Dynamic Monitor Allocation in the IBM J9 Virtual Machine

by

Marcel Dombrowski

Bachelor of Science in Computer Science
Bonn-Rhein-Sieg University of Applied Sciences, 2010

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

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth B. Kent, PhD, Faculty of Computer Science
Rainer Herpers, PhD, Faculty of Computer Science

Examiner Board: Gerhard Dueck, PhD, Faculty of Computer Science
David Bremner, PhD, Faculty of Computer Science
External Examiner: Yves Losier, PhD, Faculty of Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April 2013
©Marcel Dombrowski, 2013
Abstract

With the Java language and sandboxed environments becoming more and more popular, research needs to be conducted into improving the performance of these environments while decreasing their memory footprints. This thesis focuses on a dynamic approach for growing monitors for objects in order to reduce the memory footprint and improve the execution time of the IBM Java Virtual Machine. According to the Java Language Specification every object needs to have the ability to be used for synchronization. This new approach grows monitors only when required. The impact of this approach on performance and memory has been evaluated using different benchmarks and future work is also discussed. On average, a performance increase of 0.6% and a memory reduction of about 5% has been achieved with this approach.
Dedication

For my parents

Klaus-Dieter Dombrowski and Gisela Dombrowski

and my brothers

Sascha Dombrowski and Marco Dombrowski.
Acknowledgements

It is a pleasure to thank those who made this thesis possible. Therefore I would like to especially thank my family, who supported me not only while writing this thesis, but throughout my whole course of studies. Furthermore I would like to thank my friends, who have been so supportive and respectful, although I did not have much spare time in the last couple of months.

In particular I would like to thank my advisors Kenneth B. Kent from the University of New Brunswick, Michael Dawson from IBM Canada, and Rainer Herpers from the Bonn-Rhein-Sieg University of Applied Sciences. Furthermore, I would also like to thank my colleagues Kim Briggs, Manfred Jendrosch, Nicolas Neu, Patros Panos, Kristine Richard, Andrew Somerville, Chenwai Wang, and Yang Wang. Without their help writing this thesis would have been next to impossible.

I also gratefully acknowledge the financial support of the EU-Canada Joint Study and/or Training Programme; project: “Building Internationally Dis-
tributed Computer Science Joint Degree Programmes" agreement number: 2008-1811 / 001-001 CPT CANHET. I would also like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Finally, I would also like to acknowledge IBM’s support and contribution to this project.

Marcel Dombrowski
# Table of Contents

Abstract

Dedication

Acknowledgments

1 Introduction and Motivation

2 Basics

2.1 Java

2.1.1 Language

2.1.2 Virtual Machine

2.1.3 Just-in-Time Compiler

2.1.4 Garbage Collection

2.2 Monitors and Locks

2.3 Hashcode

2.4 Benchmarks

2.5 Related Work

2.5.1 Dynamic Monitor Allocation

Abstract  
Dedication  
Acknowledgments  
1 Introduction and Motivation  
2 Basics  
2.1 Java  
2.1.1 Language  
2.1.2 Virtual Machine  
2.1.3 Just-in-Time Compiler  
2.1.4 Garbage Collection  
2.2 Monitors and Locks  
2.3 Hashcode  
2.4 Benchmarks  
2.5 Related Work  
2.5.1 Dynamic Monitor Allocation
2.5.2 Problem Formulation ............................................. 23

3 Project Design ......................................................... 25
  3.1 Requirements ...................................................... 25
  3.2 Approach ........................................................... 28
  3.3 Design Decisions ................................................... 29
    3.3.1 Hardware ....................................................... 29
    3.3.2 Software ....................................................... 30

4 Realization ............................................................. 31
  4.1 Code Structure .................................................... 32
  4.2 Evaluation of Hashcode Bits in an Object ....................... 34
  4.3 Architecture ...................................................... 37
    4.3.1 Overloading Header Bits .................................... 39
    4.3.2 Creating a New Slot in an Object ......................... 40
    4.3.3 Force-moving an Object ..................................... 43
    4.3.4 Integration into the Virtual Machine ....................... 47

5 Evaluation ............................................................. 54
  5.1 Approach .......................................................... 55
  5.2 Evaluation of the Implementation ............................... 56
    5.2.1 Monitor Growing ............................................. 56
    5.2.2 Force-Moving an Object .................................... 60
    5.2.3 Integration ................................................... 64
5.3 Benchmarking ........................................... 67
5.4 Results .................................................. 69
5.5 Usage for the JIT ........................................ 76

6 Conclusion and Outlook .............................. 78

Bibliography ............................................. 87

A Appendix .............................................. 88

A.1 Hashed Objects ........................................... 88
    A.1.1 Dummy Program .................................. 88
    A.1.2 SciMark Benchmark .............................. 90
    A.1.3 SPECjbb2005 Benchmark ....................... 92
A.2 Program for Evaluation of the Approach .......... 93
A.3 Program for Benchmarking ......................... 94
A.4 Benchmark Results ..................................... 99
    A.4.1 CPU Results for SPECjbb2005 ................. 99
    A.4.2 Memory Usage Results for SPECjbb2005 ...... 103

Vita

viii
List of Tables

5.1 Average CPU Results of the SPECjbb2005 Benchmark ........ 70
5.2 Average Memory Usage of the SPECjbb2005 Benchmark .... 71
5.3 Percentage for the Average Memory Usage and CPU Usage of
the SPECjbb2005 Benchmark Compared to the Reference JVM. 72
5.4 Time Results in Milliseconds for the DaCapo Benchmark with
the JIT Disabled ........................................ 75
5.5 Time Results in Milliseconds for the Self-written Locking Bench-
mark with the JIT Disabled .............................. 75
5.6 Statistics for Blocking Objects. ................................. 76
A.1 CPU Results of the SPECjbb2005 Benchmark ............. 102
A.2 Memory Usage of the SPECjbb2005 Benchmark ............ 105
List of Figures

2.1.1 The TIOBE Programming Community Index .......................... 7
2.1.2 The Difference Between a Native Program and a Java Program 10
2.1.3 The Overall Structure of the JIT Compiler ........................ 12
2.1.4 A Generational GC Technique ......................................... 14
2.1.5 The Scavenging GC Technique ........................................ 15
2.2.1 Flat Lock Structure ..................................................... 17
2.2.2 Inflated Lock Structure ................................................ 19

4.3.1 The Current Layout of an Object Before and After it has been Moved ......................................................... 40
4.3.2 The New Layout of an Object Before and After it has been Moved .......................................................... 41
4.3.3 Flowchart for Updating an Object .................................. 50
4.3.4 UML-like Sequence Diagram for Handling a Possible Deadlock Situation on Updating an Object ......................... 52

5.2.1 Possible Memory Layout for Objects with Grown Lockwords 60
5.2.2 Hierarchy of the Objects Used to Evaluate the Force-move Method ........................................... 61
5.2.3 Possible Memory Layout Before Moving an Object with Three Objects in the Portrayed Area ................. 63
5.2.4 Possible Memory Layout After Moving an Object with Three Objects in the Portrayed Area ................. 63
5.4.1 Results for the CPU Usage of the SPECjbb2005 Benchmark . 73
5.4.2 Results for the Memory Consumption of the SPECjbb2005 Benchmark ........................................ 74

A.1.1 Number of Objects in a Simple Dummy Program ........ 88
A.1.2 Number of Hashed Objects in a Simple Dummy Program . 89
A.1.3 Percentage of Hashed Objects in a Simple Dummy Program . 89
A.1.4 Number of Objects in the SciMark Benchmark ............ 90
A.1.5 Number of Hashed Objects in the SciMark Benchmark . . 90
A.1.6 Percentage of Hashed Objects in the SciMark Benchmark . 91
A.1.7 Number of Objects in the SPECjbb2005 Benchmark ........ 92
A.1.8 Number of Hashed Objects in the SPECjbb2005 Benchmark . 92
A.1.9 Percentage of Hashed Objects in the SPECjbb2005 Benchmark 93
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>JCL</td>
<td>Java Class Library</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-in-Time</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>SPEC</td>
<td>Standard Performance Evaluation Corporation</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Motivation

International Business Machines Corporation (IBM) is one of the major Java Virtual Machine developers. Founded in 1911 IBM is one of the biggest employers worldwide today with over 400,000 employees [1] in over 12 research laboratories worldwide. IBM is a hardware and software manufacturer and also offers consulting services in different areas such as mainframe computers.

In their Center for Advanced Studies (CAS) initiative, IBM has currently 20 locations worldwide where university students and professors collaborate with IBM [2]. In Atlantic Canada this initiative is also called Center of Advanced Studies: Atlantic (CASA). The University of New Brunswick (UNB) is part of this initiative since 2010 [3]. "CASA’s mission is to conduct research aimed at advancing the performance and concurrency of software executing on multicore systems" [3].
IBM has a long history working on Java: the IBM T.J. Watson Research Center has been working on the IBM Jikes Research Virtual Machine, a virtual machine used to research new methods to improve Java [4]. Furthermore, IBM has put effort into building a Java virtual machine in Java, the IBM Jalapeño Virtual Machine, which is made especially for Java servers [5]. Finally, IBM is building the IBM J9 Virtual Machine, a virtual machine that is used in many IBM products and used by many consumers worldwide. In their CASA initiative IBM focuses their research on the J9 Virtual Machine.

As will be explained in Section 2.1.2 the Java Virtual Machine is the middleware that runs Java applications. This makes the Java Virtual Machine the bottleneck for all applications that it is running. Therefore new ways always need to be researched and implemented that speed up the Java Virtual Machine and make it less memory consuming.

This thesis is structured as follows: in Chapter 2, the background to understand the remainder of this thesis is given. This includes a background in the Java language, as well as the Java virtual machine, just-in-time compilation, and garbage collection. Furthermore, this Chapter gives an introduction into monitors and locking, as well as hashcodes and benchmarks. Section 2.5 will then show related work to this research as well as the problem formulation of this project. The design of this project is then explained in Chapter 3. It
starts with the requirements for this project, followed by the approach and the design decisions. In Chapter 4 the actual implementation is explained while its evaluation is found in Chapter 5. Finally, this project concludes in Chapter 6 and gives an outlook on possible future work.
Chapter 2

Basics

This chapter presents the necessary background information in order to understand the remaining chapters of this report. It will start with an explanation of the Java language in Section 2.1.1 and the Java Virtual Machine (JVM) in Section 2.1.2. As parts of this project are done in the Garbage Collection (GC), the GC itself is briefly explained in Section 2.1.4. Another feature of Java is the Just-in-Time compilation of code which is introduced in Section 2.1.3.

As this thesis focuses on monitor allocation for objects, the mechanics behind monitors and locks are explained in Section 2.2. A brief introduction in hashcodes is given in Section 2.3 because this project uses the hashcode bits in the object header. Finally, as this project is evaluated using benchmarks, an explanation and introduction of benchmarks and benchmarking is given...
in Section 2.4.

2.1 Java

Java itself is split into two major components: the Java language itself as well as the Java Virtual Machine. Every machine that wants to run Java applications needs to have a JVM installed. This has the advantage that Java code behaves the same on every machine that has the JVM installed. Java has been specified to consist of interpreted and native code. To achieve this behaviour Java makes use of a Just-in-Time (JIT) compiler. The JIT compiler translates parts of the bytecode into machine code.

