parked on that object. Also referred to as parking thread-time.
- The timestamp of the last parking or unparking event.
- The timestamp of the first monitored parking event.

3.6 Algorithms

Our algorithms were designed with thread-safety, efficiency and overhead minimization in mind. In particular, we use synchronization techniques to avoid any race conditions and we keep a small number of data in our data structures while deriving the rest of them after the measuring phase. Furthermore, we perform most of the work right before the parking of a thread and by doing so we are not heavily influencing the critical path of the application because the about to be parked thread is not part of it.

3.6.1 Initialization

The first time an object is used to park a thread we allocate a new data structure for it, initialize it accordingly and associate it with that object. A second allocation and initialization takes place when the parking data collection starts which, as explained in Section 3.3.1, can potentially be at a different time.

3.6.2 Parking Events

Right before the parking of a thread takes place we execute the following steps:

- Increment the thread parks counter.
- Increment the concurrently parked threads counter.

- Check if the park collection threshold has been reached and:
  - If it is the first time:
    - Save the current timestamp
    - Store the thread's name
    - Capture the thread's stack trace
    - Store the park blocker's class name
  - Update the total parking thread-time.
  - Update the parking real-time.
  - Update the last event timestamp.
  - Check and potentially update the maximum number of concurrently parked threads counter.

3.6.3 Unparking Events

Right after a thread is unparked we execute the following steps:

- Decrement the concurrently parked threads counter.
- Check if the park collection threshold has been reached and:
  - Update the parking thread-time.
  - Update the parking real-time.
  - Update the last event timestamp.

3.7 Metrics

Our reporting tool provides data which is explained in the following subsections. The data is collected per different park-blocking object. The updates of the data take place at every parking or unparking event, depending on what that metric measures.

3.7.1 Thread Name

When the collection threshold for a park-blocking object is reached, we capture the invoking thread's name.

3.7.2 Stack Trace

When the collection threshold for a park-blocking object is reached, we capture the invoking thread's stack trace. It is important to note that the same parking object can block threads later at different code paths but the entire object's thread parking measurements will be aggregated in the final report.

3.7.3 Class Name

This metric is the name of the class of the park-blocking object. We measure parking contention data for all objects belonging to or extending the following classes:

- `java/util/concurrent/locks/AbstractOwnableSynchronizer`

- `java/util/concurrent/locks/AbstractQueuedSynchronizer $ConditionObject`
- `java/util/concurrent/Exchanger$Node`
- `java/util/concurrent/SynchronousQueue$TransferStack`
- `java/util/concurrent/LinkedTransferQueue`
- `java/util/concurrent/ForkJoinPool`
- `java/util/concurrent/Phaser`

3.7.4 Hash Code

Every Java object has a hash code and this metric provides the park-blocking object's hash code. Alongside the class name, it can be used to match Java objects with park contention results when the stack trace is ambiguous.

3.7.5 Times Parked

This metric provides the total number of times a thread was parked on that park-blocking object.

3.7.6 Parked Now

The metric shows how many threads are parked at a specific moment on that object. It is usually zero at the end of the execution of the application

but can vary if the data is collected by a core dump in the middle of the execution.

3.7.7 Peak Parked

Peak parked provides the maximum number of threads concurrently parked on that park-blocking object. It is calculated by the following formula:

$$PeakParked = max\{ParkedNow_i | \forall i \leq \#parkingOrUnparkingEvents\}$$

3.7.8 Parking Thread-Time

This metric provides the total thread time spent parking on that object in milliseconds. More specifically, it is calculated by the following formula, where $Time_i$ is the timestamp of the i_{th} monitored thread parking-unparking event and n the total number of those events:

$$ParkingThreadTime = \sum_{i=2}^{n} ParkedNow_{i-1} \times (Time_i - Time_{i-1})$$

The $ParkedNow$ counter is first used in the calculation and then updated. This is the reason we use $i - 1$ index, indicating the previous value of that metric.

In Figure 3.2 a scenario of thread parking on an object is displayed. The Parking Thread-Time metric is the area of the function displayed in that

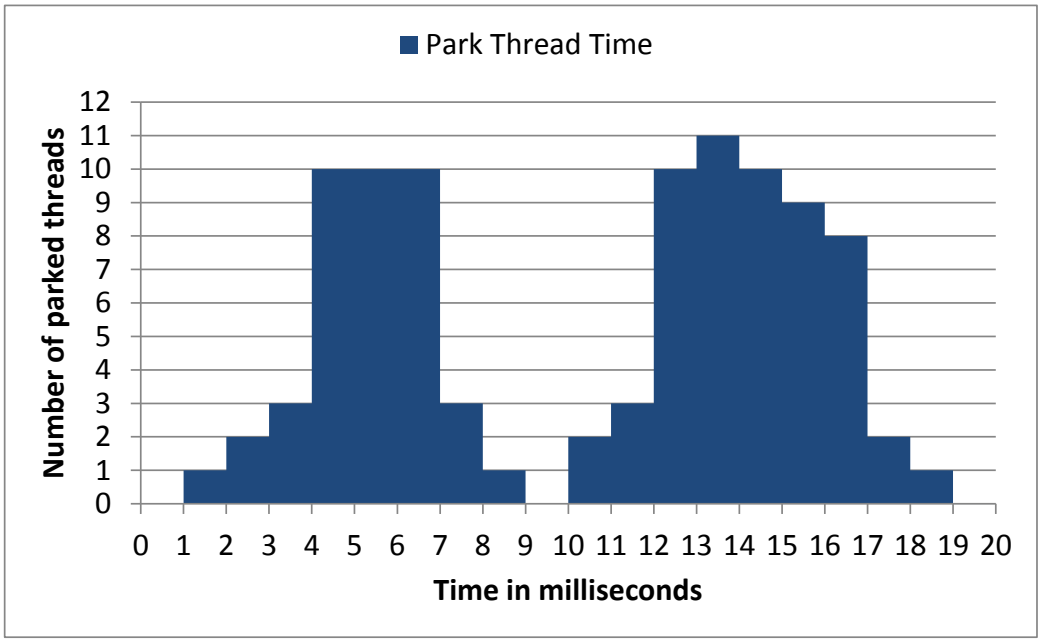


Figure 3.2: Parking Thread-Time Metric

graph.

3.7.9 Parking Real-Time

This metric provides the real time spent with at least one thread parked on that object in milliseconds. More specifically, it is calculated by the following formula, where $Time_i$ is the timestamp of the i_{th} monitored thread parking-unparking event and n the total number of those events:

$$ParkingRealTime = \sum_{i=2}^{n} min\{ParkedNow_{i-1}, 1\} \times (Time_i - Time_{i-1})$$

The $ParkedNow$ counter is first used in the calculation and then updated.

This is the reason we use the $i-1$ index, indicating the previous value of that metric.

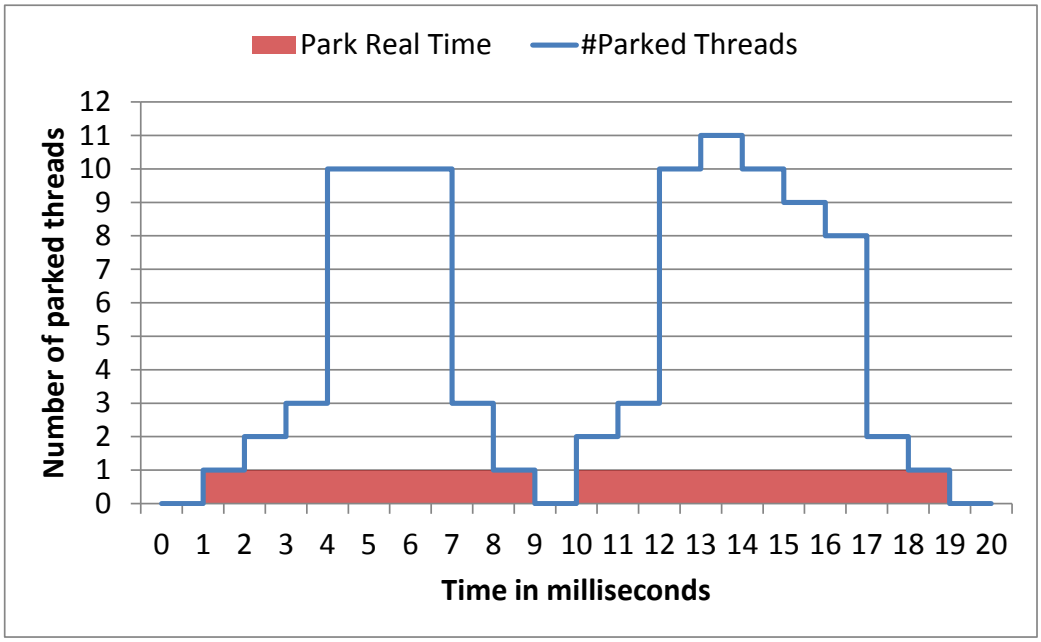


Figure 3.3: Parking Real-Time Metric

In Figure 3.3 a scenario of thread parking on an object is displayed. The Parking Real-Time metric is the colored area displayed in that graph.

3.7.10 Average Parking Time

This metric provides the average time a thread was parked on that park-blocking object. It is derived by the following formula:

$$AverageParkingTime = \frac{ParkingThreadTime}{TimesParked}$$

The reader should keep in mind that the Parking Thread-Time and not the Parking Real-Time must be used to get the average per thread because the latter is not taking into consideration the number of parked threads but only if at least one is parked.

3.7.11 Average Hold Time

This metric estimates the average time a thread's parking permit is unavailable. For locks in particular, it should be considered the average time a lock is held by a thread.

To measure this metric, we estimate the rate at which threads are unparked from the park-blocking object which is equivalent to the ratio of total time the permit was held over the number of unparks performed on that object. Although we are not able to know the exact hold time of the permit (because that would require us knowing when permits are consumed and released even when that does not lead to parking or unparking of threads, something handled by the JUC class `ParkSupport` without invoking the JVM's thread park support functionalities), we estimate it to be equal to the total time at least one thread is parked on that park blocker. Therefore, we suppose the hold time to be approximated by the Parking Real-Time metric presented in Section 3.7.9.