2.1.1 Language

Java is an object-oriented programming language that has been designed with simplicity in mind in order to make the language easy to learn. The specification of the language started in 1991 at Sun Microsystems Corporation [6]. While being related to the C++ programming language it is organized differently to be a production language [7]. This is achieved by omitting features that make C++ complex. Originally it was developed to overcome the problems of creating software for networked devices. To achieve this behaviour the compiled code has to operate on any client [8].

In the Java language it is distinguished between run time and compile time.
Compile time consists of “translating programs into a machine-independent bytecode representation” [7] and run time includes “loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution” [7].

The language itself is at a high-level which tries to abstract from the underlying machine as much as possible. This includes a Garbage Collector (GC) which handles the problems that might occur on allocation and deallocation of objects in other languages, e.g. the `delete` statement in C++ [7].

In the first years after its introduction Java gained popularity due to its new concepts and the ease of use. In October 1997 Java was ranked fourth place in the TIOBE programming community index [9] and ten years later it was ranked first place. The TIOBE programming community index is one measure of the popularity of programming languages. As of 2012 the Java language has decreased in its popularity and C has taken over first place again. An actual chart of the popularity of programming languages by the TIOBE programming community index can be seen in Figure 2.1.1.

Since Java is an object-oriented language everything is encapsulated in classes. Furthermore, Java does not separate between header and source files which results in every Java class being handled in one single file. Additionally, no prototype declaration of functions is needed, so the programmer can call methods which have been declared later in the class file. In order to give
the program a starting point a main method is needed. In Java this method needs to be public, so that it can be accessed external to the class where the method is declared, as well as static, which means that this method can be invoked without creating an instance of the class first, and without a return type (void). This results in the main method being declared as follows:

```java
public static void main(String args[])
```

With the String args[] parameter the caller can pass command line arguments to the program. These are separated in a String array and can be read and interpreted in the main method. This main method needs to be part of a class file. Additionally, only one main method is allowed for each Java project. A possible sample class file thesis can look like this:
This simple class exits immediately after launch as it contains no code. An overview of Java as well as code examples are given in [7]. Java code can be written in different ways ranging from text editors (e.g. vim on UNIX-like systems) to complex Integrated Development Environments (IDE). Two popular IDEs are Eclipse [10] and NetBeans [11]. Then the Java code can be compiled to bytecode which can be executed by a Java Virtual Machine.

2.1.2 Virtual Machine

The Java Virtual Machine is the program that is used for executing bytecode. It is responsible for the abstraction of the underlying operating system, the sandboxing of the programs, as well as keeping the compiled code small in size. A Virtual Machine (VM) adds a “software layer to a real machine to support the desired architecture” [12]. For this, the VM maps the “interface and resources [of the system to virtualize] onto the interface and resources of an underlying, possibly different, real system” [12].

With this behaviour it is also possible to sandbox the VM. A sandbox puts a program into a restricted environment, where it can do no harm to the
underlying operating system [13], e.g. similar to what a virus could do. Furthermore, a VM can have an increased instruction set compared to the underlying machine. This allows the compiled code to be smaller in size than it would be if compiled for the machine directly.

Analogous, a Java VM sandboxes the compiled Java code to protect it from harming the operating system. Furthermore it provides an environment that behaves the same on all machines that are supported. The JVM itself does not understand any Java code, it is only capable of understanding the binary format of the class file [8]. The class file contains JVM instructions, which are often referred to as bytecode, a symbol table as well as other additional information.

For security, the JVM constrains the syntax and the structure of the instructions of the class file. Any language that can be converted to a valid class file can be run in a JVM [8]. The instruction set of the JVM is specified in [8] and is implemented by every vendor of a JVM. Every instruction (e.g. arraylength or instanceof) is eight bits wide and has usually an operand stack attached to it. This bytecode can then be executed by the JVM. The JVM then decides if the bytecode or parts of it are compiled to native code or are executed by the interpreter. For the compilation during runtime the JVM makes use of the Just-in-Time (JIT) compiler. A brief introduction of the JIT will be given in Section 2.1.3.
Figure 2.1.2 shows a simplistic view of the difference between a native program and a Java program. Both approaches have a similar design starting with a sourcefile. A compiler then translates the sourcefile in machine code for the native approach (left path in the Figure) or it translates the Java sourcecode into bytecode (right path in the Figure). Both results can be seen as programs, the executable is machine and operating system dependent and the bytecode file is machine and operating system independent.

![Diagram](image)

**Figure 2.1.2:** The difference between a native program and a Java program. The JVM adds an additional layer in order to achieve abstraction from different systems.

The executable then runs as a native application and can make use of mid-
Software that is installed, as well as of parts of the operating system and the hardware. If system calls are needed, the Java program can ask the JVM for permission to use underlying resources. With this the Java program cannot gain unauthorized access to the operating system or the hardware because the JVM handles the communication with these resources. Figure 2.1.2 is simplified because it does not show all the elements involved, e.g. behind the compiler is usually a complete toolchain, including a linker for different object files, as well as the JVM supports the Java Native Interface (JNI), with which it is possible for certain commands to break out of the JVM to issue system calls.

As the original specification of the Java language has been done by Sun Microsystems Corporation they have one of the most widely used Java VMs. Other big vendors include IBM, Hewlett Packard, as well as Microsoft, but Microsoft discontinued their JVM implementation in 2001 after their lawsuit against Sun [14].

2.1.3 Just-in-Time Compiler

The Just-in-Time Compiler (JIT) can be used to compile code during runtime. Therefore the bytecode itself needs to be interpreted and converted to native instructions for the given processor. The JVM takes care of this. Sometimes it might be faster to compile code to native code, especially for code parts which run frequently. A JIT compiler is called during runtime
and compiles code passages into native code for faster execution. This code can then be run at full native speed. The only overhead introduced is the compile overhead as compilation happens during runtime. Thus a fast and lightweight implementation of the JIT compiler is needed [15].

Figure 2.1.3: The overall structure of the JIT compiler. This Figure shows the various steps a JIT compiler usually performs in order to generate native code [15].

Reasons for slow execution performance of Java programs are the object-oriented nature of the language itself, which involves having many small methods in many objects, that lead to methods being invoked more often [15]. Another reason is that the Java language has been specified to check for runtime exceptions in order to ensure the “validity of accesses to objects and arrays” [15]. If this access is invalid an exception needs to be thrown and the whole environment needs to be made accessible to the exception handler. This introduces a “large penalty in program execution and prevents the application of conventional loop optimization techniques” [15]. A third reason
is that additional overhead is introduced by locking on objects in synchronized methods in order to ensure the atomicity for a set of operations. The JIT addresses all of these problems by applying different techniques in order to make the code run faster while conserving the paradigms specified by the Java language. The structure of the JIT compiler can be seen in Figure 2.1.3.

2.1.4 Garbage Collection

"Garbage Collection (GC) is the automatic reclamation of computer storage" [16]. In non GC systems the developer is responsible for claiming the space of unused variables and objects. The GC's job is to find objects during runtime that are no longer used and free their space so that new objects can occupy this memory space. Wilson defined in [16] an object as garbage if "it is not reachable by the running program via any path of pointer traversals".

In 1960 two methods were introduced for automatic storage reclamation: reference counting [17] and tracing [18]. Since then tracing collectors have been more widely used than reference counting. Variants of tracing include mark-and-sweep, mark-and-compact, and the copying GC. The assumption behind tracing collectors is that it is acceptable to periodically trace all objects on the heap. Problems with reference counting collectors are the runtime overhead of the reference count as well as the inability to detect cycles and the necessity to include another GC technique to cope with cyclic garbage [19].
Figure 2.1.4: A generational GC technique [20]. Space is segmented into two different areas. The left part shows the new area, where new objects are allocated and the right part shows the tenured area, where old objects are stored.

A mark-sweep GC for instance implements two phases for garbage collection: distinguish live from dead objects by tracing all objects starting from the root set and traversing along the graphs of pointer relationships (mark phase) and reclaiming garbage by finding all unmarked objects and reclaiming their space (sweep phase). A mark-compact GC avoids the problem of fragmentation that a mark-sweep GC introduces by moving the live objects until all objects are in a contiguous space [16].

A generational GC splits the heap into a new area, called nursery, and an old area, called tenured. All objects are created in the new space and if they are used long enough they are moved into the old area (Figure 2.1.4). The new area is split into allocate and survivor space. In the allocate section new objects are introduced. If that space is filled the GC triggers the scavenger. During this process all reachable objects are either copied into the survivor space or into the tenured space. Unreachable objects in the new area are left untouched. After all objects have been copied the allocate and survivor
space switch roles (Figure 2.1.5).

The IBM J9 Virtual Machine distinguishes between different GC policies: optthruput, optavgpause, balanced, and gencon. The optthruput policy disables concurrent marking, while optavgpause enables it. The gencon policy uses concurrent and generational GC techniques and balanced uses a hybrid approach for GC [21].

2.2 Monitors and Locks

A lock is needed when more than one thread or process wants to operate in a critical region. A critical region is a piece of code that involves a shared resource. Multiple threads working on one shared variable or object can
lead to undesired behaviour depending on the value of the variable. Thus variable locks are needed to ensure variable consistency along all threads. In its simplest form a lock is another variable which is checked for being empty, e.g.

```java
if (lock == 0)
```

If this variable is not set the accessing process will own the variable by writing a value different from 0 to it. This has to be done atomically because if not multiple threads can set the lock at the same time. Locks can be accessed either non-blocking as shown in the above code example or blocking which could be

```java
while (lock != 0);
```

This statement blocks the current thread until the lock is free. There are different implementations of locks available with semaphores and mutexes being the most popular. A mutex, which is derived from the term mutual exclusion, is the aforementioned lock which includes the atomicity. It can be entered (either blocking or non-blocking) and exited. The implementation behind the mutex needs to guarantee atomicity [22]. A semaphore as originally introduced by Dijkstra in [23] is a counting mutex. It allows a specific amount of threads to enter a region and after this amount has been reached it blocks access to this resource. There are two types of semaphores: a counting semaphore, which allows more than one thread access, and a binary semaphore, which is identical to a mutex.
A monitor is an object or module which can be safely used by more than one thread concurrently. The methods of a monitor are all executed with mutexes. It provides abstraction to the programmer as the programmer then does not have to cope with the locking and unlocking of variables. The monitor itself handles the integrity of the defined region by only allowing one thread at a single given time to be active [24].

![Flat lock structure](image)

**Figure 2.2.1:** Flat lock structure [25]. In the unlocked state the lockword is empty. If the lock is set it incorporates the thread ID. The first bit shows whether a lock contention occurred.

The IBM J9 Virtual Machine uses Tasuki locks as described in [25]. If a thread tries to lock on a variable, at first the lock is checked if it is free (Figure 2.2.1). If it is free the calling thread occupies this variable. If it is not free the calling thread tries to inflate it by obtaining the object's monitor, entering the monitor, and then checking if the lock is not inflated. If it is not inflated it sets the inflation bit and then checks again if the lock is available. If it is free it inflates the lock by removing the inflation bit, acquiring the lock, and notifying all threads waiting on the monitor (Figure 2.2.2). If the lock is not available the process or thread waits on the monitor. The
difference between this approach and a traditional locking approach is that with the Tasuki lock no busy waiting will occur. Threads can continue in execution and will be notified if the lock is available. Therefore, a Tasuki lock is a hybrid approach to locking. Objects in the IBM J9 VM can be used to ensure consistency of variables if multiple threads work on this variable, e.g.

```java
public synchronized void doSomethingSynchronized() {
    ...
}

Object obj;

public void doSomethingSynchronized2() {
    synchronized(obj) {
        ...
    }
}
```

The method `doSomethingSynchronized()` uses itself as a monitor and the method `doSomethingSynchronized2()` uses the method object `obj` for monitoring [8].
Figure 2.2.2: Inflated lock structure [25]. The first bit shows whether a lock contention occurred. If a contention occurred a heavyweight monitor is instantiated and its pointer is saved in the flat lock variable.

2.3 Hashcode

Hashcodes are the values returned by a hash function. A hash function $h$ usually maps large data sets onto smaller data sets and can be mathematically described by:

$$h : \mathcal{X} \rightarrow \mathcal{Y}, \ |\mathcal{X}| > |\mathcal{Y}|, \ \mathcal{Y} = \{0, 1\}^n,$$

(2.3.1)

with $x \in \mathcal{X}$ being an element or a message to hash and $h(x) \in \mathcal{Y}$ being the corresponding binary hashcode with a fixed length $n$ [26].

Hashcodes are applied in order to efficiently find and identify larger objects. When checking all objects to look for one specific object it can take very long to check every bit of it in order to find if it matches the object that the program is looking for. Thus, it is faster to map all objects onto a smaller set so that comparisons can be computed efficiently. To avoid recalculation of the hashcode it makes sense to store the hashcode somewhere. This also avoids additional computational overhead and can speed up programs that rely on
recurring searches for objects. Popular hashing algorithms are MD5 [27] and SHA-1 [28].

The IBM J9 Virtual Machine is focused on processing speed as it is the underlying program of user applications. Therefore, it utilizes fast hashing algorithms to affect the runtime of the user program as little as possible.

2.4 Benchmarks

After changes have been made to a previously existing system or program these changes need to be evaluated. Different kinds of metrics exist that support this evaluation, such as benchmarks. A benchmark can be used to evaluate the performance of the original implementation against the optimization that has been done [29].

Dufour et al. described in [30] the following metrics for benchmarking programs: program size and structure, data structures, polymorphism, memory use, and concurrency and synchronization. With the results of benchmarking these metrics, the developer can compare each individual result against the original implementation and decide if it is useful to include these changes into the existing systems. Sometimes it can also happen that only in some metrics the new implementation is better than the old. Then a tradeoff needs to be made and it needs to be decided whether or not to include the new
Different benchmarks for the Java VM include the SPECjbb benchmark from the Standard Performance Evaluation Corporation [31] and the DaCapo benchmark from the DaCapo Project [32]. The DaCapo benchmark is a benchmark suite which has been released as open source [32].

2.5 Related Work

This section will show what research has previously been done in the field of monitors as well as dynamic monitor allocation. As the JVM environment is only one field where monitors are used, related work in other areas will also be covered. After that, Section 2.5.2 will introduce the problem of this thesis by showing why the existing solutions are not sufficient and what can also be done. Chapter 5 discusses whether the research that has been made justifies integrating this dynamic approach into the JVM.

2.5.1 Dynamic Monitor Allocation

Most research for monitors in the JVM environment has been done on space and time efficiency. Dice introduced in [33] relaxed locks. A relaxed lock only uses one word in the object header and only needs one atomic compare-and-swap operation for locking and no atomic instructions for unlocking. Yang et al. showed in [34] an implementation of a lightweight monitor. They
embedded a 32 bit lock in each object for “efficient lock access while other monitor data structures are managed using a hash table” [34]. As already described in Section 2.2 the IBM JVM uses Tasuki locks for monitoring [25]. These locks are a hybrid approach as depending on the usage they are either thin and lightweight or can have a heavyweight monitor behind it when more than two threads are using them.

In [35] Bacon et al. showed their implementation of a thin lock, where they reduced the locking to only a single compare-and-swap operation. Furthermore, Bacon et al. showed in [36] a featherweight synchronization algorithm that allows to lock and unlock with “only a few” machine instructions. Other research includes reducing the memory footprint of the JVM. In 2006 Bacon et al. showed in [4] an approach on how to thin out an object by overloading parts of it. They were able to reduce the two word header of their JVM to a single word header. But their approach still used two word headers for objects that utilize the synchronized method. Kawachiya showed in [37] an improvement of the reservation lock, where “the reservation is not cancelled and the owner thread can always acquire its locks very quickly”. With this asymmetric lock they observed an improvement of the performance of up to 7.9%.

Agesen et al. showed in [38] an implementation of a meta-lock, which they showed is fast, compact, robust under contention, and flexible. Another ap-
approach can be done from the thread perspective and is shown by Bogda and Hölzle in [39]. They showed a “flow-insensitive, context-sensitive data-flow analysis” [39] that finds situations in which no synchronization is needed. Aldrich et al. described in [40] a similar approach that removes unnecessary synchronization.

While these approaches improve overall performance and memory overhead they did not address the core problem: every object, even if not used for locking, still has a lockword. When removing lockwords of objects that are not used for locking the memory overhead could be further reduced. For this Kahlon and Wang proposed in [41] the removal of redundant synchronizations by only using thread-local computation. With this approach they were able to completely remove some lockwords while avoiding synchronization problems. Usui et al. showed in [42] a technique that determines whether a synchronized block is best executed transactionally or with a mutex. This technique only applies to systems with transactional memory.

2.5.2 Problem Formulation

The JVM relies on space and time efficient programming as it is the bottleneck for every Java program. Thus, always new ways to accelerate the JVM and reducing its memory footprint are of interest. This includes using more efficient algorithms for already existing parts while still following the JVM specification. This specification limits the areas where enhancements can be
made, as every JVM needs to behave the same and handle Java programs accordingly.

Thus far no effort has been made to reduce every object’s memory footprint by only allocating a lockword to objects that are being locked on to. According to the JVM specification every object needs to be capable of being locked on to. Therefore, if not every object has a lockword during instantiation it needs to have the ability to dynamically grow a monitor if necessary.

As of now the, IBM J9 VM allocates a lockword for objects whose classes have been specified to receive a lockword. This means that also objects who are not used for locking have a lockword because their class has been specified to receive a lockword. Therefore, a lot of memory is potentially wasted. This thesis focuses on decreasing this footprint by only giving those objects lockwords that are either specified to have lockwords or that lock on to an object. After these changes have been made, the new approach needs to be evaluated to determine if the tradeoff between reducing the memory footprint while potentially increasing the computational requirements is feasible or if the performance is too adversely affected.
Chapter 3

Project Design

This chapter explains the design of this project. It will start off with the requirements for this project, which are based on the problem formulation as shown in Section 2.5.2. These requirements are introduced in Section 3.1. After that, the approach for this project is explained in Section 3.2. Based on the requirements and the approach, design decisions have to be made and these are formulated in Section 3.3. The framework used for this thesis is introduced in Section 3.3.1 and the software part of this thesis is presented in Section 3.3.2.

3.1 Requirements

For this research project, the requirements are as follows:

- The approach must be transparent to the Java programmer.
• The VM changes must be kept to a minimum.

• The VM must function as before internally except for the dynamic monitor part.

• Find a header bit to overload as every bit in the header already has a purpose.

• If an object has no monitor but is being locked, the VM should grow a monitor and use it.

• Evaluate the JVM with the dynamic monitor allocation approach.

In order to comply with the Java Virtual Machine [8] and the Java Language [7] Specification, the behaviour of the JVM as well as of any user program should not change. This means that the way the programmer locks on to an object does not change, i.e. that the programmer does not have to manually grow a monitor for an object. Furthermore, the user or programmer should not be able to see that any changes have been made except for memory and performance impacts.

The changes to be made in the JVM need to be kept to a minimum, so that as few files as possible need to be altered. Every change can create instabilities with other parts of the code, so that changes should only be made where they are necessary. If changes in different parts need to be made these changes need to be evaluated if they are working correctly with and without
the new implementation active. This leads to the requirement that the JVM needs to function the same way as without the dynamic monitor approach.

A bit in the object header needs to be used in order to determine whether the object has a dynamically allocated lockword. Therefore, at first it needs to be investigated which of these header bits can be used for this purpose. This can be achieved by checking how often various bits are used in different scenarios. In the JVM, objects can been specified to not being assigned a monitor upon instantiation. If this is the case and the user program or the JVM tries to lock using this object the dynamic monitor allocation approach should grow a monitor for this object and then lock on to this newly grown lockword.

For the evaluation of this project two JVMs need to be compared against each other. The first JVM applies the dynamic monitor allocation approach and the second JVM uses the same code but without the new approach. Metrics for evaluation include the performance and memory impact, as well as how many objects grow monitors in different scenarios. Details of the evaluation approach are provided in Section 5.1.
3.2 Approach

To be able to meet the requirements from Section 3.1, the general task needs to be composed into different subtasks. The first task is determining which of the header bits can be used for being overloaded with the indicator that this object has a grown lockword. This can be achieved by creating statistics of how often different bits are being used. For the creation of these statistics different scenarios will be utilized as will be described in Section 4.2.

After it has been determined which bit is being utilized for indication of having a grown lockword the next part will be the actual growing of a lockword. This task can be split into several different subtasks: at first the object needs to be moved, as the object does not necessarily have any space left either before or after its memory location. Thus, it needs to be moved to a location where it has enough space to store additional information. The second part is then the instantiation of a lockword and storing a flag in the header signalling that this object has a grown lockword.

After that, the approach needs to be integrated into the current system. This means that it needs to be determined where the VM tries to lock on an object. Then changes need to be made so that the VM can utilize the new approach if the object does not have a lockword.
The JVM as such can be classified as part of an agile software development with an iterative and incremental development. This means that new features will be extended in every new iteration with one of these features being the dynamic monitor growth. Therefore, a classical top-down design and bottom-up implementation can only be used to some extent.

3.3 Design Decisions

With the requirements defined in Section 3.1 and the approach formulated in Section 3.2, the decisions for the design of this research project can be made. The design decisions for the hardware will be shown in Section 3.3.1 and the software decisions will be introduced in Section 3.3.2.

3.3.1 Hardware

As the JVM runs on various hardware and software configurations it needs to be decided on a sample setup for this project. As the University of New Brunswick (UNB) and IBM supplied their Center for Advanced Studies (CAS) laboratory with computers, these will be used for implementing this project. The machine used for this project consists of an Intel Core i7-2600 processor, which consists of four cores and due to its hyperthreading ability it can utilize eight threads simultaneously. Furthermore, this computer features eight gigabytes of RAM.
This machine will be used for the entire project. The evaluation of this project will also be done on this hardware. Section 6 will discuss other benchmarks that need to be done on other hardware for a more thorough evaluation.

3.3.2 Software

The hardware specified in Section 3.3.1 features an Intel x86 compatible CPU, therefore, only software compatible to this architecture is used. With the Microsoft Windows operating system being one of the most widely used operating systems for consumers, the decision has been made to use Microsoft Windows 7 Professional (64 bit version). With IBM being one of the Eclipse Strategic Developer Members [43] and Eclipse an IDE that can be used for large scale projects, the Eclipse IDE will be used for this project as well.