Furthermore, there is no need to keep track of the times threads were unparked on a park blocker because we can infer it by subtracting the number of threads parked now from the total number of parks. Consequently, the

following formula is used to calculate the metric:

$$AverageHoldTime = \frac{HoldTime}{TimesUnparked} = \frac{ParkingRealTime}{TimesParked - ParkedNow}$$

We display an example of the calculation of this metric in Figure 3.4. As presented in the chart, the Average Hold Time changes each time an unpark occurs. While no threads are parked on that object, its Average Hold Time does not change because at this point we estimate that the permit is not used at all. We need to remind the reader at this point that this is a derived metric and there is no need for it to be calculated at run-time; we only show it as such in the presented figure for illustration and explanatory purposes.

3.7.12 Real-Time Utilization

With this metric we provide the percentage of real-time at least one thread was parked on that park blocker in respect to the total execution time of the application. We calculate it using the following formula:

$$\%RealTimeUtil = 100 \times \frac{ParkingRealTime}{TotalExecutionTime}$$

3.7.13 Thread-Time Utilization

This metric calculates the percentage of thread-time threads were parked on that object when compared to the total execution time of the application. It

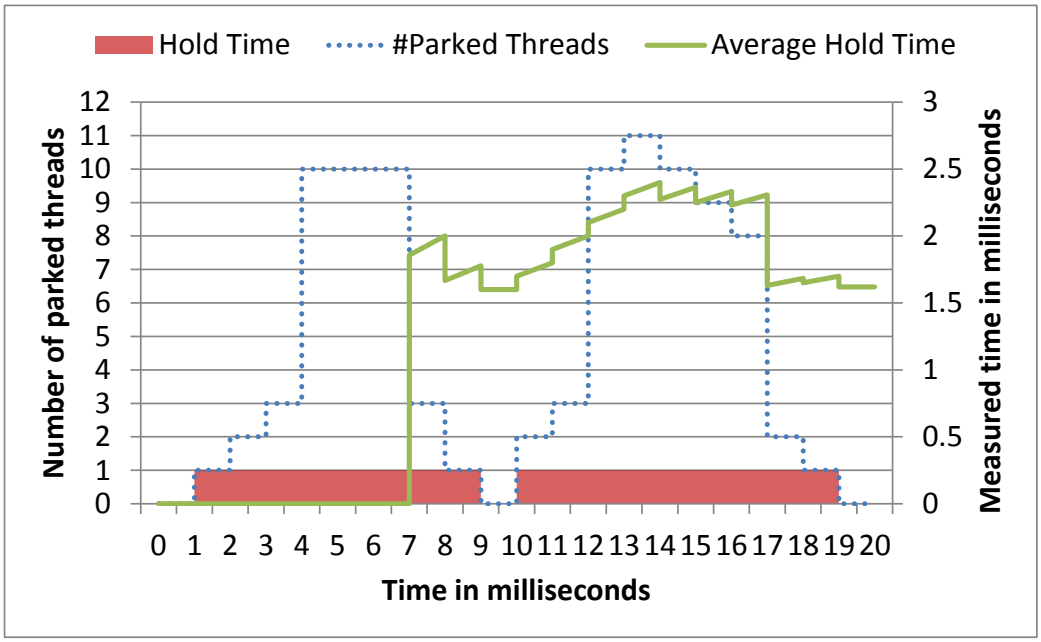


Figure 3.4: Average Hold Time Metric

is derived as such:

$$\%ThreadTimeUtil = 100 \times \frac{ParkingThreadTime}{TotalExecutionTime}$$

3.7.14 Real-Life-Time Utilization

With this metric we provide the percentage of real time at least one thread was parked on that park blocker with respect to the total time park data were collected for that object. The metric targets segmented applications which use park blockers for a time shorter than the lifetime of the program. We calculate it using the following formula, where $Time_i$ is the timestamp of the i_{th} monitored thread parking-unparking event and n the total number

of those events:

$$\%RealLifeTimeUtil = 100 \times \frac{ParkingRealTime}{Time_n - Time_1}$$

3.7.15 Thread-Life-Time Utilization

This metric calculates the percentage of thread time threads were parked on that object when compared to the total time park data was being collected for that object. The metric targets segmented applications which use park blockers for a time shorter than the lifetime of the program. It is derived as defined in the following formula, where $Time_i$ is the timestamp of the i_{th} monitored thread parking-unparking event and n the total number of those events:

$$\%ThreadLifeTimeUtil = 100 \times \frac{ParkingThreadTime}{Time_n - Time_1}$$

3.8 Getting the Results

At the end of execution of the application or when a Java dump is requested we print a table with all the metrics described in Section 3.7 for all the parking records with the number of parks exceeding the printing threshold. The data is printed on a special file for Java dumps, alongside the other data provided by the dump and at the standard error at the end of the application's execution.

Thread name	Stack trace	Class name
Thread-6	sun/misc/Unsafe(park) [...] jucmicro4/JUCMicro4$TestThread(run:196)	java/util/concurrent/locks/ReentrantLock$FairSync
Thread-9	sun/misc/Unsafe(park) [...] jucmicro4/JUCMicro4$TestThread(run:196)	java/util/concurrent/locks/ReentrantLock$FairSync

Hash code	Times parked	Peak parked	Park thread time	Park real time	Avg park time
A0BF29C	3224.00	64.00	6260087.00	100028.00	1941.71
BE831856	164.00	10.00	6721.00	4219.00	40.98

Avg hold time	Parked now	%Real time util	%Thread time util	%Real life time util	%Thread life time util
31.03	0.00	0.98	61.35	1.00	62.58
25.73	0.00	0.04	0.07	0.04	0.07

Figure 3.5: Example of Collected Metrics from the Standard Error

```
[...]
1PARKINFO
1PARKINFO1 Class - java/util/concurrent/locks/ReentrantLock$NonfairSync
1PARKINFO2 Hash Code - B751B896
1PARKINFO3 Park Summary - Times parked  Peak parked Parking thread-time Parking    real-time
        Average parking time       Average hold time   Parked now  %%Real-time util    %%Thread-
time util       %%Real-life-time util       %%Thread-life-time util
1PARKINFO4       90     63     71324 1784  792.488889  63.714286    62     81.797341%
        3270.243008%       100.000000%       3997.982063%
1PARKINFO5 First Thread Name - Thread-6
1PARKINFO6 Stack trace
2PARKTRACE sun/misc/Unsafe(park)
2PARKTRACE java/util/concurrent/locks/LockSupport(park:197)
2PARKTRACE java/util/concurrent/locks/AbstractQueuedSynchronizer(parkAndCheckInterrupt:845)
2PARKTRACE java/util/concurrent/locks/AbstractQueuedSynchronizer(acquireQueued:878)
2PARKTRACE java/util/concurrent/locks/AbstractQueuedSynchronizer(acquire:1208)
2PARKTRACE java/util/concurrent/locks/ReentrantLock$NonfairSync(lock:225)
2PARKTRACE java/util/concurrent/locks/ReentrantLock(lock:301)
2PARKTRACE jucmicro3/JUCMicro3$Lock3(lock:181)
2PARKTRACE jucmicro3/JUCMicro3$TestThread(JUC:306)
2PARKTRACE jucmicro3/JUCMicro3$TestThread(run:268)
[...]
```

Figure 3.6: Example of Collected Metrics from a Java Dump File

In Figure 3.5 we display an example of collected park contention data taken from the standard error using one testing application. Furthermore, in Figure 3.6 we display part of the output of our tool in a Java dump file again using a testing application.

3.9 Summary

In this chapter we introduced the features of our tool, its architectural design and the metrics it provides. In particular, we examined how the J9 JVM provides support for the thread parking functionality, described the parameters and options of our tool, presented the architecture, algorithms and data structures we used, defined all the park related provided metrics and discussed the available output options for the measurements of results. In the following chapter we continue by investigating the impact our changes have on the JVM concerning performance and memory requirements.

Chapter 4

Tool Overhead

4.1 Introduction

Changes to the code of any application usually have an impact on its performance. In our case, adding profiling features to the J9 JVM is also expected to create an extra overhead to the application's performance. However, it should also be expected that no statistically significant difference in performance occurs when the monitoring is switched off. Therefore, our analysis is split into two parts. First, we investigate the overhead in our modified VM with our monitoring tool switched off; second, with the monitoring tool switched on. The basis for measuring the overhead is the unmodified JVM on which we based our changes. Furthermore, we also examine the overhead of the free-on-print mode of the tool.

All of the VMs were compiled by using the same compilers and compilation

settings. Moreover, all test runs were performed on the same machine. These ensure that the results of our experiments are meaningful and comparable.
As a benchmarking tool we used SPECjbb2013 and we ran multiple iterations for each presented case on an AMD64 architecture machine with 16 physical and 32 logical cores, running Linux. The arguments for running the benchmark in all runs of all cases were `-Xmx8g -Xms8g -Xaggressive -Dreflect.cache=boot -Dcom.ibm.crypto.provider.doAESInHardware=true -Dcom.ibm.enableClassCaching=true -jar specjbb2013.jar -m COMPOSITE -ikv`. Both throughput and resident memory overheads were investigated and our results are discussed in the remainder of this chapter.

4.2 Statistical Metrics

SPECjbb2013 provides two throughput metrics: max-jops and crit-jops. For measuring the resident memory requirements, we periodically checked the reported number of pages in `/proc/<pid>/statm`.
We use the following metrics to aggregate our collected measurements, where t_i is the i_{th} measurement and N is the total number of iterations performed:

- Average:
$$\mu = \frac{1}{N}\sum_{i=1}^{N} t_i$$
- Standard Error:
$$SE = \frac{\sigma}{\sqrt{N}}$$

σ is the standard deviation of the sample and is calculated by the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \mu)^2}$$

- Relative Standard Error:

$$RLE\% = 100 \times \frac{SE}{\mu}$$

- Throughput Overhead:

$$ThroughputOverhead = 1 - \frac{\mu_{modified}}{\mu_{baseline}}$$

- Throughput Overhead with Positive Error:

$$ThroughputOverheadMax = max\left\{1 - \frac{\mu_{modified} \pm SE_{modified}}{\mu_{baseline} \pm SE_{baseline}}\right\}$$

- Throughput Overhead with Negative Error:

$$ThroughputOverheadMin = min\left\{1 - \frac{\mu_{modified} \pm SE_{modified}}{\mu_{baseline} \pm SE_{baseline}}\right\}$$

- Memory Overhead:

$$MemoryOverhead = \frac{\mu_{modified}}{\mu_{baseline}}$$

- Memory Overhead with Positive Error:

$$MemoryOverheadMax = max\left\{\frac{\mu_{modified} \pm SE_{modified}}{\mu_{baseline} \pm SE_{baseline}}\right\}$$

- Memory Overhead with Negative Error:

$$MemoryOverheadMin = min\left\{\frac{\mu_{modified} \pm SE_{modified}}{\mu_{baseline} \pm SE_{baseline}}\right\}$$

The reader should notice that the overhead definitions change between memory and throughput. That is the case because we are trying to associate positive overhead values with drops in the performance which happen inversely for those two metrics. More specifically, higher throughput but lower memory footprint are associated with better performance.