The IBM JVM has been written in C and C++ programming languages, as well as in the IBM proprietary Builder programming language. These three languages are supported by the Eclipse IDE either natively or with plugins and therefore, these languages will also be used for the implementation of this project.
Chapter 4

Realization

This chapter describes the implementation side of this project. It starts with explaining the structure of the code in Section 4.1. After that, Section 4.2 shows how the percentage of hashed objects has been determined, which is a requirement for the remainder of the project as a header bit needed to be overloaded for this approach. In Section 4.3 the architecture is explained, with the implementation of the overloading technique shown in Section 4.3.1. Section 4.3.2 then shows how to create a new slot in an already existing object. The forced-moving of an object is described in Section 4.3.3. Finally, the interaction of all different parts is covered in Section 4.3.4. This section also includes the changes required to integrate this approach into the JVM.
4.1 Code Structure

The JVM code can be segmented into the following components:

- Builder
- Garbage Collection
- Interpreter
- JIT
- Natives
- Out-of-Process Debugging
- Port Library
- Realtime
- Thread Library

Each component consists of different projects. Builder contains all object and method definitions. All Builder projects are written in IBM’s proprietary Builder language and they contain the bootstrapper, common data structures, the JVM definition, the memory model definition, as well as different utilities to build the corresponding C/C++ and assembly files.

The Garbage Collector includes the memory manager, which is responsible
for managing access to the heaps by having different heap iterators. Furthermore, it includes the object model which is used for example in determining the size of an object, retrieving the hashcode, evaluating header bits, as well as determining the type of the object. The Garbage Collection part also consists of the different GC techniques as well as their concurrent and realtime counterparts.

In the Interpreter component, all implementation for the execution of Java code can be found. As the IBM JVM supports interpreted as well as JIT compiled code the JIT counterpart can be found in the JIT component. The JIT contains the Just-in-Time compiler for different architectures, including the ARM, PowerPC, or Intel x86 architecture.

The Natives component contains everything needed in order to load the VM. Therefore, it consists of the bootstrapper, compiler initialization, and the Java Class Library (JCL) loading. The VM also supports out-of-process debugging with which it is possible to test and debug JVM code with Java programs that test components of the JVM. This out-of-process debugger can check the GC, the JIT, the interpreter, and other components of the code for their correctness.

The abstraction to the underlying system is achieved with the Port Library. Every system call and datatype is mapped onto JVM functions which behave...
the same on all systems.

The Realtime component contains alternate implementations for every part of the VM that is realtime critical. Whenever for instance a GC is triggered, the execution of the main program is paused. If at any point during the GC a realtime decision needs to occur, it would fail with the conventional implementation. The realtime component handles these situations.

Similar to the Port Library, the Thread Library contains abstractions for the usage of threads independent to the underlying operating system. As the usage of threads is different on every operating system (for instance threads on Windows and Linux behave differently) the Thread Library maps VM threads onto system threads.

4.2 Evaluation of Hashcode Bits in an Object

One possible approach to store the lockword is by overloading the hashcode bits in an object. Therefore, it needs to be evaluated if the percentage of objects that are hashed is small enough in order to consider these bits for overloading. Thus, different approaches were applied in order to obtain an estimated likelihood of how many objects are hashed during the execution of Java code:

1. A single GC call
2. SciMark benchmark

3. SPECjbb2005 benchmark

The first item is a self-written program which only calls the Garbage Collector (GC) and then exits:

```java
public static void main(String args[]) {
    System.gc();
}
```

The second item is a mathematical Java benchmark developed by the National Institute of Standards and Technology (NIST) [44]. The last item in the above list is a Java server benchmark developed by the Standard Performance Evaluation Corporation (SPEC) [45].

With these programs, it is possible to estimate the number of objects that are hashed in a Java program. For evaluation the object count and the count of hashed objects during runtime is used in order to have a bigger sampling size. This has been done by iterating over all objects on the heap after every GC operation:

```javascript
function countHashedObjectsIterator(item) {
    if current item is an object
        if object has been hashed
            objectsHashed++;
            objectCount++;
```
function GC_PostCollector()
    for all objects
        countHashedObjectsIterator(object)
    fileOutput(time, objectsHashed, objectCount)

This code block iterates over all objects on all heaps and evaluates if the object has been hashed. This segment is run every time the GC is finished. If it has been hashed the variable objectsHashed is incremented. It also increases the variable objectCount whenever the current heap segment is an object. Furthermore, it also creates a file on the harddrive and writes the current program runtime in seconds, as well as the current object count and the count of all hashed objects in it, so that charts can be computed from these statistics. All of aforementioned programs are executed multiple times in order to receive stable results.

The object count and the hashed object count can be seen for the self-written program in Figure A.1.1 and A.1.2, for the SciMark benchmark in Figure A.1.4 and A.1.5, and for the SPECjbb2005 benchmark in Figure A.1.7 and A.1.8. For the self-written program the hashed object percentage is as shown in Figure A.1.3, the percentages for the SciMark benchmark are shown in Figure A.1.6, and the percentages for SPECjbb2005 benchmark are shown in Figure A.1.9. Especially in Figure A.1.9 it can be seen that the longer a Java program is running the lower the percentage of the hashed objects is.
With these scenarios it can be said that in an average Java program less than one percent of the objects are being hashed. Thus, the hashcode bit can be used for overloading its functionality with the dynamic lockword.

4.3 Architecture

Since less than one percent of the objects are hashed, hashcode bit can be overloaded with a flag for the dynamic lockword. Section 4.3.1 shows how this particular bit is overloaded. After that Section 4.3.2 shows how the dynamic lockword is instantiated. As an object needs to be relocated for the instantiation of the lockword, Section 4.3.3 describes how to force-move an object while also preserving all references to it.

Section 4.3.4 describes how all components are integrated into the VM. There are different subtasks that need to be considered, these are:

- Object is created.
- Object is moved.
- Accessing the lockword.
- Accessing the hashcode.

Certain object types can be specified to receive lockwords upon instantiation. This means that for the first scenario the configuration needs to be adapted
to remove lockwords for objects which are unlikely to be locked. Then the VM will instantiate objects of that kind without a lockword. Currently, only the GC can move an object, therefore the GC needs to verify if the object that it wants to move already possesses a lockword. If it does not own a lockword it would then need to grow it. If it already has a lockword, it needs to copy it. Therefore, it needs to distinguish between a lockword and a dynamic lockword.

If the lockword is accessed, a different verification needs to be done: the VM needs to determine whether the object already has a lockword from instantiation or if it received a lockword when it was moved. If the object does not have a lockword, it needs to grow a lockword. After that the VM has to return the correct lockword.

Accessing the hashcode results in two possible scenarios: (1) the object has been moved or (2) the object has not been moved. For the first scenario the object already has a hashcode and lockword, thus nothing further needs to be done, as accessing the hashcode does not change anything in the lockword. For the latter scenario, the object has not been moved yet and the object also does not have the hashcode. This means that the VM needs then to calculate the hashcode for the object. The hashcode is not stored at this time, therefore the object stays at the same spot and no growing of the lockword is required at that time as well. Section 4.3.4 will show how these situations
have been realized.

4.3.1 Overloading Header Bits

The object header in the IBM J9 VM holds flags for various alterations that can be made to an object, e.g. to indicate whether an object has been hashed. For this the object contains an additional status field whose bits represent different functionality.

Overloading a header bit means, that both functions of the bit are applied at the same time. Thus, the bit in the header with the lowest usage needs to be found. Section 4.2 showed the likelihood of objects that are hashed during the execution of different programs. GC bits are used more often than hash bits because a GC occurs more often than a function requesting an object’s hashcode. With a usage of less than one percent for the hashcode bit, the decision has been made to overload the bit indicating whether an object has been moved.

As stated in Section 3.1 the changes that are introduced to the VM code must be kept to a minimum. Therefore, the name for the flag indicating whether the object has a hashcode is not changed but it is now used for indicating that the object has a grown lockword as well. This means that no change needs to be made to existing code and only new code needs to make use of this flag.
4.3.2 Creating a New Slot in an Object

In order to create a new slot for the lockword in an object an approach similar to creating the hashcode for an object is used. This approach needs to be applied to all cases that are stated in Section 4.3. Previously, the VM appended a hashcode at the end of the object, once the object is moved. This can be seen in Figure 4.3.1. The dynamic monitor allocation technique utilizes the same technique in order to append the lockword. Figure 4.3.2 shows a possible layout for a moved object.

![Diagram of object layout before and after moving](image)

**Figure 4.3.1:** The current layout of an object before and after it has been moved. Before moving, an object does not incorporate an extra word for additional information. Once moved, an additional field is created which is used to store the hashcode.

These figures are simplified as they do not show for instance a previously owned lockword. As it can be decided that an object does not need a lockword the object’s class lockword offset can be indicating that the derived objects do not have a lockword.
The functionality to move an object already exists in the GC. With this technique the GC is able to move objects that are required to be moved during its compaction stage. Section 4.3.3 shows how an object can be moved without invoking a full GC by borrowing its logic to move an object.

To now be able to determine the new location of the lockword, modifications to the object model need to be done. The IBM J9 VM distinguishes between two different objects, an array object and a mixed object. Hence, the global object model and the specific object models need to be updated. The global object model incorporates two new methods in order to be able to retrieve the correct lockword offset and to initialize a new lockword in an object.
This new approach also needs modification in the retrieval and checking for the lockword as a lockword is now no longer at a predefined location. The check to see if an object has a lockword is now realized as follows:

```javascript
function checkIfObjectHasALockword(whichObject)
    if object.class has a lockword
        return true
    if object has been moved
        return true
    return false
```

Previously this method only determined whether the object has a lockword. As every object’s class has this variable but not every object has a lockword it needs to be determined if the class provides a lockword. The method then determines if the object has been moved, because then the object would have grown a lockword. If neither of the two statements are true the method fails and the caller knows that the object does not have a lockword and would then need to dynamically grow a lockword.

In order to retrieve the lockword the object needs to process the following code:

```javascript
function getLockword(object)
    if object has been moved
        if object is arrayObject
            return lockword for this arrayObject
        else
            ...
```

42
return lockword for this mixedObject
else
    return lockword by using offset provided by the class

If it has been determined that the object has been moved and that it is an
array object, the lockword can be found at the address of the object plus the
lockword offset of this array object, so if an object whichObject is at memory
address 0x00010000 and the lockword offset would have a value of 32, the
lockword for this particular object can be found at the address 0x00010020.
The calculations for a mixed object or an object with a class lockword offset
can be calculated analogous to the calculations for an array object.

4.3.3 Force-moving an Object

Section 4.3.2 has shown how an additional slot in an object can be created.
To be able to do this in a live system, enough free space at the end of the
object is needed. As this is not always the case, the object needs to be moved
to a different location that has enough free space to store the additional data.
The following pseudocode shows how this is realized in the IBM J9 VM:

function force-move_an_object(object)
    if object has been moved before
        return object
    else
        allocate new object with size (old size + size of
          hashcode and lockword)
copy contents of the old object into the new object
get lockword and hashcode offset
write hashcode at hashcode location

// we should not end up here when we already have a
lockword (fail-safe implementation)
write either old lockword or 0 at memory location (object +
lockword offset)

set new object has been moved field

acquire exclusive access to heap
iterate all objects on all heaps
  if object is referencing the old object
    update reference to point at the new object
update new object
fix all roots to point at the new object where necessary

remove old object

release exclusive access to heap

return new object

At first it needs to be determined if the object has been moved before. If it
has been moved the object already owns a lockword and therefore does not
need to be moved again. Thus, the method can return the old object. If it
has not been moved before the VM asks the memory manager to allocate a new object in tenured space which is eight bytes wider. These eight bytes will allow the storage of the hashcode and the lockword.

After that the contents of the old object are copied into the new object. This means that for instance if object old is located at memory address 0x000100020 with a length of 32 bytes and if the memory manager creates a new object at e.g. the address range 0x00010500 to 0x00010528 which has the size of 40 bytes, the algorithm would start at position 0x000100020 and copy the contents byte-wise over to memory address 0x00010500 for all 32 bytes of the original object.

As the old object has then been copied to a new location the lockword and hashcode offsets are retrieved. Then the hashcode of the old object is calculated and written into the hashcode slot of the new object. The hashcode of the old object needs to be used as this is the ID of the object for as long as the object is alive. Using the hashcode of the new object might lead to undesirable behaviour as it then might be treated as a new object.

If the original object already has a lockword it is then copied into the lockword slot. If not, a new lockword will be computed. Every new lockword is initialized with the value zero, as locking on this thin lock is done with an atomic compare-and-swap operation, which checks if the lockword is zero
and then exchanges the zero with the thread’s address. The case that the object already owns a lockword should never occur and is just implemented as a fail-safe as the retrieval of the lockword prefers a grown lockword over an already instantiated lockword.

At this point the object has been fully instantiated and therefore the header flag for showing that the object has been moved is toggled. With this flag the VM then knows that this object contains a hashcode as well as a lockword. To be able to use this object all references to the old object need to be updated. Therefore, at first exclusive access to the heap is needed. Then an iterator is instantiated which iterates over all objects in the VM. As every object might have more than one reference to other objects all references need to be evaluated if they point to the old object and need to be updated if necessary.

Then the new object can be updated as well. For this it copies all references that the old object contained. As not only objects might refer to the old object, in the next step all roots need to be traversed and updated as well. The root set is the "internally derived set of references to the contents of the stacks and registers of the JVM threads and other internal data structures at the time that the GC was called" [46].

Finally, the old object can be removed as the new object incorporates ev-
everything from the old object and the old object is not referenced any more.
At this point exclusive access to the heap can be released as well. The new
object is then returned to the calling method as this method might otherwise
operate on the old object.

To be able to access this force-move method it needs to be part of the memory
manager. Only GC related methods have the correct scope to easily manip­
ulate the heaps and thus the force-move method has been put in there. To
allow the VM as well as VM threads to call the force-move method it needs
to be part of the memory manager. The memory manager handles memory
based operations that are required to occur outside of the GC, e.g. allocat­
ing a new object can also be called by the VM or a thread of the VM but
the GC needs to take care of the actual allocation. Section 4.3.4 shows how
the force-move method is integrated into the IBM J9 VM. Currently, the
force-move method only works for mixed objects and contiguous arrays, as
non-contiguous arrays are scattered over different parts of the heap. Chapter
6 talks about future improvements to this method.

4.3.4 Integration into the Virtual Machine

The final part is to integrate this new approach into the existing JVM code.
As explained in Section 4.3.3 the JVM can access memory manager functions
outside of a GC call. Functions that need to be accessible outside of a GC
call need to be specified in the Builder code. Thus, the force-move method
has been made part of the memory manager.

Once an object has been determined to dynamically grow a lockword the `getDynamicMonitorForAnObject` method is triggered. The algorithm for acquiring a dynamic monitor is as follows:

```plaintext
function getDynamicMonitorForAnObject(object)
    if object does not have a lockword
        try and acquire growing_mutex
        if we cannot acquire mutex
            make our object visible to the gc
            release exclusive access to the heap
            acquire growing_mutex
            acquire exclusive access to the heap
            if object has not been updated
                force-move_an_object(object)
            release growing_mutex
        else
            force-move_an_object(object)
            release growing_mutex
    return object
```

Figure 4.3.3 shows this algorithm as a flowchart. When entering this method, the currently active thread owns exclusive access to the heap. If the object already has a lockword this method does nothing and returns the object. If not it tries to acquire the monitor update mutex `growing_mutex`. This mutex is used in this method to only allow one thread to dynamically grow
a monitor for a given object and is initialized during creation of the JVM. As the JVM spawns multiple threads and because of Java programs running in parallel there is a possibility that multiple threads try to retrieve the lockword for the same object simultaneously. This would result in one object being moved to two different locations and ultimately cause the JVM to crash or not exit properly as both objects are being locked but only one of them will be unlocked.

If the mutex cannot be acquired this algorithm knows that another thread is currently updating an object. This acquisition call is non-blocking in order to prevent a deadlock situation. A deadlock can occur in this function if one thread is holding exclusive access to the heap and a second thread is holding the monitor update mutex, but both threads are actively waiting on the respective other mutex. Thus, if the monitor update mutex cannot be acquired and with the knowledge that another thread is currently updating an object the calling thread knows that the other thread needs exclusive access to the heap to update the monitor. Therefore, this thread makes the object pointer visible to the GC and releases exclusive access to the heap to the other thread. The thread needs to make this object pointer visible to the GC as the other thread might be updating the same object. With the GC now being aware of that object the force-move function will then update this object pointer as well. The thread then waits actively on the monitor update mutex to be released.
Figure 4.3.3: Flowchart for updating an object. If an object needs a lockword two mutexes are acquired in order to avoid deadlocks. Once the object has been moved these mutexes are released.
The second thread can in the meantime acquire exclusive access to the heap and finish updating the object. After the second thread is done it releases exclusive access to the heap and the monitor update mutex and returns the updated object. Then the first thread, which is still actively waiting on the monitor update mutex, can acquire this mutex as well as exclusive access to the heap. After that the thread needs to check if the object has been updated and if not it calls the force-move method. Once this is done it releases the monitor update mutex and returns the object. Figure 4.3.4 shows the avoidance of the deadlock situation when two threads try to update the same object at the same time in an UML-like sequence diagram. Additional threads would have the same behaviour as can be seen for Thread 2.

Builder generates assembly code that calls these methods. The Java bytecode interpreter has been modified to check every object for a lockword as soon as the execution requires the object to enter a monitor. As soon as the bytecode interpreter detects that a monitor operation is required it pushes the required object on top of the execution stack. Then upon entering the monitor this object is popped off the stack and checked for a lockword. If it has none, a lockword will be grown by force-moving the object and allocating the lockword. Then this object can enter the monitor either blocking or non-blocking with this lockword.
Figure 4.3.4: UML-like sequence diagram for handling a possible deadlock situation on updating an object. The additional mutex which is only used for the getDynamicMonitorForAnObject method avoids a deadlock situation but enforces active waiting.
For the JIT part, a similar approach is used. As soon as an object is used for locking, the same check will be done before actually locking on the object. This ensures that the object, no matter if used in interpretation or in compiled code, has a lockword. An additional check has been implemented in the actual enter methods for the blocking and non-blocking approach. This is to ensure that all cases which are not covered by the previous approach are handled correctly by performing an additional check if the object has a lockword.
Chapter 5

Evaluation

To be able to compare this approach against the existing implementation, the project needs to be evaluated at first. Section 5.1 explains how the evaluation of the project has been done, as not only the implementation evaluation is of interest but also its performance compared to the existing JVM. After that, each part is evaluated in Section 5.2. Section 5.2.3 assesses the integration of this approach by doing an overall evaluation. Once the project has been evaluated it can be benchmarked in order to determine its performance and to compare it against the existing implementation. This is done in Section 5.3. Before concluding this chapter a discussion about the usefulness for the JIT to incorporate this approach is given in Section 5.5. Finally, the results of the benchmarks and the feasibility of this approach are discussed in Section 5.4.
5.1 Approach

In order to evaluate this project the implementation needs to be verified for correctness first. As the Java Virtual Machine needs a Java program to execute code different Java programs are needed to test the functionality of this code. As this thesis focuses on locking, Java programs are needed that do locking in various ways and on a broad range of objects. These programs are used for the overall verification of the implementation. Each subpart can be individually verified internally with different checks in the JVM. This ensures that the approach is behaving as expected.

The comparison of the performance of the new approach over the existing implementation is done with benchmarks. As shown in Section 2.4 there are different benchmarks available that can test the performance and memory consumption of the JVM. A set of selected benchmarks is used to evaluate different extrinsic metrics. Furthermore, intrinsic metrics that these benchmarks cannot capture will be compared as well, by implementing the capture and output of these metrics into the modified and original JVM. These intrinsic metrics include the amount of objects that are being locked as well as the amount of objects with a grown lockword that are used for locking. With these metrics a conclusion on the overall rating of the approach can be drawn.
5.2 Evaluation of the Implementation

Before being able to evaluate the whole project every part needs to be evaluated independently. Therefore, the evaluation will be done in the same order as the implementation: from the smallest part to the biggest part. As nothing needs to be changed in the existing code for overloading the hashcode bit in the object's header the overloading part does not need to be evaluated for the existing code. This bit is also used in the dynamic monitor allocation and therefore it is evaluated in Section 5.2.1.

After the growing of the monitor has been verified for its correctness the force-moving of an object needs to be evaluated. This is done in Section 5.2.2. Finally, the integration of the force-move method into the existing JVM can be evaluated. Section 5.2.3 shows this.

5.2.1 Monitor Growing

After the implementation of growing a monitor, this method needed to be checked that the code was working as expected. Thus, different objects have been allocated in the JVM and were assigned grown monitors. Then these objects were checked immediately and later during execution if they still possessed a lockword. In order to achieve this behaviour the GC was used to carry out these operations. Whenever a GC is triggered the GC was updated to do the following:
function triggerGC()
    if evalObjects have not been instantiated
        for all evalObjects
            instantiate evalObject as moved object
            append lockword at evalObject
        for all evalObjects
            print evalObject.address
            print evalObject.lockword
    [...]  

At the first GC call the objects evalObjects have not been instantiated yet. Therefore the GC instantiates every object in the set of evalObjects as a moved object. As described in 4.3.2 a moved object has additional bytes of data appended to it where the first bytes are used for the hashcode and the remaining bytes for the lockword. Furthermore the lockword slot is initialized for every object and the lockword is appended for every object in the set of evalObjects. The set of evalObjects has been specified to contain a different amount of objects every run. The amount of objects was hardcoded into the GC. In the flattest state the lockword consists only of a number. Thus, different numbers were hardcoded for the lockwords in order to check that the lockwords are at the correct location and have the correct value. After that the GC prints out the object’s address as well as the object’s lockword for every object in the set of evalObjects.

The Java program used to check these objects for correctness is as follows:
public class TestMonitors {
    public static void main(String args[]) {
        System.gc();
        System.gc();
        System.gc();
    }
}

This simple program triggers the GC three times and then exits. During the first GC call the evalObjects are instantiated. After that the GC outputs the address and lockword for every object in the set of evalObjects on the screen. The second and third GC call skips the first part as these objects are already instantiated and just outputs the addresses and lockwords.