4.3 Throughput Overhead

Throughput is the most natural way of measuring performance as it can be directly mapped to achieved work per unit of time.

4.3.1 Baseline Measurements

For our baseline measurements, we compiled the unmodified VM and ran SPECjbb2013 on it for 80 iterations. The aggregated results are displayed in Table 4.1. Our level of uncertainty for these measurements is at most 0.2%.

Table 4.1: Aggregated Throughput Measurements for Unmodified VM

Mode	**Unmodified VM**
Average of max-jops	15212.7375
Standard Error of max-jops	25.19087598
Relative Error of max-jops	0.17%
Average of crit-jops	6552.7
Standard Error of crit-jops	12.99887495
Relative Error of crit-jops	0.20%

4.3.2 First Version

In the initial version of our tool, we extended all Java classes with an extra field used to store information on how to associate a park blocker with its respective parking record. All Java classes were extended, even those not monitored. In Figure 4.1 we provide an illustration of the technique used.

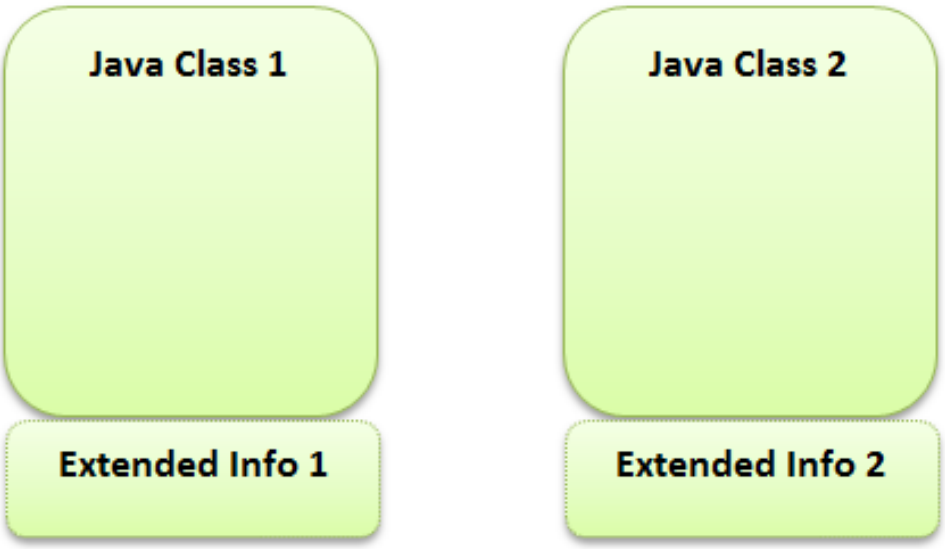


Figure 4.1: Illustration of First Approach

4.3.2.1 Modified VM with Measuring Switched Off

We compiled the first version of the modified VM and ran SPECjbb2013 on it for 32 iterations without using the parameter which would switch our tool on. The aggregated results are displayed in Table 4.2. Our level of uncertainty for these measurements is at most 0.34%. The overhead results are also displayed graphically in Figure 4.2.

Table 4.2: Aggregated Throughput Measurements for First Modified VM and Our Tool Off

Mode	**ModOff v1**
Average of max-jops	15231.75
Standard Error of max-jops	34.89593905
Relative Error of max-jops	0.23%
Average of crit-jops	6582.15625
Standard Error of crit-jops	22.12685856
Relative Error of crit-jops	0.34%
Overhead of max-jops	-0.12% (-0.52% $\sim$ 0.27%)
Overhead of crit-jops	-0.45% (-0.99% $\sim$ 0.09%)

Our results suggest that the measured overhead is comparable to zero; therefore, our modifications did not have any significant impact on the VM when the monitoring tool was not activated.

4.3.2.2 Modified VM with Measuring Switched On

We compiled the first version of the modified VM and ran SPECjbb2013 on it for 32 iterations using the parameter which switches on our measuring tool with its default arguments. The aggregated results are displayed in

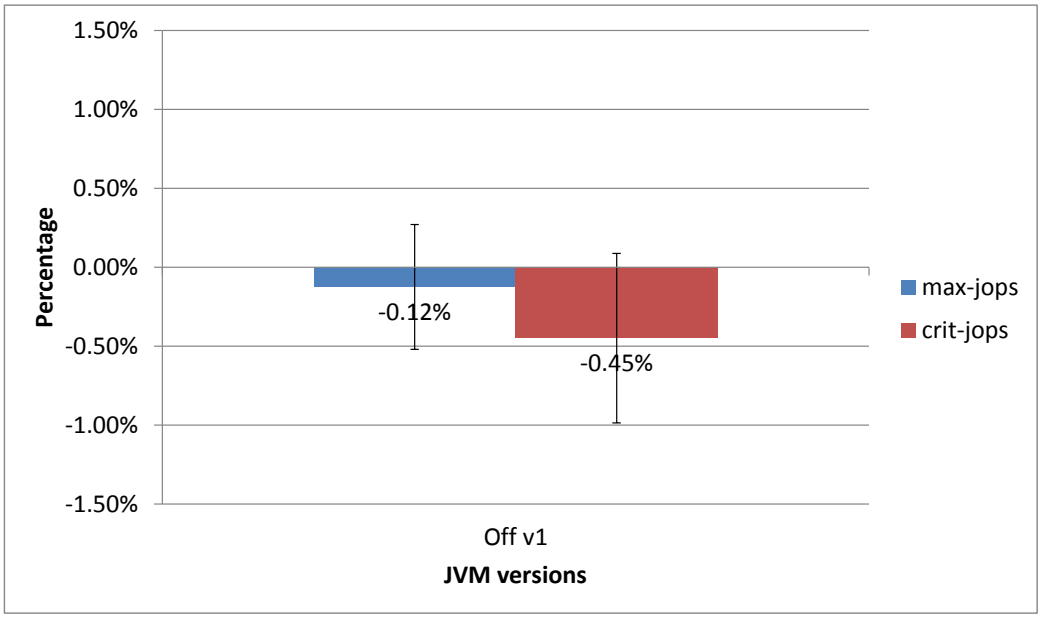


Figure 4.2: Overhead of First Version Off Versus Baseline

Table 4.3. Our level of uncertainty for these measurements is at most 0.24%.

The overhead results are also displayed graphically in Figure 4.3.

Table 4.3: Aggregated Throughput Measurements for First Modified VM and Our Tool Active

Mode	**ModOff v1**
Average of max-jops	15099.78125
Standard Error of max-jops	24.60757372
Relative Error of max-jops	0.16%
Average of crit-jops	6500.3125
Standard Error of crit-jops	15.53445248
Relative Error of crit-jops	0.24%
Overhead of max-jops	0.74% (0.42% $\sim$ 1.07%)
Overhead of crit-jops	0.80% (0.36% $\sim$ 1.23%)

Our results suggest that the measured overhead is in the area of 0.8%; therefore, our modifications had a measurable impact on the VM when the monitoring tool was running which can potentially lead to skewed results. To address this issue, we redesigned some aspects of our implementation which led to the second and final version of our tool. The performance overhead results of the second version are presented in the following subsection.

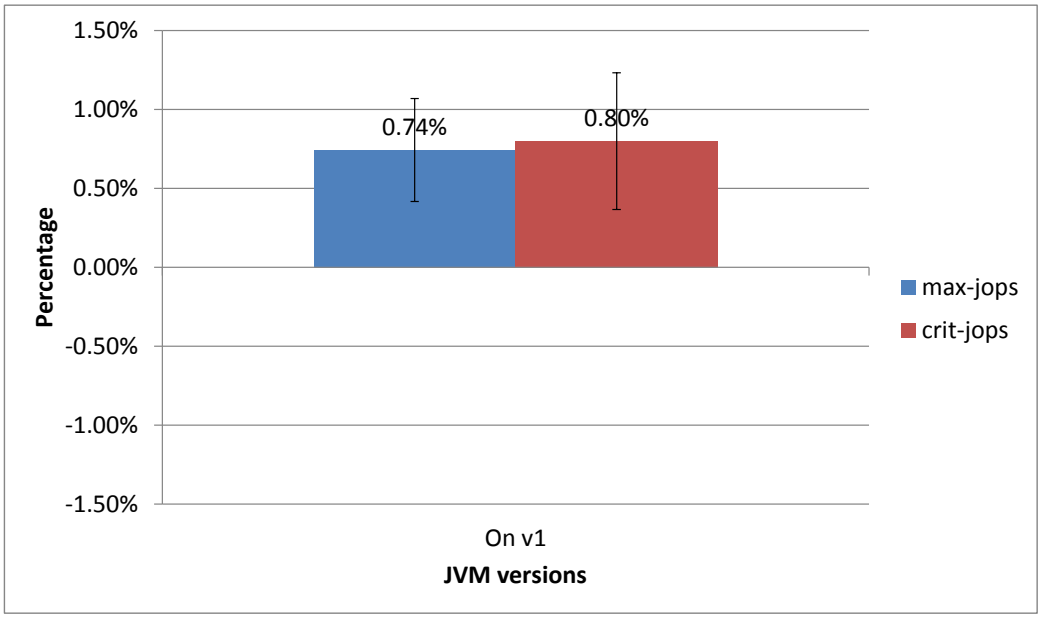


Figure 4.3: Overhead of First Version On Versus Baseline

4.3.3 Second version

In our second and final approach, we removed the extra field from the Java classes and instead created a linked list index which, for each monitored class, stores the information on how to associate a park blocker with its respective

parking record. In our tests, we found that around 10 records are expected to be stored in that index; therefore, a linked list is sufficient and there is no need to use more complex indexes, like AVL-trees, which would perform faster, were the number of records higher.

The second version was preferred instead of the first due to its reduced memory requirements and because it had lower performance overhead when turned on. In Figure 4.4 we provide an illustration of the technique used.