With these outputs it was possible to manually check these objects for consistency, meaning that the object stayed at the same location in the memory, and their corresponding lockwords. Possible output of this evaluation is as follows:

first GC iteration
object at address 0x22aa5210 has lockword 0
object at address 0x22aa5220 has lockword 0
object at address 0x22aa5230 has lockword 0

As the GC decides where to put the objects and because the operating system assigns the memory for the JVM the addresses of the objects change every
time the JVM is created. Therefore the addresses of the object are only valid for one execution run. The above example shows only the output of the first `System.gc()` operation and shows that in one case three objects with their addresses being `0x22aa5210`, `0x22aa5220`, and `0x22aa5230`, have all the same hardcoded value 0. The lockword of every object is retrieved with the `getLockwordOffset(object)` function as previously described in Section 4.3.2. The second and third GC call in the sample Java program yielded the same results for all objects. The memory layout for this example can be seen in Figure 5.2.1. On the left side of the figure the memory address can be seen in a hexadecimal format. The top of the figure depicts the current byte for that address, e.g. the first row and third column would be the byte at the address `0x22aa5210 + 2` which would be byte `0x22aa5212`. The values `xx` are arbitrary hexadecimal numbers with `0 ≤ x ≤ f_{16}`. In this simplified case the first four bytes in every line are the object’s header, the next four bytes the object’s body. These eight bytes make up the entire object. In this case only a four byte object has been allocated with the flag that the object has been moved. This flag then allows the GC to allocate additional bytes at the object’s end where in this case the first four bytes make up the hashcode and the remaining four bytes the lockword. In the above example the lockword was 0, therefore all four bytes have a value of 0.

Changing the lockword from the value 0 to arbitrary other integer numbers yielded this number as the output for all three GC calls. With these results,
it can be concluded that growing an additional field in an object is working.

5.2.2 Force-Moving an Object

The next part to evaluate is the force-moving of an object. For this the algorithm from Subsection 5.2.1 has been modified as follows:

```java
function triggerGC()
    if evalObject has not been initialized
        initialize evalObject
        initialize parentObject
        initialize childObject
        evalObject.reference(childObject)
        parentObject.reference(evalObject)
        print evalObject, parentObject, childObject
        print evalObject.lockword
        force-move_an_object(evalObject)
```

Figure 5.2.1: Possible memory layout for objects with grown lockwords. The value $x$ represents an arbitrary hexadecimal number. All three objects have a four byte lockword appended which is initialized as zeroes.
print evalObject, parentObject, childObject
print evalObject.lockword

This function initializes on the first call the objects evalObject, parentObject, and childObject. After that the method updates evalObject to reference childObject, and parentObject to reference evalObject. This hierarchy can be seen in Figure 5.2.2. With this hierarchy it is possible to evaluate if updating all objects work as well as if updating the object to force-move works. Updating all objects can be verified by checking if the reference of parentObject points to the correct evalObject even after moving. The correctness for the update of the object can be verified if the moved object still points to childObject after the force-move.

![Hierarchy of objects](image)

**Figure 5.2.2:** Hierarchy of the objects used to evaluate the force-moving method. The object to evaluate is evalObject. The relationship of two other objects to evalObject have been defined by the arrows.

After the hierarchy has been established statistics about these three objects
are displayed. These statistics include the addresses of these objects, as well as the object they are referencing. Furthermore, the lockword for `evalObject` is displayed. Then the object `evalObject` is force-moved. This is only done on the first time the GC is triggered. Then, every time a GC is triggered the addresses and references for the objects `evalObject`, `parentObject`, and `childObject` are displayed, as well as the lockword for `evalObject`. The output can look as follows:

```plaintext
evalObject 0x22aa5210
childObject 0x22aa5200
parentObject 0x22aa51f0
parentObject 0x22aa51f0 pointing to evalObject 0x22aa5210
evalObject 0x22aa5210 pointing to childObject 0x22aa5200
evalObject 0x22aa5210 lockword is 0
(before move) evalObject 0x22aa5210
(after move) evalObject 0x22aa5220
parentObject 0x22aa51f0 pointing to evalObject 0x22aa5220
evalObject 0x22aa5220 pointing to childObject 0x22aa5200
evalObject 0x22aa5220 lockword is 0
```
Looking at the memory for the range between 0x22aa51f0 and 0x22aa5220 before moving the object is as shown in Figure 5.2.3:

![Possible memory layout before moving an object with three objects in the portrayed area.](image)

**Figure 5.2.3:** Possible memory layout before moving an object with three objects in the portrayed area.

After moving the memory is as shown in Figure 5.2.4:

![Possible memory layout after moving an object with three objects in the portrayed area.](image)

**Figure 5.2.4:** Possible memory layout after moving an object with three objects in the portrayed area.

Before moving the object `evalObject` (Figure 5.2.3) the space `free Space` is not occupied and can be used for allocation by the GC. During moving...
of the object the GC allocates a new object in the space free Space and before finishing moving the object the old object is deleted. This results in the object evalObject at the space of free Space and the deletion of the old object at the space evalObject, as shown in Figure 5.2.4.

5.2.3 Integration

Before the system can be benchmarked the integration of the approach needs to be evaluated as well. For this, a Java program has been written that would have two threads lock on the same object. This object does not have a lockword beforehand. The Java code for this program can be seen in the Appendix A.2.

This program creates a class called lockOnString that implements the Runnable interface, which means that this class can be part of a thread. The class has two class variables, a string str, which contains the character sequence STRING, as well as an integer that is instantiated with the value zero. This class has only one accessible method, which is add(). This method has the synchronized modifier, which means that the instantiated object's lockword from the lockOnString class is used for synchronization of this method. The method run() locks on the string str and does some computation on it in order to stay in this synchronized block long enough for the other threads to actively wait on the lock. The main method of this program locks on a different string and instantiates two threads and runs those.
These threads then enter the synchronized block and have to wait for the lock to be free in order to continue. To now be able to use the approach from this thesis for locking on the objects in this Java program, lockwords need to be removed from some object types. The IBM J9 VM features the possibility to remove lockwords from certain object types. For this either the command line toggle

```
-Xlockword:mode=default,noLockword=XYZ
```

with XYZ being the Java class to be removed, e.g. `java.lang.String` [47]. By default the IBM J9 VM has lockwords for strings disabled, thus above program works without any additional modification with the new approach.

In order to be able to see that objects have actually grown a monitor the implementation has been modified to display an output on the screen when an object has been moved. This output also includes the class name of the object. With this information it is then possible to see whether the correct objects have been moved. A possible output of running this program on the modified JVM can look as follows:

```
0x029361c8 (java/lang/String) moved to 0x029361f0, grown lockword
0x02936198 (java/lang/String) moved to 0x02944a10, grown lockword
STRINGThread-6
STRINGThread-6Thread-6
STRINGThread-6Thread-6Thread-6
```

65
The first two lines are the information the JVM displayed. It found two objects that were being locked but did not have any lockwords. These objects were at the memory addresses 0x029361c8 and 0x02936198. Both of these objects were of the type java/lang/String which previously was set to not have any lockwords. Thus, the JVM detected that these objects need lockwords in order to continue and therefore called the force-move method so that these objects would receive lockwords. The object at the address 0x029361c8 corresponds to the string t, which was declared in line 29 of the code in Appendix A.2 and used for synchronization in line 30. The second line refers to the object String str which is used by both threads for synchronization.

After both objects have been used for synchronization the first thread outputs the first three messages. In this example it would be the thread with the internal thread number six which holds the lock and outputs its three messages. The second thread with the thread number five waits for the first thread to release the lock and then utilizes it itself and outputs the remaining three lines.

Besides this sample Java program the SPECjbb2005 and DaCapo bench-
mark have been run in order to verify the functionality of this implementa-
tion. Section 5.3 talks about the way the benchmarking has been done as
well as the benchmarks used. The final results are shown and explained in
Section 5.4.

5.3 Benchmarking

In order to evaluate the implementation, different benchmarks have been
utilized. As most benchmarks are only capable of evaluating extrinsic pa-
rameters, e.g. CPU and memory usage, display outputs have been put into
the JVM to show intrinsic parameters as well. These overall list of param-
eters that are being evaluated include:

- Execution time.
- CPU usage.
- Memory usage.
- Number of objects.
- Number of moved objects.
- Number of blocking locking operations on moved objects.
- Number of non-blocking locking operations on moved objects.
- Overall number of blocking locking operations.
• Overall number of non-blocking locking operations.

The benchmarks used are SPECjbb2005 and DaCapo benchmarks. As these benchmarks do not lock very often on objects an additional program has been written that does more locking. Furthermore, the SPECjbb2005 does not take the CPU time into account that is used to dynamically grow the monitors, as this part is done before the actual measurements of the benchmark. The Java program written can be seen in Appendix A.3. This program instantiates ten threads and five objects without lockwords that are used for locking. Furthermore, every thread locks on every one of these objects and performs calculations while holding the lock in order for the other threads to actively wait on the lock release. After being done this program outputs the heap size as well as the time needed for execution.

To compare performance with respect to the dynamic monitor allocation this program is better than the SPECjbb2005 benchmark as it takes the new approach into account as well. SPECjbb2005 has a warmup phase before it actually benchmarks in which all objects are instantiated and tested. Therefore the results from this program are more accurate than for the SPECjbb2005 benchmark with regard to execution time.

Every benchmark has been run with the following command line parameters:

```
java -Xjvm:XYZ -Xint -Xgcpolicy:optthruput BENCH
```
with \textit{XYZ} being the JVM to benchmark and \textit{BENCH} being the benchmark to run. The \texttt{-Xint} toggle disables the JIT, which has been used for the first set of benchmarks. For the second set of benchmarks the JIT has been enabled to have more comparative results. The \texttt{-Xgcpolicy} toggle specifies the GC policy to use, in this case \texttt{optthruput} as the implementation only covers contiguous arrays and only this policy enforces contiguous arrays. With all other GC policies discontiguous arrays are possible. Furthermore, the default options for objects without lockwords have been used.

To be able to acquire stable results every benchmark has been run several times in the following order: \textit{thesis, thesis, thesis, reference, reference, reference, thesis, reference, thesis, reference}. The term \textit{thesis} specifies the JVM with the modifications from this thesis and \textit{reference} describes the default JVM on which the implementation has been made.

### 5.4 Results

The results of these benchmarks for SPECjbb2005 can be seen in Table 5.1 for the CPU usage and in Table 5.2 for memory consumption. These numbers are only preliminary numbers as IBM's official configuration has not been used for this evaluation. In both tables \texttt{DM JVM} refers to the dynamic monitor allocation approach that has been evaluated in this project and \texttt{RJVM} corresponds to the reference VM on which this approach has been built. The
CPU usage has been measured in business operations per second (bops). This unit cannot be directly converted in more well known formats, such as MIPS. Thus, the results have to be compared against each other to see the impact. The percentages of the dynamic monitor allocation approach compared to the reference JVM can be seen for the CPU and the memory usage in Table 5.3. Detailed results can be seen in Appendix A.4.1 for the CPU usage and in Appendix A.4.2 for memory consumption.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>8</td>
<td>12858.8</td>
<td>12770.6</td>
</tr>
<tr>
<td>9</td>
<td>12842.2</td>
<td>12769.4</td>
</tr>
<tr>
<td>10</td>
<td>12783.6</td>
<td>12713.4</td>
</tr>
<tr>
<td>11</td>
<td>12756.8</td>
<td>12693.4</td>
</tr>
<tr>
<td>12</td>
<td>12715.6</td>
<td>12627.4</td>
</tr>
<tr>
<td>13</td>
<td>12655.2</td>
<td>12605.6</td>
</tr>
<tr>
<td>14</td>
<td>12614.4</td>
<td>12566.8</td>
</tr>
<tr>
<td>15</td>
<td>12532.6</td>
<td>12465.8</td>
</tr>
<tr>
<td>16</td>
<td>12465.4</td>
<td>12413.4</td>
</tr>
<tr>
<td>Average</td>
<td>12,691.62</td>
<td>12,625.09</td>
</tr>
</tbody>
</table>

**Table 5.1:** Average CPU results of the SPECjbb2005 benchmark in business operations per second [bops]. DM JVM refers to the JVM incorporating the dynamic monitor allocation technique. RJVM is the reference JVM against which the dynamic monitor allocation technique is tested.
Benchmarking with the JIT disabled leads to a performance increase of 0.53% over the reference implementation. With the JIT enabled the performance increased 0.68%. This averages in an overall increase of 0.60%. These results do not take the actual growing part into consideration. The increase can be explained with these objects normally relying on heavyweight monitors, which take additional computation for locking and unlocking. In the new JVM these objects used a lightweight monitor which they possessed for locking.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>8</td>
<td>561</td>
<td>507.2</td>
</tr>
<tr>
<td>9</td>
<td>713.2</td>
<td>638.8</td>
</tr>
<tr>
<td>10</td>
<td>584.8</td>
<td>732.2</td>
</tr>
<tr>
<td>11</td>
<td>689.8</td>
<td>926.4</td>
</tr>
<tr>
<td>12</td>
<td>712.8</td>
<td>620.2</td>
</tr>
<tr>
<td>13</td>
<td>718.4</td>
<td>704.4</td>
</tr>
<tr>
<td>14</td>
<td>749.6</td>
<td>887.6</td>
</tr>
<tr>
<td>15</td>
<td>760.4</td>
<td>804.2</td>
</tr>
<tr>
<td>16</td>
<td>896</td>
<td>779.4</td>
</tr>
<tr>
<td>Average</td>
<td>709.56</td>
<td>733.38</td>
</tr>
</tbody>
</table>

Table 5.2: Average memory usage of the SPECjbb2005 benchmark in MB. DM JVM refers to the JVM incorporating the dynamic monitor allocation technique. RJVM is the reference JVM against which the dynamic monitor allocation technique is tested.
With the JIT disabled the heap decreased to 96.75% of the size of the reference VM. Enabling the JIT leads to a decrease to 93.31%. Both average to a decrease to 95.03%. As heavyweight monitors consume more space than lightweight monitors lots of space was saved for these benchmarks.

<table>
<thead>
<tr>
<th>Threads</th>
<th>CPU Usage</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JIT</td>
<td>no JIT</td>
</tr>
<tr>
<td>8</td>
<td>100.69 %</td>
<td>100.64 %</td>
</tr>
<tr>
<td>9</td>
<td>100.57 %</td>
<td>100.75 %</td>
</tr>
<tr>
<td>10</td>
<td>100.55 %</td>
<td>100.42 %</td>
</tr>
<tr>
<td>11</td>
<td>100.50 %</td>
<td>100.58 %</td>
</tr>
<tr>
<td>12</td>
<td>100.70 %</td>
<td>100.95 %</td>
</tr>
<tr>
<td>13</td>
<td>100.39 %</td>
<td>100.70 %</td>
</tr>
<tr>
<td>14</td>
<td>100.38 %</td>
<td>100.46 %</td>
</tr>
<tr>
<td>15</td>
<td>100.54 %</td>
<td>100.93 %</td>
</tr>
<tr>
<td>16</td>
<td>100.42 %</td>
<td>100.67 %</td>
</tr>
<tr>
<td>Average</td>
<td>100.53 %</td>
<td>100.68 %</td>
</tr>
</tbody>
</table>

Table 5.3: Percentage for the average memory usage and CPU usage of the SPECjbb2005 benchmark compared to the reference JVM.

The results of the benchmarks can also be seen in Figure 5.4.1 and 5.4.2. For the CPU usage results, longer bars indicate better performance and for the
memory usage results, shorter bars indicate less memory consumption.

For the DaCapo benchmark only the JVM without the JIT has been evaluated. The results can be seen in Table 5.4. The modified JVM had a performance increase of 1.45% over the reference implementation, which is due to the fact that the lightweight monitors perform faster than the heavyweight monitors that would have been there without this implementation.

![Comparison of the Average Performance](image)

**Figure 5.4.1:** Results for the CPU usage of the SPECjbb2005 benchmark. The Reference JVM refers to the JVM against which the JVM with the dynamic monitor allocation approach is tested. Both JVMs were tested with and without the JIT.

The results for the self-written locking benchmark can be seen in Table 5.5. The modified JVM had a performance increase of 14% over the reference
JVM, due to the locking only taking part in objects with grown lockwords. The reference JVM relied for these objects on heavyweight monitors as well.

Finally, the results for the intrinsic parameters are shown in Table 5.6. The column Obj specifies the number of objects that the JVM had instantiated during runtime, Mvd Obj the number of objects that were force-moved because they did not have a lockword, !Block specifies the number of non-blocking monitor enter calls in total, Mvd !Block the amount of non-blocking monitor enter calls for moved objects, Block specifies the number of blocking monitor enter calls in total, and Mvd Block the amount of blocking monitor enter calls for moved objects.

![Comparison of the Average Memory Consumption](image)

**Figure 5.4.2:** Results for the memory consumption of the SPECjbb2005 benchmark. The Reference JVM refers to the JVM against which the JVM with the dynamic monitor allocation approach is tested. Both JVMs were tested with and without the JIT.
As the self-written locking program only locks on objects that have grown lockwords, the amount of blocking monitor enter calls is 100%. Internally other objects utilize locks as well and they make up a big part of the non-blocking monitor enter calls. Thus, only 1.93% of all non-blocking monitor enter calls were related to the grown lockwords.

Finally, these results can be tweaked by carefully selecting the objects that do receive lockwords. Depending on the usage of the program disallowing
lockwords for certain objects might yield to different performance and memory improvements as this evaluation has shown. The JIT was not working for the DaCapo benchmark as well for the self-written Java programs. An explanation for this is shown in Section 5.5.

<table>
<thead>
<tr>
<th>Run</th>
<th>Obj</th>
<th>Mvd Obj</th>
<th>!Block</th>
<th>Mvd !Block</th>
<th>Block</th>
<th>Mvd Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11123</td>
<td>5</td>
<td>4344</td>
<td>84</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>11092</td>
<td>5</td>
<td>4342</td>
<td>84</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>11135</td>
<td>5</td>
<td>4341</td>
<td>84</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>11107</td>
<td>5</td>
<td>4343</td>
<td>84</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>11135</td>
<td>5</td>
<td>4346</td>
<td>84</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Average</td>
<td>11118.4</td>
<td>5</td>
<td>4343.2</td>
<td>84</td>
<td>35.4</td>
<td>35.4</td>
</tr>
</tbody>
</table>

Table 5.6: Statistics for blocking objects. "Obj" refers to the number of objects, "Mvd Objects" to the number of moved objects, "!Block" for non-blocking monitor enter calls for all objects, "Mvd !Block" for non-blocking monitor enter calls for moved objects. "Block" and "Mvd Block" refer analogues to the corresponding blocking monitor enter calls.

5.5 Usage for the JIT

As the JIT is not working for this implementation changes to the JIT need to be made. In the current state the JIT code has not been modified at all. A Java program might work without any problems with the JIT enabled in some cases. This is due to the fact that the JVM decides to use the interpreter for that specific part of the code rather than to use the JIT. As described in Section 2.1.3 the JIT is only utilized for code passages that are frequently called. As the self-written Java programs relied on the repetitive
execution of the same code the JIT might have been invoked to compile these code fragments. Thus, at the first call native code would try to lock the object without a lockword and would then cause the failure.

In order to adapt the JIT to the dynamic monitor allocation, the access of the monitor needs to be adapted to the same approach as for the interpreter. This includes the retrieval of the lockword, the check for the object possessing a lockword, as well as the monitor calls which can be blocking and non-blocking, and the force-move method. As the JIT compiles bytecode to native code, the JIT needs to be adapted on all different supported architectures, e.g. x86, ARM, and PowerPC.
Chapter 6

Conclusion and Outlook

This thesis has shown the successful implementation of the dynamic allocation of monitors for objects in the IBM J9 VM. With this approach the VM does not need to have monitors for objects anymore as they can be dynamically grown. As the memory in the JVM is managed by the GC the object needs to be moved to a new location before a monitor can be allocated for this object. Furthermore, the contents of the old object are moved to the new location. Then all objects on all heaps need to be checked and updated if necessary, so that these objects accept the new location of the object. The same applies to all roots. Finally, the old object can be safely removed.

The methods for retrieving the lockword as well as for checking if an object possesses a lockword were updated as well. In order to make the JVM use this implementation the interpreter has been adapted to check every ob-
ject for a lockword prior to locking and if none is present the interpreter grows a lockword.

Benchmarking has shown that the performance as well as the memory usage can be improved when using dynamic monitor allocation. Depending on how many objects receive dynamic monitors the performance and memory impact can be changed. This means that depending on the Java program the parameters for which objects receive lockwords need to be tweaked in order to receive better results. Section 5.4 has shown that using this approach on a program that heavily locks on strings with the strings not receiving lockwords beforehand, the performance increased up to 14%. With only strings being the objects to not receive lockwords beforehand other programs that do not solely rely on string locking receive a lower increase, e.g. the SPECjbb2005 benchmark has only an increased performance of 0.6%.

The main focus of this research was to reduce the memory footprint, thus the performance increase was a side effect. It has to be expected that the more objects do not receive lockwords during instantiation at some point the performance will be worse than with the reference implementation, as for every object that receives a dynamic lockword all heaps are updated. This will ultimately lead to lots of wasted computational time if done excessively.

Section 5.4 has shown that the memory footprint can be reduced to at least
95.03%. Unfortunately, the DaCapo benchmark did not output any heap sizes and for the self-written Java programs the heaps were too small to notice a difference. Thus, it can be expected that the more objects have their lockwords removed the smaller the footprint will be but a tradeoff needs to be found between memory footprint reduction and performance decrease.

As explained in Section 5.5 the JIT could benefit from this approach by incorporating these changes as well or by redirecting the monitor calls to the interpreter. Furthermore this approach can be improved by extracting more parts to Builder, which will then translate it into assembly code, rather than using C code. One example is the retrieval of the lockword offset for arrays. Upon checking an object for a lockword in assembly, Builder calls a C method to retrieve the lockword offset if it has determined that the object is an array object. For a mixed object, Builder code has been written to retrieve the lockword offset. This is only a minor example, as this example will not adversely increase performance. The current implementation does not cover non-contiguous arrays. Therefore, it can only be used with the optthruput GC policy. In order to make this implementation work with other GC policies, the case of non-contiguous arrays also needs to be covered in the force-move method.