Figure 4.4: Illustration of Second Approach

4.3.3.1 Modified VM with Measuring Switched Off

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 32 iterations without using the parameter which would switch our tool on. The aggregated results are displayed in Table 4.4. Our level of uncertainty for these measurements is at most 0.32%. The overhead results are also displayed graphically in Figure 4.5.

Our results suggest that the measured overhead, is comparable to zero; therefore, our modifications did not have any significant impact on the VM when the monitoring tool is not running.

Table 4.4: Aggregated Throughput Measurements for Second Modified VM and Our Tool Off

Mode	**ModOff v2**
Average of max-jops	15164.03125
Standard Error of max-jops	35.9655457
Relative Error of max-jops	0.24%
Average of crit-jops	6548.096774
Standard Error of crit-jops	23.07870154
Relative Error of crit-jops	0.35%
Overhead of max-jops	0.32% (-0.08% $\sim$ 0.72%)
Overhead of crit-jops	-0.02% (-0.57% $\sim$ 0.53%)

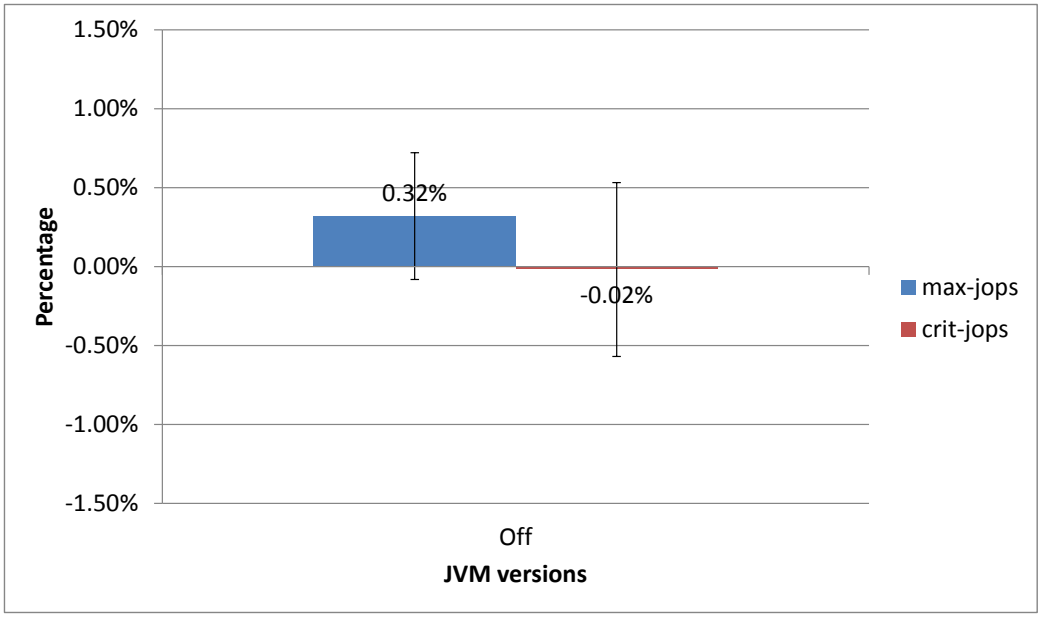


Figure 4.5: Overhead of Second Version Off Versus Baseline

4.3.3.2 Modified VM with Measuring Switched On

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 32 iterations using the parameter which switches our tool on and uses

its default arguments. The aggregated results are displayed in Table 4.5. Our level of uncertainty for these measurements is at most 0.33%. The overhead results are also displayed graphically in Figure 4.6.

Table 4.5: Aggregated Throughput Measurements for Second Modified VM and Our Tool Active

Mode	**ModOn v2**
Average of max-jops	15186.1875
Standard Error of max-jops	30.05351469
Relative Error of max-jops	0.20%
Average of crit-jops	6578.03125
Standard Error of crit-jops	21.45705578
Relative Error of crit-jops	0.33%
Overhead of max-jops	0.17% (-0.19% $\sim$ 0.54%)
Overhead of crit-jops	-0.39% (-0.91% $\sim$ 0.14%)

Our results suggest that the measured overhead is comparable to zero; therefore, our modifications did not have any significant impact on the VM when the monitoring tool is running and the possibility of skewing results due to interference is minimized.

4.3.3.3 Modified VM with Measuring Switched On and Free-on-print Mode Active

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 32 iterations. We used the parameter to turn our tool on and also switched the free-on-print mode on. The aggregated results are displayed in Table 4.6. Our level of uncertainty for these measurements is at most 0.35%.

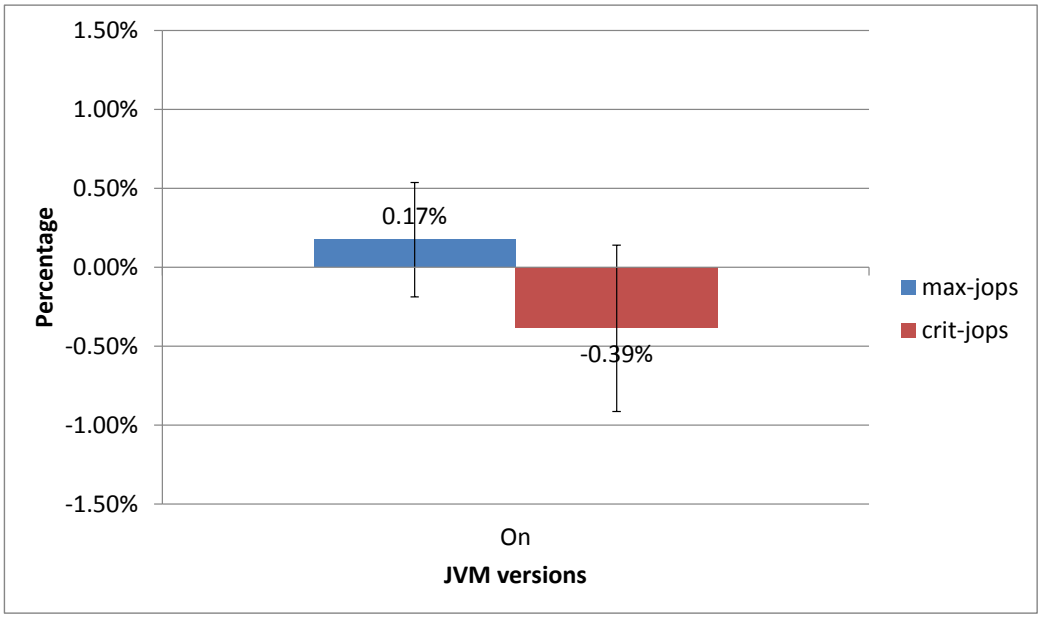


Figure 4.6: Overhead of Second Version On Versus Baseline

The overhead results are also displayed graphically in Figure 4.7.

Table 4.6: Aggregated Throughput Measurements for Second Modified VM and Our Tool Active with the Free-on-print Mode Active

Mode	**ModOn v2**
Average of max-jops	15159.4375
Standard Error of max-jops	31.49593654
Relative Error of max-jops	0.21%
Average of crit-jops	6558.6875
Standard Error of crit-jops	23.28321877
Relative Error of crit-jops	0.35%
Overhead of max-jops	0.35% (-0.02% $\sim$ 0.72%)
Overhead of crit-jops	-0.09% (-0.74% $\sim$ 0.55%)

Our results suggest that the measured overhead is comparable to zero; there-

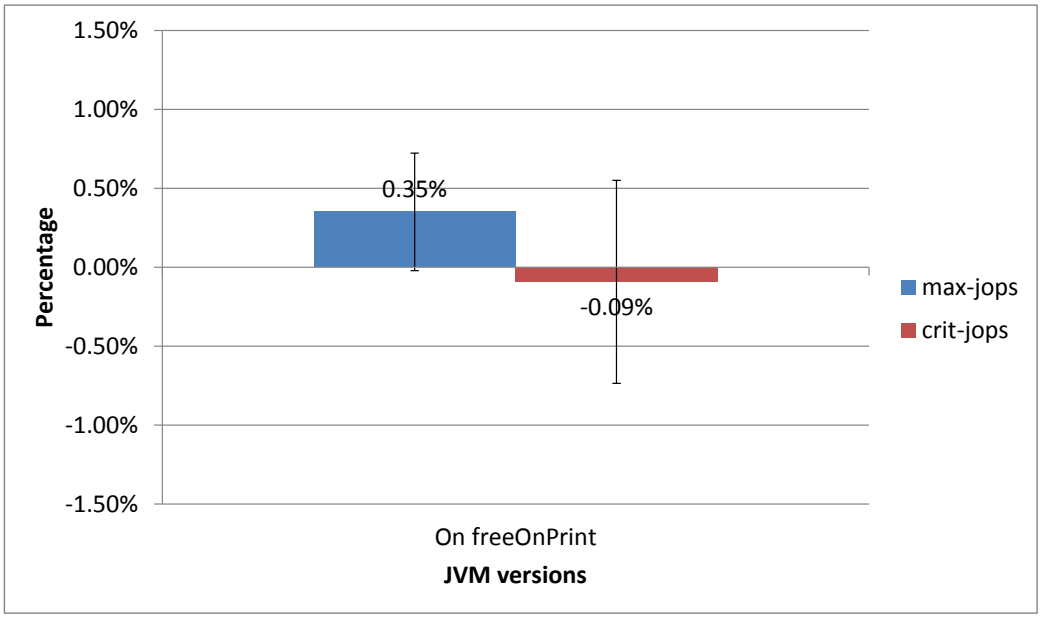


Figure 4.7: Overhead of Second Version On with Free-on-print Mode Enabled Versus Baseline

fore, our modifications did not have any significant impact on the VM when the monitoring tool is running with the free-on-print mode activated. Consequently, switching the free-on-print mode does not have a measurable impact on the JVM's throughput.

4.4 Resident Memory Overhead

Besides measuring the implications our modifications had on the throughput of the JVM, another important aspect that needs to be examined is the effect on memory footprint. To that end, we have measured the resident memory requirements of the J9 JVM while running the SPECjbb2013 benchmark. We

have tested the second and final version of our tool and compared our results to the unmodified and compiled by us JVM. As presented in the remainder of this section, the induced memory overhead of our tool is very low and almost comparable to 0.

In order to get the resident memory of the tested applications, we had a daemon script running on the background which read the number of resident memory pages of that application using its process id from the special file `/proc/<pid>/statm` every second. The daemon was sleeping between measurements so it did not cause any major disturbance to the running application. The daemon's bash code is displayed in Listing A.1.

4.4.1 Baseline Measurements

We compiled the unmodified VM and ran SPECjbb2013 on it for 8 iterations while having the resident memory daemon running. The aggregated results are displayed in Table 4.7. Our level of uncertainty for these measurements is 0.04%.

Table 4.7: Aggregated Resident Memory Measurements for Unmodified, Baseline VM

Mode	**unMod**
Average Resident Memory (Pages)	2,122,466.04
Standard Error of Average Resident Memory	910.6447433
Relative Error of Average Resident Memory	0.04%

4.4.2 Modified VM with Measuring Switched Off

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 8 iterations with our tool switched off while having the resident memory daemon running. The aggregated results are displayed in Table 4.8. Our level of uncertainty for these measurements is 0.05%. The results are also displayed graphically in Figures 4.8 and 4.9.

Table 4.8: Aggregated Resident Memory Measurements for the Second Version of the Modified VM and Our Tool Switched Off

Mode	**Off**
Average Resident Memory (Pages)	2,123,436.39
Standard Error of Average Resident Memory	1,078.785674
Relative Error of Average Resident Memory	0.05%
Average Resident Memory Overhead	0.05% (-0.05% $\sim$ 0.14%)

The results suggest no measurable resident memory overhead when our tool is not running and any changes are statistically insignificant.

4.4.3 Modified VM with Measuring Switched On

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 8 iterations with our tool switched on while having the resident memory daemon running. The aggregated results are displayed in Table 4.9. Our level of uncertainty for these measurements is 0.05%. The results are also displayed graphically in Figures 4.10 and 4.11.

The results show that the resident memory overhead is very low, around

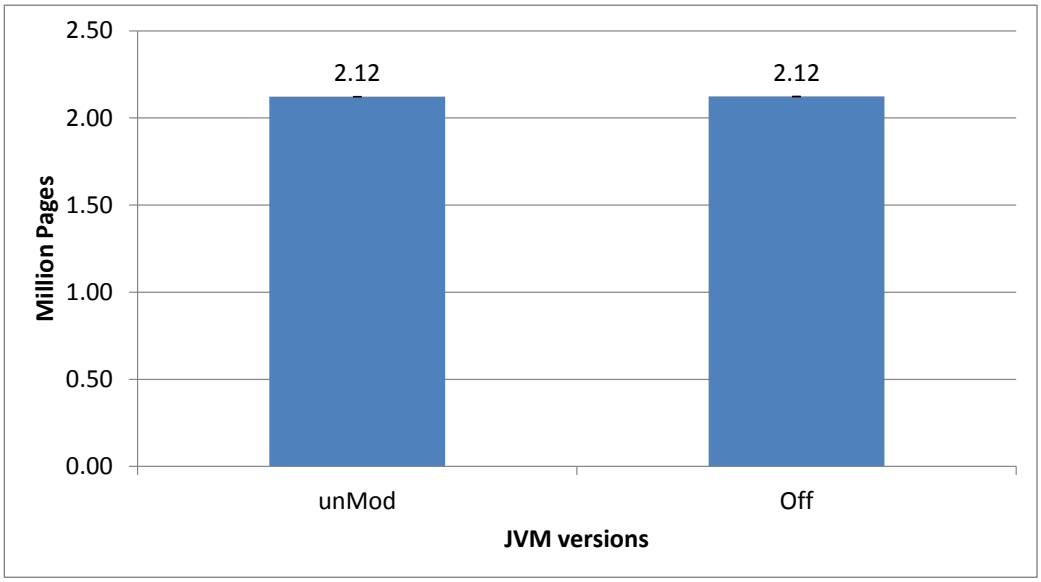


Figure 4.8: Resident Memory of Off Versus Unmodified JVM

Table 4.9: Aggregated Resident Memory Measurements for the Second Version of the Modified VM and Our Tool Switched On

Mode	**On**
Average Resident Memory (Pages)	2,128,199.00
Standard Error of Average Resident Memory	1,074.213322
Relative Error of Average Resident Memory	0.05%
Average Resident Memory Overhead	0.27% (0.18% $\sim$ 0.36%)

0.27%.

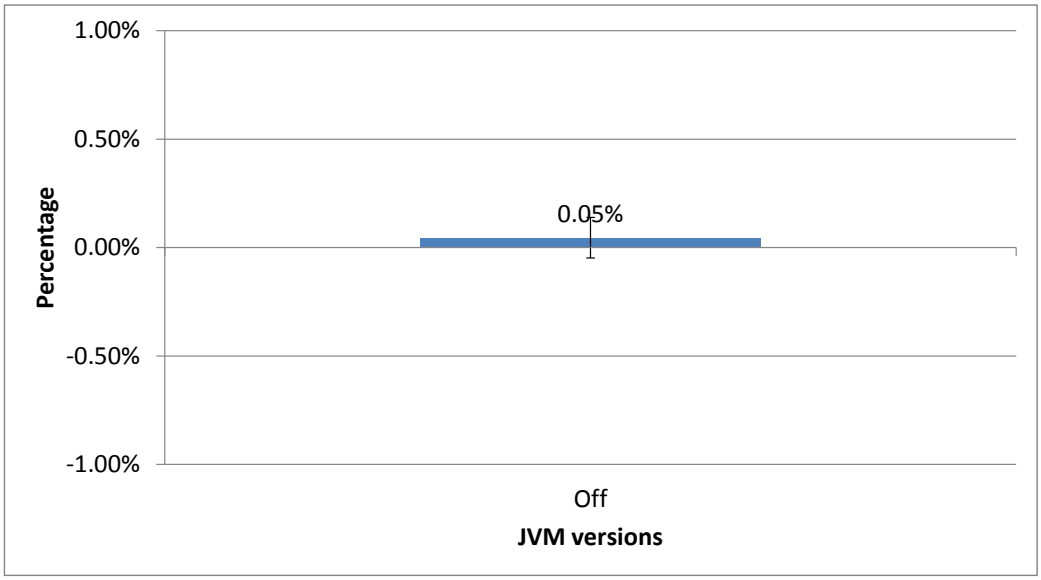


Figure 4.9: Overhead of Resident memory of Off Versus Unmodified JVM

4.4.4 Modified VM with Measuring Switched on and Free-on-print Mode Active

We compiled the second version of the modified VM and ran SPECjbb2013 on it for 8 iterations. Our tool was switched on and the free-on-print mode was activated while also having the resident memory daemon running. The aggregated results are displayed in Table 4.10. Our level of uncertainty for these measurements is 0.05%. The results are also displayed graphically in Figures 4.12 and 4.13.

The results show that the resident memory overhead is very low, almost comparable to 0, when our tool is running with the free-on-print mode switched on.

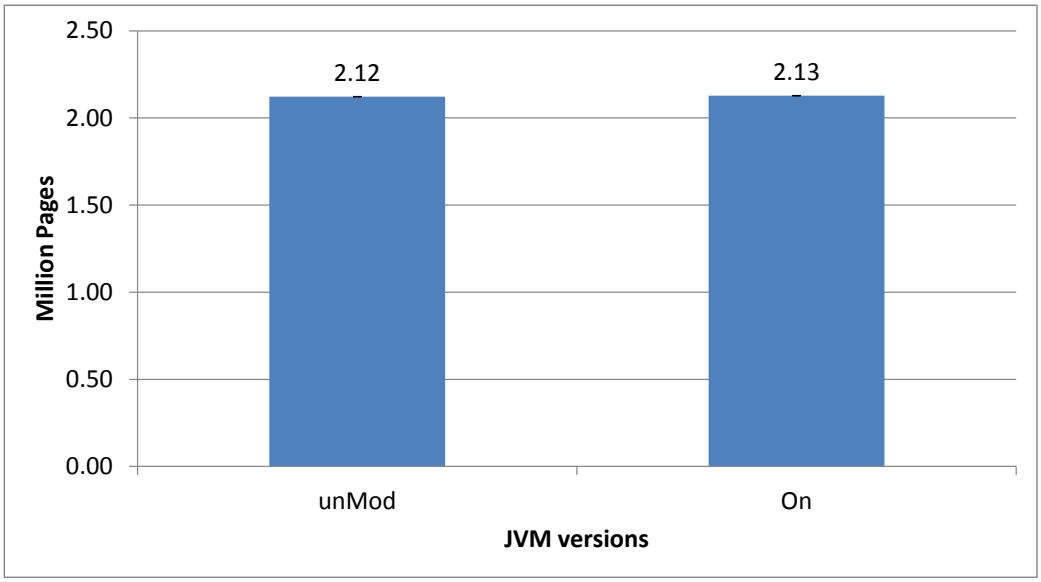


Figure 4.10: Resident Memory of On Versus Unmodified JVM

Table 4.10: Aggregated Resident Memory Measurements for the Second Version of the Modified VM and Our Tool Switched On with the Free-on-print Mode Active

Mode	**On Free-On-Print**
Average Resident Memory (Pages)	2,124,790.35
Standard Error of Average Resident Memory	1,090.539487
Relative Error of Average Resident Memory	0.05%
Average Resident Memory Overhead	0.11% (0.02% $\sim$ 0.20%)

4.5 Summary

In this chapter we have investigated the impact our modifications had on the J9 JVM. In order to have comparable results, our baseline was the unmodified version of the VM compiled by us so that no compiler related interference

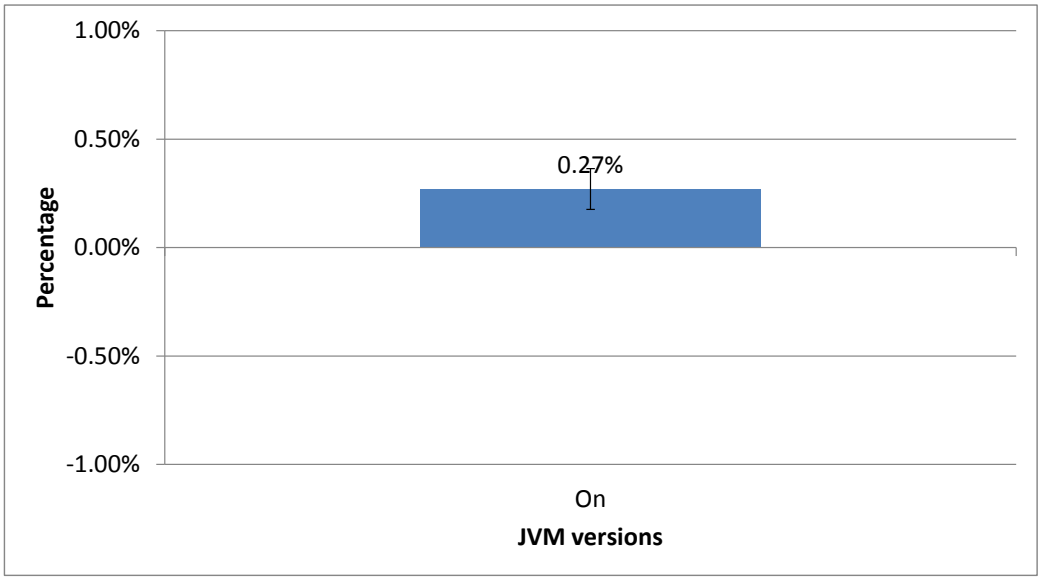


Figure 4.11: Overhead of Resident Memory of On Versus Unmodified JVM

skews the results. Furthermore, our analysis was extended in both use cases of the modified VM, i.e. with and without our new tool switched on. Finally, besides measuring the throughput related overhead, we have also investigated the memory footprint overhead.

In our initial approach the performance overhead was considered unacceptably high for the *modified on* version. That led us to a partial redesign of our implementation, writing the second and final version of the tool. In that case, both performance and memory overhead were acceptably low for all tested cases. Consequently, the impact of our modifications to the J9 JVM is minimal or even negligible.

In the next chapter we investigate some test cases where our tool demon-

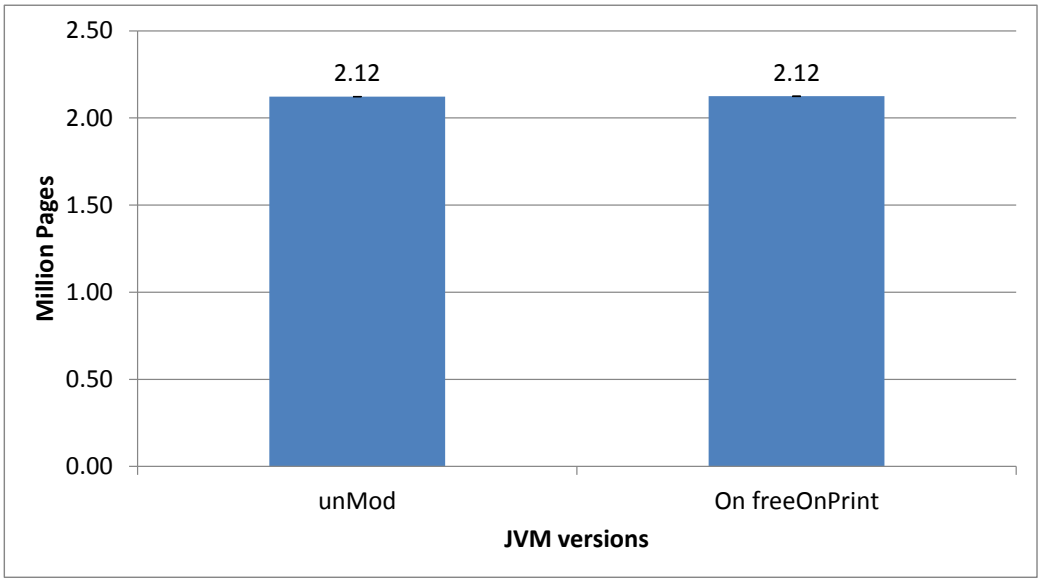


Figure 4.12: Resident Memory of On/Free-on-print Versus Unmodified JVM

strates its usefulness.

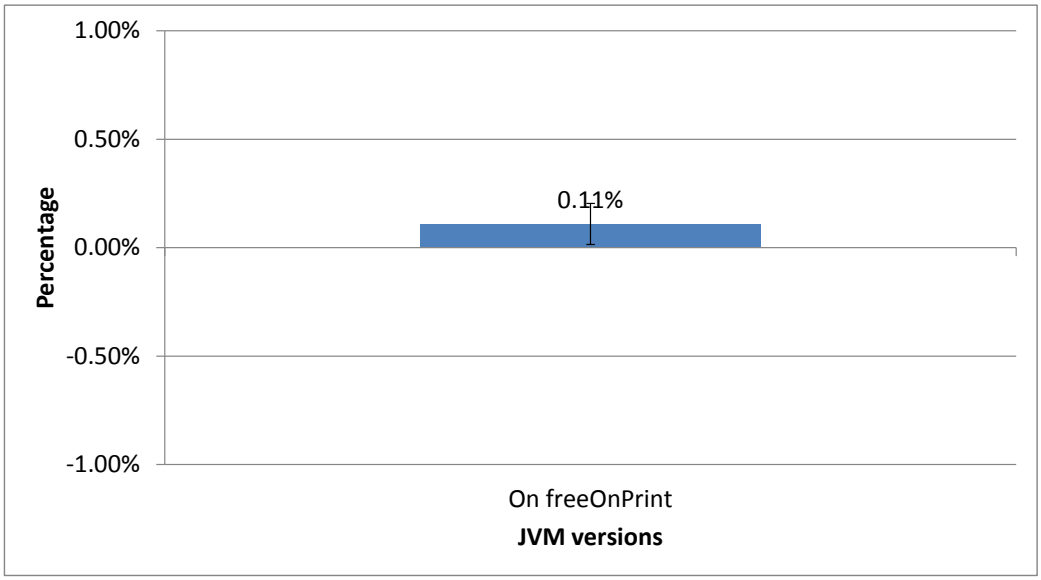


Figure 4.13: Overhead of Resident memory of On/Free-on-print Versus Unmodified JVM

Chapter 5

Pattern Profiling

5.1 Introduction

So far we have presented Java Util Concurrent, how it uses thread parking as a means of synchronization and emphasized its importance in multithreaded Java development. We have also described a new low overhead tool, incorporated in the J9 JVM, which provides a number of thread park related metrics.

The question we attempt to answer in this chapter is how useful these new metrics are and under what circumstances they can be used by Java developers to identify and potentially optimize under-performing code. We investigate this question by using our tool to profile two different programming patterns of JUC locks that cause bottlenecks in applications. We also profile the equivalent intrinsic lock patterns using Healthcenter to compare results.

Finally, we perform optimizations to the code and measure its throughput in order to prove that the locks suggested by the profiling tools as bottlenecks are the correct ones.

In all cases, we used a testing machine with AMD64 architecture, running Linux for operating system with 16 physical and 32 logical cores. Furthermore, 64 threads were used concurrently, executing the given code. The performance results displayed are aggregated from several iterations and error bars with the standard error of the sample are used to show the level of certainty of our measurements.

5.2 Large-Critical-Section Bottleneck

One coding pattern of interest is having a number of locks which are acquired equally frequently but their critical sections differ in execution time. The bottleneck in that case should be the lock that has the largest critical section but that is not always easy to figure out just by examining the code and a profiling tool, like ours, can potentially discover the problematic lock.

We used the code of Listing 5.1 to simulate a lock bottlenecking situation with three locks acquired repeatedly by the running threads. The protected critical sections contain sleeping commands with significantly different sleeping times. Furthermore, we also use counters to measure the throughput of the application. Finally, two functions are implemented, one with JUC and the other with intrinsic locks.

Listing 5.1: Large Critical Section Bottleneck Code Pattern

```
static final long S1=4, S2=16, S3=64;
public void JUC() {
        l1.lock();
        try {
                counter1++;
                try {
                        Thread.sleep(S1);
                } catch (InterruptedException ex) {
                        interrupt();
                        return;
                }
        } finally {
                l1.unlock();
        }
        localCounter++;
        /*Similarly for two more JUC locks*/
}
public void sync() {
        synchronized (l1) {
                counter1++;
                try {
                        Thread.sleep(S1);
                } catch (InterruptedException ex) {
                        interrupt();
                        return;
                }
        }
        localCounter++;
        /*Similarly for two more intrinsic locks*/
}
```

5.2.1 Park Profiling

We profiled the JUC version of the pattern using our tool and we present the metrics that scored significantly different values for the different locks on Table 5.1. We used the stack trace, class name and hash code to identify each of the three JUC locks and the result of that identification is the first

column of the table.

Table 5.1: JUC Locks Large-Critical-Section Bottleneck Park Contention Results

Lock	**Times Parked**	**Peak Parked**	**Thread Park Time (ms)**	**Park Real Time (ms)**	**Avg Park Time (ms)**	**Avg Hold Time (ms)**	**% Real Time Util**	**% Thread Time Util**	**% Thread Life Time Util**
3	1681	63	6225735	99992	3703.59	59.48	99.60	6201.05	6226.23
2	83	50	33783	1348	407.02	16.24	1.34	33.65	2504.30
1	67	59	8507	288	126.97	4.30	0.29	8.47	2953.82

The park contention results clearly show that the bottleneck in the code is lock number 3 as indicated by all the displayed metrics which are significantly higher for this lock. In particular both Thread Parking Time and Parking Real-Time metrics were at least an order of magnitude higher for the third lock in comparison to the other two. As a result, the utilization metrics scored near maximum values for lock number 3, which are 100% for the the Real-Time Utilization and 6,400% for the Thread-Time-Utilization, because we used 64 threads running concurrently. Finally, one more useful result is provided by the Times Parked metric which was two orders of magnitude higher for lock number 3 in comparison to the other two.

Another interesting aspect of the provided metrics is that the Average Hold Time was estimated with an error of at most 7% for all three locks as the reported times closely matched the preset sleeping ones.

Consequently, we can be certain that our tool clearly distinguishes between

the three JUC locks and explicitly identifies lock number 3, the one with the largest critical section, as the main bottleneck in the code.

5.2.2 Intrinsic Locks Large-Critical-Section Bottleneck Profiling

Using IBM's profiling tool Healthcenter, we profiled the equivalent version of the code which uses intrinsic instead of JUC locks. We did that in order to investigate how an established lock profiling tool handles a case as close as possible to our testing case which enables us to draw conclusions on the ability of our tool to identify bottlenecks. Healthcenter's results are displayed in Table 5.2.

Table 5.2: Intrinsic Locks Large-Critical-Section Bottleneck Profiling Results

Lock	**%miss**	**Gets**	**Slow**	**%util**	**Average hold time**
3	96	47	45	86	111710899
2	73	109	80	52	29385772
1	53	105	56	13	7552445

Healthcenter's results show that the most likely bottleneck is lock number 3 as indicated mainly by the %util and Average Hold Time metrics for which that locked registered significantly higher values than the other two. Healthcenter reports time in clock cycles rather than milliseconds which, for our particular testing machine (1,862 MHz), equates to around 59.7ms, 15.7ms and 4.0ms

for locks number 3, 2 and 1 respectively, also closely matching the equivalent metric of our tool.

Therefore, our tool's reported data correlates with Healthcenter's, an established profiling tool, when the equivalent code with intrinsic instead of JUC locks was used. It is important to note that Healthcenter does not report park contention data, so no results were produced for the JUC version of the code in its report.

5.2.3 Simulated Optimization

So far we have established that lock number 3 has to be the bottlenecking factor in this code. In order to prove this, we simulated optimizing all three locks by halving the sleeping time of their critical sections, one at a time, and measuring their performance by using the following throughput formula:

$$Throughput = \frac{\sum_{i=1}^{\#threads} localCounter_i + counter1 + counter2 + counter3}{TotalExecutionTime}$$

We ran each different case 64 times and present the aggregated results in Table 5.3. The throughput results are measured in increment operations per millisecond and the measured relative standard error was at most 0.003%.

Using the collected throughput data, we plotted the attained speedup of each different case in Figure 5.1. The results clearly show that for both JUC and intrinsic locks a significant speedup was measured only when lock number

Table 5.3: Simulated Optimization Results for Large-Critical-Section Bottleneck Pattern

Lock Type	**Baseline**	**Lock 1 Optimized**	**Lock 2 Optimized**	**Lock 3 Optimized**
JUC	0.096485	0.096497188	0.096500313	0.189027031
Intrinsic	0.096174063	0.096183125	0.096185937	0.18763125

3 was optimized. Therefore, that proves that this lock was the bottleneck and our tool was able to identify it. Similarly, for the intrinsic locks case, Healthcenter was also able to find the bottleneck.

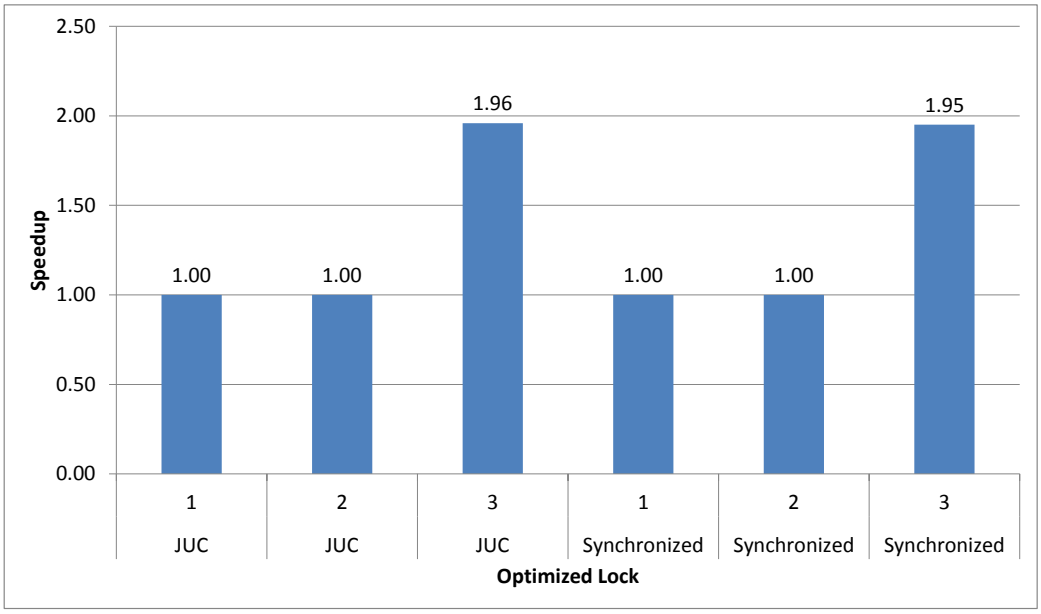


Figure 5.1: Speedup of Simulated Optimizations for Large-Critical-Section Bottleneck Pattern

5.2.4 Pattern Conclusions

Consequently, %Real-Time Util, %Thread-Time Util and %Thread-Life-Time Util, which scored significantly higher values for the bottlenecking lock in comparison to the other two, can identify a bottleneck even if no comparison with other locks could have been made because they scored very close to their maximum possible values, i.e. 100% and 6,400% (because 64 threads were used). Similar conclusions can be drawn for the Peak Parked metric which registered very close to the maximum possible number, 63 threads, showing that at least at one point during the execution of the application all but one thread were blocked on that lock, something which qualifies as the definition of a bottleneck.

5.3 Frequently-Acquired-Lock Bottleneck

Another interesting coding pattern is using a number of locks which have equally lengthy critical sections but are acquired with different frequencies. The bottleneck in that case should be the lock that has the highest acquisition frequency but that is not always easy to figure out just by examining the code and a profiling tool, like ours, can potentially discover the problematic lock. We used the code of Listing 5.2 to simulate a lock bottlenecking situation with two locks acquired repeatedly by the running threads. The protected critical sections contain sleeping commands with equal sleeping times but the probability a thread selects between the two critical sections is not equal

but 3 times higher for the first lock. Furthermore, we are also using counters to measure the throughput of the application. Finally, two functions are implemented, one with JUC and the other with intrinsic locks.

Listing 5.2: Frequently-Acquired-Lock Bottleneck Code Pattern

```
static volatile long S1 = 32, S2 = 32;
public void JUC() {
            ReentrantLock l;
            long S;
            boolean mode1 = ThreadLocalRandom.
                        current().
                        nextDouble() < p;
            if (mode1) {
                l = l1;
                S = S1;
            } else {
                l = l2;
                S = S2;
            }
            l.lock();
            try {
                if (mode1) {
                    counter1++;
                } else {
                    counter2++;
                }
                try {
                    Thread.sleep(S);
                } catch (InterruptedException ex) {
                    interrupt();
                    return;
                }
            } finally {
                l.unlock();
            }
            localCounter++;
        }
        public void sync() {
            ReentrantLock l;
            long S;
            boolean mode1 = ThreadLocalRandom.
                        current().
```

```
                    nextDouble() < p;
            if (mode1) {
                l = l1;
                S = S1;
            } else {
                l = l2;
                S = S2;
            }
            synchronized (l) {
                if (mode1) {
                    counter1++;
                } else {
                    counter2++;
                }
                try {
                    Thread.sleep(S);
                } catch (InterruptedException ex) {
                    interrupt();
                    return;
                }
            }
            localCounter++;
        }
    }
```

5.3.1 Park Profiling

We profiled the JUC version of the pattern using our tool and we present the values of the metrics that registered significant differences between the locks in Table 5.4. We used the stack trace, class name and hash code to identify each of the two JUC locks and the result of that identification is the first column of the table.

The park contention results clearly show that the bottleneck in the code is lock number 1, as indicated by all the displayed metrics. In particular, both Thread Parking Time and Parking Real-Time registered values at least 2

Table 5.4: JUC Frequently-Acquired-Lock Bottleneck Park Contention Results

Lock	**Times Parked**	**Peak Parked**	**Thread Park Time (ms)**	**Park Real Time (ms)**	**Avg Park Time (ms)**	**% Real Time Util**	**% Thread Time Util**	**% Real Life Time Util**	**% Thread Life Time Util**
1	3224	64	6260087	100028	1941.71	98.03	6134.75	100	6258.33
2	164	10	6721	4219	40.98	4.13	6.59	4.29	6.83

orders of magnitude higher for the first in comparison to the second lock. As a result, the utilization metrics scored near maximum values for this lock, which are 100% for the the Real-Time Utilization and 6,400% for the Thread-Time Utilization, because we used 64 threads. Furthermore, one more useful result is provided by the Times Parked metric which was one order of magnitude higher for lock number 1 in comparison to the other.

Another interesting aspect is provided by the Peak Parked metric which registered the maximum possible number of concurrently parked threads for lock number 1, reaching 64. This shows that at least one time, all threads were blocked on the first lock. This might seem counterintuitive as one might expect the maximum to be 63, allowing at least one thread to run in the lock's critical section. However, a thread can be unparked and just before we decrement the concurrently parked counter, another thread can be parked incrementing it again and reaching the peak momentarily. For this scenario to take place, the lock must be very heavily contested, another indication that lock 1 is the bottleneck in the code.

Consequently, we can be certain that our tool clearly distinguishes between the two JUC locks and explicitly identifies lock number 1, the one most frequently acquired, as the main bottleneck in the code.

The Average Hold Time, not displayed in the park contention metrics table, was 31.02ms and 25.72ms for the first and second JUC lock respectively. These numbers are very good approximations of the actual 32ms sleeping time set for the critical sections of both locks and although they do not help us distinguish between the locks' influence in bottlenecking the application, they provide evidence that our approach to computing the metric is correct.

5.3.2 Intrinsic Locks Profiling

Using Healthcenter, we profiled the equivalent version of the code which uses intrinsic instead of JUC locks. We did that in order to investigate how an established lock profiling tool handles a case as close as possible to our testing case. This enables us to draw conclusions on the ability of our tool to identify bottlenecks. Healthcenter's results are displayed in Table 5.5.

Table 5.5: Intrinsic Bottleneck Profiling Results

Lock	**%miss**	**Gets**	**Slow**	**%util**	**Average hold time**
1	83	92	76	83	57998175
2	55	55	30	50	58866220

Healthcenter's results show that the most likely bottleneck is lock number

1 as indicated by the %util, %miss, Gets and Slow metrics for which that locked registered higher values than the other. Healthcenter reports time in clock cycles rather than milliseconds which, for our particular testing machine (1,862 MHz), equates to around 31ms for both locks. Therefore, our tool's reported data correlate with Healthcenter's, an established profiling tool, when the equivalent code with intrinsic instead of JUC locks were used. We again remark that Healthcenter does not report park contention data, so no results were produced for the JUC version of the code in its report.

5.3.3 Simulated Optimization

So far we established that lock number 1 has to be the bottlenecking factor in this code. In order to prove this, we simulated optimizing both locks by halving the sleeping time of their respective critical sections, one at a time, and measured their performance by using the following throughput formula:

$$Throughput = \frac{\sum_{i=1}^{\#threads} localCounter_i + counter1 + counter2}{TotalExecutionTime}$$

We ran each different case 64 times and present the aggregated results in Table 5.6. The throughput results are measured in increments per millisecond and the measured relative standard error was at most 0.19%.

Using the collected throughput data, we plotted the attained speedup of each different case in Figure 5.2. The results clearly show that for both JUC and intrinsic locks a significant speedup was measured only when lock number

Table 5.6: Simulated Optimization Results for Frequently-Acquired-Lock Bottleneck Pattern

Lock Type	**Baseline**	**Lock 1 Optimized**	**Lock 2 Optimized**
JUC	0.083770938	0.16535375	0.0837525
Intrinsic	0.083163594	0.162668906	0.083208281

1 was optimized. Therefore, that proves that this lock was the bottleneck and our tool was able to identify it. Similarly, for the intrinsic locks case, Healthcenter was also able to find the bottleneck.

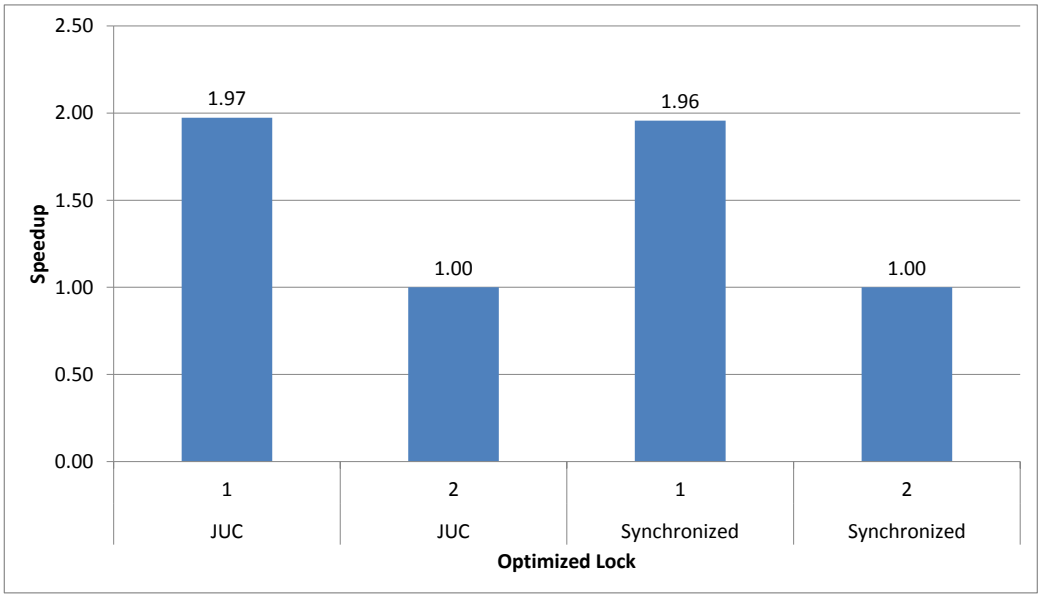


Figure 5.2: Speedup of Simulated Optimizations for Frequently-Acquired-Lock Bottleneck Pattern

5.3.4 Pattern Conclusions

Consequently, %Thread-Time Util and %Thread-Life-Time Util, which scored significantly higher values for the bottlenecking lock in comparison to the other, can identify a bottleneck even if no comparison with other locks could have been made because they scored very close to the maximum possible value, 6,400% since 64 threads were used. Similar conclusions can be drawn for the Peak Parked metric which registered the maximum possible number, 64 threads, showing that at least at one point during the execution of the application all threads were blocked on that lock, something which qualifies as the definition of a bottleneck.

5.4 Summary

In this chapter we investigated possible uses of our proposed park contention measurement tool. In particular, our main goal was to show that the tool can identify JUC lock bottlenecks in Java code. To that end, we implemented two coding patterns that used JUC locks in which one of them was the bottleneck; our tool was able to provide us enough information to distinguish it. Furthermore, we also implemented the equivalent intrinsic locks solution for the pattern and used Healthcenter for profiling, whose results correlated with ours. Finally, we simulated optimizing all critical sections, by halving their durations one at a time, and measured any speedups from the original version. The designated locks were the only ones that produced speedups

when optimized.

More specifically, in the first pattern we used equally frequently acquired locks but with different critical section sizes, whereas in the second pattern, the locks had the same critical section size but varying acquisition frequencies. The bottleneck in the first pattern was the lock with the largest critical section and in the second the one with the highest acquisition frequency. This was confirmed by both our tool's and Healthcenter's report as well as by the simulated optimization we performed.

Chapter 6

Conclusions

6.1 Overview

We proposed a new JUC park contention measurement tool incorporated in the J9 JVM and showed its usefulness in finding bottlenecks in Java code; thus, providing a means to the developer to optimize and accelerate their applications. Our tool has negligible overhead, as measured using the SPECjbb2013 benchmark, which means that the provided results are accurate since no practical interference to the code's critical path occurs.

We started by presenting the necessary background which includes Java, the JVM, Java's built-in thread support and discussed how these constructs can be potentially misused making their use not so straightforward to the uninitiated. A solution to that is the Java Util Concurrent (JUC) library which not only provides ready solutions for a number of different multi-threaded

needs but which are also thread-safe avoiding a number of misuse cases. In particular, JUC provides explicit locks which extend the functionality of the Java intrinsic locks (implemented by the keyword `synchronized`) by supporting, polled, timed and non-block structured acquisitions as well as read-write locks.

JUC uses the `LockSupport` class as a means of synchronizing threads. This class parks threads on a park blocking Java object, effectively blocking them from execution until a thread unpark command is issued. To the best of our knowledge there is no profiling tool embedded in a JVM which measures park related metrics and, with our current work, we aspire to cover that gap. We modified IBM's J9 JVM to capture park contention data by instrumenting the code which blocks a parked thread. We associated a parking record for each park blocking object where we stored all the necessary information for our measurements. Furthermore, we also used an extended park record, allocated and initialized after a designated park collection threshold is reached, which stores the bulk of our data, saving memory and reducing overhead. Moreover, we also provide a means to dispose park records associated with dead park blocking objects after their results are printed with the free-on-print mode of the tool.

We provide the following park contention metrics:, Thread Name, Stack Trace, Class Name, Hash Code, Times Parked, Parked Now, Peak Parked, Parking Thread-Time, Parking Real-Time, Average Parking Time, Average Hold Time, Real-Time Utilization, Thread-Time Utilization, Real-Life-Time

Utilization and Thread-Life-Time Utilization.

As far as the overhead of our tool is concerned, we have measured it using the SPECjbb2013 benchmark. We compared the baseline unmodified and compiled by us version of the J9 JVM with the modified version with our tool switched off, switched on, and switched on with the free-on-print mode enabled. In all three cases the measured overhead was statistically insignificant and comparable to 0.

Finally, in order to show how our tool can be useful, we used it to find the bottleneck in two JUC lock patterns. In the first case a large critical section was the problem, whereas in the second, a frequently acquired lock. In both cases our tool was able to identify the bottleneck which was confirmed by simulating optimizations and measuring their speedups. Another confirmation of the usefulness of our tool came from Healthcenter, IBM's Java profiling tool (it does not provide park related metrics), when we profiled with it the equivalent intrinsic lock code patterns and its results correlated with ours. Consequently, providing profiling information and tools for JUC thread parking enables notable optimizations of application performance.

6.2 Future Work

IBM's Healthcenter provides a very nice graphical way of presenting profiling data but it lacks park related metrics. A proposed future work can be to incorporate our tools results to Healthcenter's input which would give

developers unified and integrated profiling capabilities.

Bibliography

[1] *The da capo benchmark suite*, http://www.dacapobench.org/, August 13, 2013.

[2] *IBM Java Virtual Machine (JVM)*, http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fuser%2Fjava_jvm.html, April 17, 2013.

[3] *IBM Monitoring and Diagnostic Tools for Java - Health Center Version 2.1*, http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/#benefits, April 17, 2013.

[4] *Java platform, standard edition 7 api specification*, http://w3.java.ibm.com/java/docs/java7/api/, April 17, 2013.

[5] *Java SE HotSpot at a Glance*, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html, April 17, 2013.

[6] *Multicore software development kit*, https://www.ibm.com/developerworks/mydeveloperworks/

groups/service/html/communityview?communityUuid=
9a29d9f0-11b1-4d29-9359-a6fd9678a2e8, August 13, 2013.

[7] *Specjbb2013*, http://www.spec.org/jbb2013/, August 13, 2013.

[8] *Specjbb2013 user guide*, http://www.spec.org/jbb2013/docs/userguide.pdf, August 13, 2013.

[9] Guancheng Chen and Per Stenstrom, *Critical lock analysis: diagnosing critical section bottlenecks in multithreaded applications*, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 71.

[10] David Eklöv, Nikos Nikoleris, and Erik Hagersten, *A Profiling Method for Analyzing Scalability Bottlenecks on Multicores*, Tech. Report 2012-030, Department of Information Technology, Uppsala University, October 2012.

[11] B. Goetz, T. Peierls, J. Bloch, J. Bowebeer, D. Holmes, and L. Doug, *Java concurrency in practice*, Addison-Wesley Professional, 2006.

[12] James Gosling, Bill Joy, Guy L. Steele Jr, Gilad Bracha, and Alex Buckley, *The java language specification*, Addison-Wesley, 2013.

[13] Tim Lindholm and Frank Yellin, *The Java virtual machine specification*, vol. 297, Addison-Wesley Reading, 1997.

[14] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield, *Analyzing lock contention in multithreaded applications*, ACM Sigplan Notices, vol. 45, ACM, 2010, pp. 269–280.

Appendix A

Resident Memory Overhead Daemon

Listing A.1: Resident Memory Measurement Daemon

```
#!/bin/bash
# Usage: resmem command

$* 2>/dev/null >/dev/null &
pid=$!
done=0
while [ $done -eq 0 ]; do
        r=`cat /proc/$pid/statm 2>&1 | awk '{print $2}'`
        if [[ $r == /proc/* ]] ;
        then
                done=1
        else
                echo $r
                sleep 1
        fi
done
```

Vita

Candidate's full name: Panagiotis Patros
University attended (with dates and degrees obtained): Degree of Informatics and Telecommunications, National and Kapodistrian University of Athens, 2010
Publications: N/A
Conference Presentations: N/A