Performance benefits could be made from introducing a queue for the force-moving. This means that whenever an object without a lockword needs to
be locked, it is added to the queue and after the queue is full or when a GC is triggered every object in the queue will be force-moved and receive a lockword. For this to work, lockwords need to be introduced as soon as they are needed, meaning that the object is enqueued and a lockword is instantiated somewhere on the heap and the object uses this lockword until it is force-moved. During the moving of the object the previously used lockword would then be written at its corresponding location within the object. With this approach the walking of all heaps would not need to be called as many times as objects are needed to be moved but only one time with a queue. During this one walk all objects are checked if they refer to the moved objects.
Bibliography


85


Appendix A

A.1 Hashed Objects

A.1.1 Dummy Program

Figure A.1.1: Number of objects in a simple dummy program.
Figure A.1.2: Number of hashed objects in a simple dummy program.

Figure A.1.3: Percentage of hashed objects in a simple dummy program.
A.1.2 SciMark Benchmark

Figure A.1.4: Number of objects in the SciMark benchmark.

Figure A.1.5: Number of hashed objects in the SciMark benchmark.
Figure A.1.6: Percentage of hashed objects in the SciMark benchmark.
A.1.3 SPECjbb2005 Benchmark

**Figure A.1.7:** Number of objects in the SPECjbb2005 benchmark.

**Figure A.1.8:** Number of hashed objects in the SPECjbb2005 benchmark.
Figure A.1.9: Percentage of hashed objects in the SPECjbb2005 benchmark.

A.2 Program for Evaluation of the Approach

```java
public class lockOnString implements Runnable {
    String str = "STRING";
    int i = 0;

    public synchronized void add() {
        i++;
    }

    public void run() {
        synchronized(str) {
            str += Thread.currentThread().getName();
            System.out.println(str);
        }
    }
}
```
try {
    Thread.sleep(1000);
} catch (Exception e) {} 
str += Thread.currentThread().getName();
System.out.println(str);
try {
    Thread.sleep(1000);
} catch (Exception e) {}
str += Thread.currentThread().getName();
System.out.println(str);
add();
}

public static void main(String[] args) {
    Runnable los = new lockOnString();
    String t = "T";
    synchronized(t) {
    Thread one = new Thread(los);
    Thread two = new Thread(los);
    one.start();
    two.start();
    }
}

A.3 Program for Benchmarking
public class LockHeavy implements Runnable {
    static final int X_RESOLUTION = 1000;
    static final int Y_RESOLUTION = 1000;
    static final int threadCount = 10;

    static int threadDone = 0;

    int maxiter = 10;
    int threadNumber;
    static String str[];
    static int maxStr = 5;

    public LockHeavy(int thrNumber) {
        this.threadNumber = thrNumber;
    }

    public synchronized void calculateMandelbrot(double xmin, double xmax, double ymin, double ymax) {
        double dx, dy;
double absvalue, temp;
double z_real, z_imag, c_real, c_imag;
int k;

dx = (xmax - xmin) / X_RESOLUTION;
dy = (ymax - ymin) / Y_RESOLUTION;

/* calculate values for every point in complex plane */
for (int i = 0; i < X_RESOLUTION; i++) {
    for (int j = 0; j < Y_RESOLUTION; j++) {
        /* map point to window */
        c_real = xmin + i * dx;
        c_imag = ymin + j * dy;
        z_real = z_imag = 0.0;
        k = 0;

        // do iterations for point i,j
        do {
            temp = z_real * z_real - z_imag * z_imag + c_real;
            z_imag = 2.0 * z_real * z_imag + c_imag;
            z_real = temp;
            absvalue = z_real * z_real + z_imag * z_imag;
            k++;
        } while (absvalue < 4.0 && k < maxiter);
    }
}

96
public void run() {
    for (int i = 0; i < maxStr; i++) {
        synchronized (str[(this.threadNumber + i) % maxStr]) {
            calculateMandelbrot(this.threadNumber * X_RESOLUTION
                / this.threadCount, ((this.threadNumber + 1) * X_RESOLUTION / this.threadCount) - 1, 0, Y_RESOLUTION);
        }
    }
    threadDone++;
}

public static void main(String[] args) {
    Thread pool[] = new Thread[threadCount];
    long timeStart, timeEnd;

    str = new String[maxStr];
    for (int i = 0; i < maxStr; i++)
        str[i] = String.valueOf(i);

    // --------------- benchmark start ---------------
    timeStart = System.currentTimeMillis();
    for (int j = 0; j < threadCount; j++) {
        pool[j] = new Thread(new LockHeavy(j));
        pool[j].start();
    }
while (!(threadDone == threadCount));
timeEnd = System.currentTimeMillis();
System.out.println("Heap:\u" + (1.0 * Runtime.getRuntime()
    .totalMemory() / 1024 / 1024) + "uMB");
System.out.println("Time\uneeded:\u" + (timeEnd - timeStart
    ) + "ums");
}
## A.4 Benchmark Results

### A.4.1 CPU Results for SPECjbb2005

<table>
<thead>
<tr>
<th>Run</th>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>8</td>
<td>12,883</td>
<td>12,737</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>12,852</td>
<td>12,808</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>12,787</td>
<td>12,776</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12,774</td>
<td>12,742</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12,725</td>
<td>12,656</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12,679</td>
<td>12,616</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>12,638</td>
<td>12,583</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12,537</td>
<td>12,480</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>12,416</td>
<td>12,426</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>12,699.00</td>
<td>12,647.11</td>
</tr>
<tr>
<td>Run</td>
<td>Threads</td>
<td>Without JIT</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
<td>DM JVM</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>12,879</td>
<td>12,807</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12,858</td>
<td>12,775</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12,794</td>
<td>12,670</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12,750</td>
<td>12,672</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12,738</td>
<td>12,614</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>12,678</td>
<td>12,630</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>12,636</td>
<td>12,579</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>12,554</td>
<td>12,464</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12,491</td>
<td>12,429</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>12,708.67</td>
<td>12,626.67</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>12,826</td>
<td>12,788</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12,817</td>
<td>12,766</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12,735</td>
<td>12,740</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12,763</td>
<td>12,730</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12,708</td>
<td>12,612</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>12,643</td>
<td>12,572</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>12,603</td>
<td>12,559</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>12,509</td>
<td>12,417</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12,474</td>
<td>12,395</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>12,675.33</td>
<td>12,619.89</td>
</tr>
<tr>
<td>Run</td>
<td>Threads</td>
<td>Without JIT</td>
<td>With JIT</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12,875</td>
<td>12,768</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12,840</td>
<td>12,752</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12,764</td>
<td>12,726</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12,725</td>
<td>12,676</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12,733</td>
<td>12,660</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>12,627</td>
<td>12,646</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>12,585</td>
<td>12,578</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>12,563</td>
<td>12,501</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12,472</td>
<td>12,413</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>12,687.11</td>
<td>12,635.56</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12,831</td>
<td>12,753</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12,844</td>
<td>12,746</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12,838</td>
<td>12,655</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12,772</td>
<td>12,647</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12,674</td>
<td>12,595</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>12,649</td>
<td>12,564</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>12,610</td>
<td>12,535</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>12,500</td>
<td>12,467</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12,474</td>
<td>12,404</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>12,688.00</td>
<td>12,596.22</td>
</tr>
</tbody>
</table>
Table A.1: CPU results of the SPECjbb2005 benchmark in business operations per second [bops]. DM JVM refers to the JVM incorporating the dynamic monitor allocation technique. RJVM is the reference JVM against which the dynamic monitor allocation technique is tested.

<table>
<thead>
<tr>
<th>Run</th>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
<td>DM JVM</td>
</tr>
<tr>
<td>Overall Average</td>
<td>12,691.62</td>
<td>12,625.09</td>
<td>12,709.93</td>
</tr>
</tbody>
</table>
### A.4.2 Memory Usage Results for SPECjbb2005

<table>
<thead>
<tr>
<th>Run</th>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>540</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>693</td>
<td>683</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>442</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1,004</td>
<td>978</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>720</td>
<td>686</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>727</td>
<td>671</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>609</td>
<td>949</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>777</td>
<td>716</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>904</td>
<td>839</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>563</td>
<td>583</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>724</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>485</td>
<td>442</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>479</td>
<td>970</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>727</td>
<td>651</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>743</td>
<td>639</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>613</td>
<td>909</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>779</td>
<td>670</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>924</td>
<td>794</td>
</tr>
</tbody>
</table>

| Average | 712.89 | 721.78 | 674.75 | 807.78 |

103
<table>
<thead>
<tr>
<th>Run</th>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td>Average</td>
<td>670.78</td>
<td>708.33</td>
<td>791.63</td>
</tr>
<tr>
<td>8</td>
<td>602</td>
<td>537</td>
<td>613</td>
</tr>
<tr>
<td>9</td>
<td>744</td>
<td>663</td>
<td>775</td>
</tr>
<tr>
<td>10</td>
<td>488</td>
<td>1,013</td>
<td>532</td>
</tr>
<tr>
<td>11</td>
<td>489</td>
<td>974</td>
<td>537</td>
</tr>
<tr>
<td>12</td>
<td>705</td>
<td>662</td>
<td>784</td>
</tr>
<tr>
<td>13</td>
<td>700</td>
<td>643</td>
<td>791</td>
</tr>
<tr>
<td>14</td>
<td>974</td>
<td>920</td>
<td>649</td>
</tr>
<tr>
<td>15</td>
<td>750</td>
<td>665</td>
<td>821</td>
</tr>
<tr>
<td>16</td>
<td>884</td>
<td>774</td>
<td>959</td>
</tr>
<tr>
<td>Average</td>
<td>704.00</td>
<td>761.22</td>
<td>687.75</td>
</tr>
</tbody>
</table>

<p>|        | 8 | 664 | 403 | 667 | 517 |
| 9 | 817 | 529 | 816 | 658 |
| 10 | 561 | 858 | 573 | 987 |
| 11 | 541 | 826 | 586 | 969 |
| 12 | 770 | 524 | 824 | 660 |
| 13 | 770 | 996 | 820 | 627 |
| 14 | 611 | 829 | 669 | 896 |
| 15 | 783 | 988 | 849 | 661 |
| 16 | 924 | 741 | 996 | 800 |</p>
<table>
<thead>
<tr>
<th>Run</th>
<th>Threads</th>
<th>Without JIT</th>
<th>With JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DM JVM</td>
<td>RJVM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>715.67</td>
<td>743.78</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>744.44</td>
<td>731.78</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>436</td>
<td>473</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>588</td>
<td>602</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>948</td>
<td>914</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>936</td>
<td>884</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>642</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>652</td>
<td>573</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>941</td>
<td>831</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>713</td>
<td>982</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>844</td>
<td>749</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>744.44</td>
<td>731.78</td>
</tr>
<tr>
<td>Overall Average</td>
<td></td>
<td>709.56</td>
<td>733.38</td>
</tr>
</tbody>
</table>

Table A.2: Memory usage of the SPECjbb2005 benchmark in MB. DM JVM refers to the JVM incorporating the dynamic monitor allocation technique. RJVM is the reference JVM against which the dynamic monitor allocation technique is tested.
Vita

Candidate’s full name: Marcel Dombrowski

University attended:
B.Sc. Computer Science 2010
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Nordrhein-Westfalen, Germany

Publications:


