Aggregating Performance Metrics of a Java Virtual Machine

ANDERS ÅSTRAND

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2008
Aggregating Performance Metrics of a Java Virtual Machine

ANDERS ÅSTRAND

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Stefan Nilsson
Examiner was Johan Håstad

TRITA-CSC-E 2008:125
ISRN-KTH/CSC/E--08/125--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se
Abstract

Java Virtual Machine (JVM) performance is multi-faceted. Not only are there different aspects of Java performance to consider, but workloads behave and affect a JVM differently depending on its characteristics in relation to these aspects.

This thesis demonstrates a method to characterize Java workloads from a performance point of view. A system for aggregating measurements from said workloads, depending their characteristics and user preferences, to a single performance index is implemented and presented.

Finally a set of tools were developed which enables the user to configure a specific index setup, explore the computed index structure as well as report it in various formats.
Referat

Sammanställning av prestandamätningar från en Java Virtual Machine

Prestandan av en virtuell Javamotor (JVM) är nyanserad och flersidig. Förutom att det finns mer än en aspekt av Javas prestanda att ta hänsyn till så beter sig och påverkar arbetslaster en JVM annorlunda beroende på deras karakteristisk i relation till dessa prestandaaspekter.

Under denna studie har det tagits fram en metod för att karakterisera arbetslaster som exekveras i en JVM utifrån en prestanda-synvinkel. Utöver det har ett system för att sammanställa prestandamätningar från nämnda arbetslaster till en enda prestandasiffra, eller index, implementerats. Indexet är beroende på arbetslasternas karakteristik och användarens preferenser.

Slutligen har ett antal verktyg tagits fram för att kalibrera, utforska och rapportera det beräknade indexet.
# Contents

1 Introduction 1
   1.1 Reason for this thesis ........................................ 1
   1.2 Thesis goal .................................................. 1
   1.3 Problems to solve ........................................... 2
   1.4 Document structure .......................................... 2

2 Definitions and background 4
   2.1 Nomenclature .................................................. 4
   2.2 Java Virtual Machines and performance ....................... 4
      2.2.1 The big challenge ...................................... 5
      2.2.2 Memory management ..................................... 6
      2.2.3 Code generation ......................................... 8
      2.2.4 Thread management ...................................... 10
      2.2.5 Nature of the Java language ......................... 12
   2.3 Java as a commercial platform ................................ 13
      2.3.1 Server load characteristics ............................ 13
   2.4 JRockit ....................................................... 13
      2.4.1 Throughput ............................................... 14
      2.4.2 Determinism ............................................. 14
      2.4.3 Manageability ........................................... 14
   2.5 Performance .................................................. 15
      2.5.1 Throughput ............................................... 15
      2.5.2 Latency .................................................. 15
      2.5.3 Response time ............................................ 16
      2.5.4 Real time constraints .................................. 16
      2.5.5 Start up times .......................................... 16
      2.5.6 Time to performing ...................................... 16
      2.5.7 Monitor and manageability overhead ................... 16

3 Related work 18
   3.1 Characterizing workloads .................................... 18
      3.1.1 Metrics .................................................. 18
      3.1.2 Usage of hardware ...................................... 18
      3.1.3 This research ........................................... 19
   3.2 Measuring Java performance .................................. 19
      3.2.1 Aggregation ............................................... 20

4 Analyzing workloads 21
   4.1 Methodology .................................................. 21
      4.1.1 Profiling with JRockit ................................ 22
      4.1.2 Hardware profiling ...................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3</td>
<td>Byte code analysis</td>
<td>24</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Overhead</td>
<td>24</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Perturbation</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Analysis work flow</td>
<td>25</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Creating a hypothesis</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Time spent</td>
<td>26</td>
</tr>
<tr>
<td>4.2.3</td>
<td>JVM code</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4</td>
<td>JIT compiled code</td>
<td>27</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Operating system code</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Metrics</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Virtual Machine level</td>
<td>29</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Operating system level</td>
<td>31</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Hardware level</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Measuring performance</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Methodology</td>
<td>34</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Minimizing non-determinism</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Post processing results</td>
<td>35</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Aggregation</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Analyzing results</td>
<td>40</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Comparing results</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Results</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Analysis methodology</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>Index software</td>
<td>42</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Infrastructure</td>
<td>42</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Model</td>
<td>43</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Interfaces</td>
<td>46</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Mixer table</td>
<td>46</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Index Visualizer</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Proposed index configurations</td>
<td>47</td>
</tr>
<tr>
<td>6.3.1</td>
<td>General index</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2</td>
<td>JRockit index</td>
<td>49</td>
</tr>
<tr>
<td>6.4</td>
<td>Analysis of an indexation</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and future work</td>
<td>54</td>
</tr>
<tr>
<td>7.1</td>
<td>Theoretical work</td>
<td>54</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Methodology</td>
<td>54</td>
</tr>
<tr>
<td>7.2</td>
<td>Measuring performance</td>
<td>55</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Profiling</td>
<td>55</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Data gathering</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>Software</td>
<td>55</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Infrastructure</td>
<td>56</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Mixer table</td>
<td>56</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Index Visualizer</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Appendices</td>
<td>59</td>
</tr>
<tr>
<td>A</td>
<td>Requirements and use cases</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>Nomenclature</td>
<td>4</td>
</tr>
</tbody>
</table>
I would like to thank my supervisor at Oracle, Stefan Särne, for all his help and visionary thoughts, without Stefan this thesis would never have been. I would also like to thank my supervisor at CSC, Stefan Nilsson, for his help on the theoretical work.

Additionally there are far too many engineers at Oracle to fit here that deserve acknowledgment, thanks for your patience and answering all my questions.

Finally I would like to thank Disa Gustafsson for her patience and invaluable support.
Chapter 1

Introduction

1.1 Reason for this thesis

Oracle is one of the major companies developing enterprise infrastructure software. One key component in Oracle’s software stack is the Java Virtual Machine (JVM) [42]: JRockit, which is manufactured in Sweden, Stockholm. A fair amount of software in Oracle’s software catalog is written in Java [16] and is thus executed in a JVM. Having a semi-internal JVM, Oracle is free to optimize and introduce features for these applications, gaining advantages over a “normal” JVM.

Not only is JRockit an internal JVM for Oracle products, it is optimized and targeted for server workloads. Profiled as a high-performing enterprise Java [16] VM, it is vital that the product is not only robust and rock-solid, but also on the cutting edge of performance.

One tool that may help quality assurance along with research is a performance index system. This index gives an answer to the question “How good is this JVM?” or “How well is JRockit doing what it is supposed to do?”.

1.2 Thesis goal

The goal for this thesis was laying the necessary theoretical and practical ground for the performance index system, which together resulted in a proof of concept of said system.

The meaning of index in this context is defined as a single number calculated from a set of entities. For comparison, consider a stock market index, which is calculated from a set of stock prices.

The number in this case is a weighted aggregation of a suite of performance measurements. The measurements are grouped into components which consist of one or more measurements of varying metrics. A component can also have sub-components. How the measurements are aggregated depends on a given profile.

A profile determines the weights, representing the importance, of components and sub metrics. It also contains what components of the total set of components will be used for aggregation. Think of a profile as a configuration for how the index is computed.

This approach enables Oracle to build one profile for the product manager, another profile for a specific software engineer et c.

In addition to the one number, the application delivering this index should be able to show a complete tree-view of each component and the sub metrics of that component.

The performance index can be used in multiple ways. Among others, as ...

... a measurement tool for release progression. This will be a part of quality assurance. After functional testing is complete, it can serve as a performance regression indicator.
... a research and engineering tool for new features. It can be used as a sub-goal of a project. Given a profile, a project goal can include requirements that are based on the index.

... a driver for focusing on where to work. With a complete, market and customer-driven profile the index can point out areas of improvement.

... an analysis tool used to compare products (JVMs) with each other. This is common practice when assessing how one performs in relation to another JVM, e.g an earlier version or a competitor.

1.3 Problems to solve

In order to develop an application that will aggregate performance measurements to one index there are a few problems that need to be studied and answered.

It is important to know what we mean when we talk about performance, hence a solid definition is needed.

When we know what performance is, we need to relate it to a JVM. What parts of JRockit, or any JVM for that matter, affect performance? We also need to come up with a deterministic and statistically solid way to profile and measure the workloads that is used to build our index. If the method is not deterministic enough one cannot compare measurements, since they could vary for each measurement. The same argument applies to why it has to be statistically solid. It is also good to be able to provide additional information about the measurements, as it provides possibilities for further analysis.

The profile of the workloads consists of a wide range of different metrics. The metrics are taken from various levels in the system stack, such as the hardware, operating system (OS), JVM and application itself. When analyzed, the composition of this profile gives hints of which parts of the JVM that affect the workload’s composite performance.

Given this information base and the requirements (see appendix A) from Oracle we can design a model for aggregating measurements of different metrics. This model is a part of the final application model.

The problem of how to build this index, given metrics of different kinds, also needs to be solved. There are several approaches to aggregating measurements to one score. These approaches need to be evaluated. Can existing approaches be used or is there a need for a custom one?

1.4 Document structure

This thesis is structured as follows:

Chapter one gives an introduction to and a general description of the problem and why it should be solved.

Chapter two defines the concepts that are central to this problem and explains the background of the entities involved.

Chapter three presents the methodology and tools used for analysis and characterization of the workloads. This theoretical chapter represents the main part of the work for this thesis and is the backbone that supports the indexation with workloads as components.

Chapter four explains the principles, methods and infrastructure used for running the workloads and collecting their results. It also covers useful post processing methodol-
ogy to aggregate sets of data to enable comparison. Finally it ends in some discussion on how to analyze said post processed results.

**Chapter five** presents the results of this thesis. The results include discussion about the analysis methodology and the associated tools. The software prototypes are presented, as well as the infrastructure, models and general interfaces.

**Chapter six** refers to related work from research in this field.

**Chapter seven** finishes the thesis with a discussion on what conclusions that are to be drawn from this study. Pointers for future derivative work from this thesis is also presented.

**Appendix A** contains the results of the applied methodology which is presented in chapter three. A comprehensive list of case studies on workloads of very different characteristics is presented.

**Appendix B** presents the formal software engineering requirements and use cases for the prototype software.

**Appendix C** lists some of the commonly used notions and abbreviations found in this study. The notions are not central to the material, yet could be useful for the reader to gain a deeper understanding.
Chapter 2

Definitions and background

This chapter defines the concepts that are central to the problem area of this thesis and explains the background of the entities involved.

2.1 Nomenclature

There is a vast amount of notions and key concepts that need to be thoroughly defined and explained in order to discuss performance in the JVM field. Not every bit of information used in this thesis can possibly be defined or explained in this chapter due to its massive scope.

Appended to this thesis is a compiled list of notions which are important to understand in an unambiguous way within the JVM and Java area. The reader might find it useful. A word written in italics can generally be found there. The list can be found in appendix B.

2.2 Java Virtual Machines and performance

As the performance index describes characteristics of a Java Virtual Machine it is useful to give a concise description of this product. We also need to know the general architecture of a typical JVM and how different aspects of it relate to performance.

A Java Virtual Machine (JVM) can be defined as a set of programs and libraries that enable execution of a Java program [16, 42].

It is the engine which translates the Java byte code and executes it as well as managing the Java application. The JVM is the core component of a Java execution and it is what makes it possible for Java to possess a number of strengths as a language. The following items are described as strengths of the Java language:

Productivity. Java is an object oriented language, which enables the developer to re-use code and structures. Java features automatic memory management, built-in thread synchronization and a rich standard library. Executing an application in a virtual machine could also give opportunities for increased monitoring and managing.

Reliability. Java specifies type safe casts, a minimum of the memory system exposed and built-in security manager. Lost or faulty pointers cannot happen with garbage collection which reduces memory leaks and removes crashes due to pointer arithmetics. Faulty code will in most cases result in a clear error message complete with stack trace and source code line.
Portability. Java was designed with the concept “write once, run anywhere”. It is good practice to abstract and separate platform specific code, in Java this comes automatically by design.

Not surprisingly, this does not come for free. A number of trade-offs are made by the JVM to maintain the requirements of Java.

Performance. Code running in a virtual machine suffers from overhead. The JVM must use CPU-time and resources in order to manage the memory, compile and optimize code and ensure that the execution throws the proper exceptions. It does not necessarily mean that Java programs are slower than their C counterparts, it will depend on a number of aspects which are described in this and the following chapter.

Determinism. For the same reasons that a Java execution can suffer performance-wise it may also suffer in ways of determinism. For example, if the virtual machine decides it needs to perform garbage collection or optimize a method the current executing method might need more time to perform its job.

Resources. In order to abstract the running Java application from the underlying hardware and OS the JVM needs additional resources in order to compile, manage and keep vital information. There are examples of Java executions needing up to five times as much memory as a C++ counterpart application in order to meet its performance [25].

These trade-offs are hard to work around. With the technology available at the time of writing, one is likely to experience overhead if the JVM is to be able to meet the requirements of Java. Although the JVM’s task is to do whatever it can to optimize itself and the application to work around performance issues, it is still the case today that performance remains the big challenge of JVMs.

2.2.1 The big challenge

A Java application should in theory be able to exhibit the same performance as a program written in a non-managed language. Even more so if you consider that the JVM can exploit its knowledge of the running environment and has the possibility to adapt and optimize dynamically in run-time.

The performance problem can partly be solved by investing in hardware and making sure the application executes with a minimum of friction against the JVM. This might not be optimal or even possible in some cases.

The problem with indeterminism can be critical for some applications. A rudimentary or even standard implementation of a JVM cannot give real time promises on deadlines. Nor can it meet quality of service agreements regarding responsiveness or latency.

To address the problem of indeterminism, a specification for a real-time version of Java (RTSJ) [32] has been developed by key members of the Java community. However itself suffers from two other problems: The applications need to be written explicitly for it and are no longer “Vanilla Java”. This also introduces a number of issues, such as no longer being able to rely on the JVM for memory management and thus losing one of the benefits of writing an application in Java in the first place. The other problem is that one needs a true real-time OS to run the RTSJ-enabled JVM to be able to guarantee that the real-time constraints are met. This introduces additional costs as well as limitations to server platforms.

Resources in terms of memory or input/output (I/O) is not a huge problem given today’s standards. Memory is cheap and acceptable I/O rates are easy to achieve with
commodity hardware. The resource “problem” is however much more critical when one needs a system or set of applications to scale well.

Since the metrics mentioned in this section can be of importance to software developers it is critical that the trade-offs are kept to a minimum. It is also important that research and development work to improve on this area.

Several studies have identified key performance components of JVMs [17, 22]. The following sections describes the performance-critical JVM components and concepts.

### 2.2.2 Memory management

The Java language features automatic memory management. In fact, it is an absolutely mandatory feature and it is mostly transparent to the developer.

There are several benefits with automatic memory management. A developer does not have to spend hours on end debugging crashes and bugs due to incorrect allocation or freeing. Tedious and error-prone tasks such as allocating space and freeing it are left for the virtual machine.

Secondly, since the virtual machine is in absolute control of all the pointers in the program it is free to move the objects which is being pointed to (or referenced to, in the Java world). This means that fragmentation on the heap can be remedied by moving objects closer to each other on better parts of the heap, thus freeing up the fragmented parts.

The use of fragmentation within this context refers to the placement of allocated blocks on the heap. If there is a large amount of allocated blocks separated with unallocated space of varying sizes too small to be used one usually refer to the structure of the heap as fragmented.

The drawback of automatic memory management is that it in most cases introduces application pauses. Even though studies have shown that in some cases automatic memory management can match or slightly exceed the performance of explicit memory management one usually have to make other trade-offs, such as a larger heap size [25].

**Garbage collection**

The very core of automatic memory management lies in the hands of a process known as garbage collection.

Garbage collection (GC) is the process in which the virtual machine takes care of no longer used blocks of memory. Since it is such a vital part of automatic memory management the subject has undergone extensive research. Garbage collection is still being researched, developed and polished in both industry and academia. Thus, there is vast amount of literature [21, 23, 27] on garbage collection and related subjects.

The basic concept is that when an object on the heap has one or more references pointing to it from something considered live then that object is live too. An object is considered live when it is possible that it could be accessed in the future. When the memory system can deduce that an object is no longer live in that sense, the object is discarded and the space where it resided is free for use.

**A garbage collection example, mark and sweep**

One commonly used garbage collection algorithm is “mark and sweep”.

To find out which objects are live the virtual machine starts from the execution roots, such as the application threads, follows the references and marks each object visited as live. This is known as the mark phase. It can be performed either concurrently with the application threads or by having many worker-threads mark the live objects in parallel while the application threads are suspended.
When the garbage collector knows which objects are not live it can safely dispose of them. This is known as the sweep phase. Like the mark phase, this can also be performed concurrently with the application threads or with many workers at once in parallel while the application is suspended. It is important to note that concurrent sweeping might not be fully pause less, some implementations will suspend the application for typically short periods of time. By doing this it trades some throughput for shorter response times.

### Generational garbage collection

Many collectors may also consider the age of objects in a process known as generational garbage collection. The process is based on the following observations:

- Most objects die young.
- There are fewer references from older objects to newer than vice versa.

This is generally known as the weak generational hypothesis. The JVM allocates the new objects in a special part of the heap and keeps track of which references cross between the other part and the part with the young objects. This way the JVM can avoid the cost of scanning the whole heap and most of the time it can collect garbage among the dead youngsters, which is theoretically cheaper performance wise if the weak generational hypothesis holds true.

Being able to move longer living objects to a space which is not scanned for garbage as often means we save time due to the fact that a scanned live object could be considered “wasted time” if one is only looking for garbage.

### Additional tasks

In addition to freeing up no longer used memory, the collector may also be responsible for taking care of special Reference objects e.g running finalizers associated to collected objects as well as take care of fragmentation. The latter is known as compaction.

Compaction is the process of moving objects on the heap to be able to free up unused space between them. Space which lies between objects but is too small for the allocator to be able to use is in some implementations known as dark matter. If not taken care of, the total amount of dark matter could be significant and can lead to sub-optimal memory usage.

### Garbage collection in JRockit

There are several trade-offs that can be made when performing garbage collection. Specifically, in JRockit the garbage collector can be set to let the application get as much of the CPU as possible to maximize program throughput or the virtual machine can try to minimize the time it has to stop the execution and thereby minimizing response times.

JRockit employs several algorithms, strategies and heuristics for garbage collection. Most collectors are mark-and-sweep based. A collector can be run with generational allocation and collection or without. The mark and sweep phase can be set to parallel stop-the-world or concurrent along the application respectively. The latter means that the garbage collector threads are running concurrently with the application and only stopping the world when it is absolutely critical, opening up for potentially shorter response times.

A special single-spaced concurrent collector that is highly optimized and interruptable is also available. It makes a deterministic approach to garbage collection with regards to pause times and hence it is called deterministic garbage collection. Successful usage of this collector enables the JVM to meet quality of service agreements by never introducing pauses longer than X milliseconds, where X is typically low, e.g 10 ms at todays standards.
Performance impact

Common for all algorithms is that garbage collection reduces the amount of time the application can execute within. Naturally, this leads to performance penalties. However the penalties have to be considered in relation to problems with manual memory management, such as deallocation and dangling pointers.

On the other hand, the effects of compaction such as relocation of objects could have an improvement on code or data locality.

Some implementations, such as generational variants or stop-and-copy, are also particularly effective at allocation. Compared to a free list-only implementation the JVM can in many cases skip the overhead from list-operations for simple allocation.

Different strategies and algorithms work better for different kinds of applications. For example an application which allocates many small objects which are quickly discarded makes better use of a generational garbage collector while an application that only allocates large objects with typically longer life expectancies could benefit from another method of collection.

2.2.3 Code generation

Java programs are compiled from source to a format known as byte code.

Byte code is a platform independent stack-based instruction set and can be seen as a compressed version of the original source code suitable for line-by-line interpretation.

To be able to execute the actual Java code a virtual machine can either interpret or generate native code. The latter is a process known as Just In Time (JIT) compilation [1].

Interpreting code is done line-by-line during execution, which has the benefit of faster start up times. However, it doesn’t have the benefit of saving the code for fast revisiting and must emulate the stack-based nature of the byte-code language.

Native execution of code is generally faster and thus preferred if speed is an issue. If and how the code is compiled and optimized has a great impact on the performance and execution profile of a Java program [17].

It should be noted that far from all JVMs generate native code. Some implementations have other needs than speed; simplicity and extremely low resource usage can in some cases be critical. As this study is written with a server JVM in mind, we will however not discuss the performance implications of those aspects.

Just in time compilation

Some virtual machines interpret code and only generates optimized native code for certain hot paths of code while other implementations generate code for all methods and later revisits and optimizes that code.

Regardless, a modern JVM will engage in compilation activities and much like garbage collection it will need system resources to do its job.

This has several performance impacts. If the JVM compiles code the first time it executes a method it will be slower at the start-up of a program.

Secondly, many JVMs engage in run-time profiling to detect code which is frequently executed and thus candidates for aggressive optimizations. Profiling comes with overhead of varying magnitude depending on how it is implemented.

The profiling information is used to determine which code needs to be optimized and how. The next section describes run-time optimization and its performance impacts.
Optimization

Code generated from source code can be optimized with a number of static optimizations which will generally make the code more efficient [4]. There are however optimizations that can be performed with knowledge gathered at run-time. This knowledge is often referred to as “profiling”, which contains information on what kind of optimizations that could be beneficial. Performing such optimizations on often-executed code, so called hot code, is called hotspotting [18]. By performing the optimizations on the hot code only, one will not waste time optimizing code that is barely used.

A virtual machine can gather profiling information needed for optimization in a number of ways. Either by stopping the executing threads at a given interval and sampling the stack trace or by counting the number of times a certain method is invoked. It might also be possible to use specific parts of the system hardware and operating system to gather useful data.

Regardless, when a JVM has deemed a piece of code as a good candidate for optimization, it must use system resources to perform the optimizations and make suitable changes so that the new, optimized, code is executed. It should be noted that the optimization could take place in the background in a separate thread, which would reduce the cost on multi processing systems.

Code generation in JRockit

JRockit is a pure JIT JVM and generates all executed methods to native code immediately. No code is ever interpreted, in fact JRockit lacks a run-time interpreter.

This can lead to longer start up times. However since JRockit is a server JVM and server applications tend to run for a long time this side-effect is not considered to be a too high of a penalty and thus becoming a show-stopper. It should however be considered when evaluating a JVMs performance.

Code is hotspotted via sampling. At a given interval the thread stack traces are inspected and information regarding where the JVM spends its time is stored.

After a threshold is reached the JVM will optimize a method and point out the new code so that it is properly executed.

Some virtual machines can do this while a method is executing, a process known as on stack replacement. JRockit, however, does not at the time of writing.

The implication of this is that the optimized code will not be executed until the next time the method is called.

Performance impacts

Optimization and profiling will lead to nondeterminism with regards to execution time. The first time you run a piece of code it will take X time to perform the work that particular piece of code is supposed to do, typically a longer time than subsequent executions which take X’ time, where \( X > X' \).

The reasons for this are numerous; the JVM in question could have generated native code or set up proper structures needed and the instructions executed will most likely also have generated cache-misses and needs to be fetched from memory [13].

This leads to different execution profiles as the efficiency of the code will improve over time, the actual code executed could differ from the original and reside in a different part of the memory.

A user cannot be sure that a given code-path will reach a desired state in the same way as the previous execution, nor is it guaranteed that it will be using the same amount of resources or time between executions.
Experiencing nondeterminism due to underlying systems is the case of all executions regardless if there is a virtual machine involved. However, more layers of abstraction and sources of nondeterminism, such as hotspotting, are involved when running within a JVM.

2.2.4 Thread management

Java comes with built in support for threads. Much of the performance and program behavior is in the hands of the developer when threads are involved. However, certain JVM-controlled aspects of thread management can have a great effect on performance of threaded applications.

Threads can be seen as nondependent processes that share the same virtual memory. Execution of threads is performed in parallel if there are available CPUs. Like processes, threads have to be scheduled and context-switched in and out from the CPU’s registers if there are more threads than what the hardware can handle at one time.

There are several ways to do this. Some virtual machines, e.g Jikes [18], multiplex threads on special monitoring threads, handling scheduling and signaling on their own. Other implementations use platform-specific thread implementations and generally let the underlying operating system handle it via a standard library.

Scheduling and signaling

If threads had no interaction, no signaling or shared data they would be as useful as running a separate process. The very strength of threads lie in the ease of intra-process communication and sharing of resources.

In order to avoid race conditions and to maintain correct order of execution one must be able to control threads execution given their dependencies on each other.

Examples could include barriers, joins, yields, mutual exclusion and semaphores [43].

In the cases where JVMs schedule and synchronize its own threads, the implementation will have great impact on multi threaded applications. In more modern JVMs however, thread scheduling will be handled by $O(1)$ schedulers\(^1\) of todays operating systems and such metrics are of very little importance.

However there are trade-offs to be made in this area too, even if the OS handles threads. Unsynchronized parallel or semi-parallel access to memory will result in varying and unreliable program behavior. One useful tool for controlling thread access is a concept known as locks. The next section will describe locks and their performance implication for which a JVM is responsible.

Locks

The keyword synchronized is used to ensure exclusive access to a piece of code or an object.

Java uses a concept known as monitors to enable synchronization to be somewhat transparent to a developer. A monitor is a special kind of guard object which ensure that only one thread is executing in an objects synchronized sections.

When one thread enters a synchronized block of code a mutual exclusive lock (mutex) is taken. If another thread wants to enter synchronized code while the lock is taken it will have to wait.

When more than one thread seeks to acquire a lock the object is experiencing lock contention, which halts threads from execution. Unwanted or inefficient contention leads to performance degradation.

\(^1\)The input to schedulers which performance are denoted by an ordo are the amount of tasks to be scheduled. A $O(1)$ scheduler takes a constant amount of time to schedule a task, independent on the total amount of tasks.
Performance implications and implementation details of locks

Much of the performance loss that is related to the use of synchronization is up to the developer [43]. The developer has to decide how much synchronization to use. The granularity of locking is also critical, since acquiring a lock can be more expensive than having your worker threads wait for a lock while in other cases contention on a lock could be the performance killer.

There are however details in the implementation of the actual lock that will affect performance.

First and foremost, the use of locks comes with overhead. Mainly CPU time for creating, destroying, acquiring and releasing locks but also JVM-internal memory allocation and usage for lock-specific structures.

The basic structure of a lock is the one of a thin lock. When a thread has taken a lock the identification (ID) of the thread is noted in the locks book-keeping structure. The book-keeping structure is often kept in the header of an object if the lock is thin. Acquisition of a thin lock is quite inexpensive considering that the JVM already knows what object is to be locked and the book-keeping is minimal. However, locks in Java are dynamic and can be changed to some other type at any time. See the next section for different lock variants.

Like already stated, if another thread wants to acquire a lock that is taken, proper lookups are performed to ensure whether or not it is available. In this case it will be unavailable and the thread cannot continue. If the lock is under low contention a thin lock is sufficient from a performance point of view since little needs to be done about one or two waiting threads.

Lock variants

If the lock is under heavy contention one needs better book-keeping. A lock suitable for this is called a fat lock. When using a fat lock, book-keeping information is stored globally and requires more memory. One reason for this is that fat locks store IDs of all threads wanting to acquire it. This is slower performance-wise but generally more effective under heavy contention.

If locks are acquired for very short periods of time a method known as spinning can be used. When a thread wants to acquire a lock that is already acquired it can spin a few cycles, checking whether the lock is released during this time. If the lock is released, it can continue executing its time slice operating on whatever is under that lock without the overhead of unnecessary context-switching.

Finally, a method known as lazy unlocking can be used for situations where heavy synchronization is implemented but close to zero contention and high thread affinity of locks is experienced. Using lazy unlocking, threads will continue executing without unlocking and thus skipping the overhead from releasing a lock. If the lock is acquired by the same thread over and over this is greatly beneficial. However, if another thread wants to acquire a lock under lazy unlocking it has to suspend the thread that has or had the lock originally to be sure whether it still has the lock or not. This is very expensive compared to when a lock variant capable of handling contention is used.

All of these implementations and techniques have to be used properly and scenarios where the running application would benefit from them has to be detected. The dynamics of proper locking is up to the virtual machine and could have a large impact on performance when running a multi-threaded application.
2.2.5 Nature of the Java language

Java comes with a rich standard library. Certain features of Java together with the code in the library can and in some cases most certainly will give rise to performance problems. This section lists some of the most prominent ones, although there are many more.

Java is not only object oriented, it also features polymorphism. One direct feature of polymorphism is virtual invocation. In fact, it is an opcode on the byte code level.

The dynamics of Java also gives the developer comfort in knowing that code is verified and secure. If faulty code is executed the problem will trigger a trap and a proper exception is thrown [43].

These features also comes with overhead, the following sections goes into detail and explains the performance implications.

Invocation

When a method is called, regardless of whether the JVM generates native code for it or not, it has to be resolved. Due to the fact that Java supports inheritance, methods can be virtually invoked and thus have to be located in the internal structures. This can cause some overhead in large applications.

Depending on what classes have been loaded by the class loader, what the classes implement and if the actual methods are bound to an object, if the method is synchronized or not, calling a method can come with performance penalties.

The magnitude of the penalty varies with application and JVM implementations.

Verification

Java features verification of the code it executes. This is done while loading and translating the byte code. One such verification is done on executed methods. The JVM makes sure that the rules of Java are used and that the method is safe to execute.

A large part of the verification happens during class loading, there are however several verifications that needs to be done during run time.

First the JVM makes sure that the methods actually exists. If it does it makes sure that the rules of scope and access are followed. If anything should be out of order the JVM will throw an exception of proper kind [43].

This can come with overhead if not properly implemented.

Exceptions and managed runtime

The Java language features a control and safety mechanism known as exceptions. Typically exceptions handles some condition that would interrupt or halt normal program flow.

The benefit of exception-handling languages is increased safety and error handling if used properly. It does however come with some overhead in certain cases.

Consider the example of a program that loops through an array, doing some arbitrary operation on the array in question. Non exception-handling languages will leave it up to the user to ensure that “elements” at indices that lies out bound of the array are not accessed.

In an exception-handling language, if the compiler or virtual machine cannot deduce that the code will in fact not reach beyond the arrays bounds, code for checking if an exception should be thrown must be executed for each look-up. This is quite expensive.

Another problem with exceptions is that it introduces new semantics for a program. One piece of code that could be dead-code-eliminated [4] in a non exception-handling language could in some cases be deemed necessary to execute due to the possibility of a condition that would result in a thrown exception.
The performance implications for virtual machines lies within the problem of minimizing these verifications and at the same time upholding the semantics of the application. A JVM implementation can benefit greatly under certain workloads if the proper optimizations can be performed. This is naturally true for code generation as well, and in some cases such as JRockit exception-handling is closely tied to code generation.

2.3 Java as a commercial platform

As mentioned in the previous sections of this chapter, Java comes with a set of distinguished strengths and trade-offs. In some cases, these trade-offs can either be overcome or are irrelevant. In other cases, such as enterprise business software, they might not, e.g. due to heavy application requirements.

The strengths have made Java an industry standard language and is used by enterprise software developers to create, among other software, server applications and middleware. This section describes why and how Java is used in the enterprise sector as well as the primary performance concerns in this field.

2.3.1 Server load characteristics

Server and enterprise software consists of applications that differ widely from one another, yet many of them share common characteristics. Enterprises have different requirements on their software than typical end user clients and the systems in which the applications are run also differ.

One important requirement is performance coupled with reliability. Enterprise software are typically under larger workload and stress yet it is critical to have a low or zero rate of failure. Server software also needs to scale properly in aspects such as amount of threads, heap size and input/output.

It is not uncommon for enterprises to have a high requirement on availability. The software needs to have long up times under high load without maintenance or unexpected downtime. Other server software have high requirements on response times and latency. Typical applications are e-trading, telecommunication or media streaming. Many servers also feature abundant threading, typically multiplexing request and user interaction over several worker threads.

Characteristics of server workloads have been studied thoroughly, this thesis refers to some interesting articles concerning Java server workloads [6, 27, 20, 34]. There are a number of benchmarks built to mimic these characteristics [35].

Not all JVMs are built with server applications in mind, hence not all JVM benchmark suites reflect the demands from this area. Oracle’s JVM is however built for server and enterprise software. Thus we need a wider array of benchmarks and applications to be able to index JRockit’s performance. The next section describes this JVM and its architecture.

2.4 JRockit

Oracle’s JVM implementation is called JRockit [37]. It is a propriety high-performance JVM designed for server loads. At the time of writing, JRockit ships with a complete Java Developers Kit (JDK) on modern x86 and SPARC architectures [39].

JRockit deploys state of the art solutions for automatic memory management (garbage collection), code generation and thread management. As stated in previous sections, these areas are of critical importance for the performance of a Java execution.
Oracle’s JVM also tries to meet the requirements of server and enterprise software. The aim is to deliver performance while maintaining reliability and quality of service agreements in certain aspects.

There are several key factors that make JRockit unique and defines the value of JRockit to the end users. Some of them are closely tied to actual performance, others are closer to what you could call *perceived performance*. Other factors gives added value that usually interferes with execution without impairing performance, and as such is also of a performance nature. One example of this in JRockit is JRockit Mission Control, the tool and interface for managing and supervising JRockit and its running applications.

Many of these metrics are solely measured by specific workloads or niche benchmarks and there is to date no complete bundle of applications or benchmark suite that cover all the performance areas that JRockit delivers within. As stated before, that is one of the reasons this index is under development.

This introduction gives a brief view of what JRockit offers.

### 2.4.1 Throughput

Performance is a primary feature of JRockit, and throughput performance a primary component within that feature. JRockit performs no byte code interpretation whatsoever. All code is generated to the native machine code-set on which the JVM is being executed. The JVM also performs static and dynamic optimization on the generated code.

While this can in some cases give longer start up times, it should be beneficial in the long run when the code is subsequently executed.

Memory allocation is also quite fast, due to an advanced allocation system. In some cases, such as when generational garbage collection is used, allocation can be virtually overhead free. When the application allocates blocks small enough to fit in the *nursery area* of the heap the JVM does not have to perform expensive free list operations.

JRockit also features thread local areas (TLAs) for which a thread can allocate objects on without taking a synchronization lock on the heap. These TLAs are distributed in an advanced manner (e.g. making use of caches), to reduce the overhead of acquiring said areas.

There are many benchmarks that measure throughput in various ways. A good representation of the best and most stable benchmarks should be included in the index.

### 2.4.2 Determinism

As mentioned in section 2.2.2, JRockit comes with a special garbage collector tailored for deterministic pause times. Given a system of certain standard and an application which benefits from short response times the execution will be able to meet *Service Level Agreements* (SLA) or be on par with a certain *Quality of Service* (QoS). SLA and QoS are closely related, many SLAs have QoS thresholds as requirements.

Since determinism and response times are critical to many server workloads, a fair benchmarking of this feature should be included in the performance index of JRockit.

### 2.4.3 Manageability

One of JRockits primary sell points is its manageability tools and Application Programming Interfaces (APIs). Besides implementing the JVM debugging, profiling and tooling interfaces (JVMDI, JMVPI, JMVTI and JMX) [29] JRockit also features a tooling and profiling interface of its own.

This interface is used by several tools and interfaces, namely:
The Management Console. A “console” which connects to a running JRockit and is used to monitor and manage a JVM instance. It presents data about the heap, CPU, garbage collection as well as information from MBeans [28] deployed within the JVM.

JRockit Runtime Analyzer. Described by the documentation [39, 40] as an “on-demand ‘flight recorder’ that produces detailed recordings about the JVM and the application it is running”.

The Memory Leak Detector. A tool which connects to a running JRockit giving detailed information on objects, trends and how they are allocated. Used for diagnosing memory leaks.

As manageability can be of importance to deployers of server applications it is critical that the performance overhead is minimal. As JRockit is designed for manageability with near zero overhead from the ground up it is important to measure the performance implications of having the various tools and APIs used appropriately.

2.5 Performance

Recognizing the needs of server and enterprise software, studies have shown that it is questionable to base the performance of a JVM solely on one or a few suites of benchmarks such as SPECjvm

What is referred to as performance is computer performance, more specifically Java performance. In this context, performance consists of several metrics. A metric is considered a measure of one or several properties in a running application, the JVM or the underlying operating system.

The following sections list the general metrics that are of interest beside raw throughput. First things first however, lets begin by describing throughput thoroughly.

2.5.1 Throughput

A metric of performance defined as an amount of useful work in a given time frame [46]. Best measured as the time to perform a given set of work or the inverse; the amount of work performed in a set amount of time.

2.5.2 Latency

Defined as the delay between the time an event is triggered and the time of when its effects are detected. One says that the effects of the action-to-occur are latent until it happens, thus “latency”. The goal is to minimize the latency and to reduce the occurrence of “extra latency” which could be introduced via garbage collection pauses, inefficient blocking or locking.

An example in the Java world this metric can be a triggered database transaction that needs to allocate memory. If a garbage collection is triggered due to lack of free memory the thread responsible for said transaction will have to wait. Thus the transaction will be delayed and the result, the transaction being complete, is detected later.

1 According to SPEC: The Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers. [35]
2.5.3 Response time

Response time is a metric closely tied to latency. Response time can be defined as the latency in a request-response type of event. The difference is that response time needs an entity to perceive it and often an input to trigger it.

2.5.4 Real time constraints

Real time constraints can be measured in several ways. Either by setting up quality of service agreements and measure how closely a JVM comes to satisfying them or by setting up hard real time constraints.

The latter can result in binary data and the usefulness of that is perhaps questionable unless the constraints consist of several levels.

One example could involve garbage collection pause times. One quality of service agreement can state that no individual pause on an arbitrary 50 ms window can be longer than 30 ms.

Measuring how well a JVM antes up to the agreement in terms of percentiles is a non-binary approach to get measurements.

2.5.5 Start up times

JVMs are known to take a long time to get to a running state. There are several things a JVM must do in order to begin the execution of an application. For example it must load classes and run class-initialization code. Some JVMs generate native code or set up memory structures.

This metric is defined as time from start of JVM to when the application can handle the first request.

The start up time metric is important because this use case is quite common in development as well as recovery and hot deployment.

2.5.6 Time to performing

Time to performing is a metric somewhat tied to start up times. The state of which the workload must reach is defined in much more detail and the work done by the JVM can include other tasks than simply get it to the first piece of running code.

Most JVM’s will gradually optimize methods as the application begins its execution, see section 2.1.2 for details. Run-time optimization leads to less performance in the beginning of an execution. Performance gets better as more methods are optimized and the running application gets less cycles in the beginning due to the optimizations as well.

This metric is defined as the time from starting the JVM to a given place in the execution within some given workload. A multitude of different sized workloads can be included.

One example can be warming up a system to a certain workload. Some Java-based e-trading systems start with a low workload and slowly ramp up the load to a given level over time. The reason for this is to avoid unnecessary pauses when too much load is on the server while the JVM loads classes or optimizes code.

The quicker a JVM can reach a stable running state without compromising some given quality of service the better.

2.5.7 Monitor and manageability overhead

An important use case of server and enterprise workloads is the ability to monitor and manage aspects of a deployment.
There are several ways to get this kind of information and interaction with a JVM. Some of the common ways are the standard interfaces JVMDI, JMVPI and JMVTI [29]. The two former are ancestors of the latter. JRockit complements this by have its own interface which may give more insight and specific details of the application under execution.

Parts of what is available through these interfaces will have overhead effects on the code being executed. Since it is up to the JVM to implement these interfaces, especially so in the explicit custom-built case, it is important that the overhead is minimized.

This is in fact several metrics. One example could be percentage overhead on throughput when dumping stack frames via JMVTI, another could be throughput overhead when running an application with JRA enabled.
Chapter 3

Related work

The work in this thesis can be divided in the following way:

- Characterization of workloads which includes profiling and analysis.
- Benchmarking and measuring workloads.
- Gathering performance results, transforming and aggregating.

The following sections presents related research in these fields.

3.1 Characterizing workloads

There have been many studies on how to characterize Java workloads. Yefim et al. [23] dissects the SPECjvm98 benchmarks from a cache and memory behaviour standpoint. Karlsson et al. [27] has performed another memory-related study on Java-based middleware, their objects of study were SPECjbb2000 and SPECjAppServer2001. Seshadri and Mericas [34] characterize multi threaded Java server applications and focus their efforts on SPECjbb2000 and Volanomark. Their study focuses on thread-related metrics.

All these studies emphasize on the importance of good cache behaviour.

3.1.1 Metrics

Eeckhout et al. [17] analyze how Java programs interact with a virtual machine at the micro architectural level. They lay out some general factors that affects performance in a virtual machine. Their method makes heavy use of performance counters and one final conclusion of this research is that behaviour at the hardware level can vary greatly between virtual machines.

To label a program with one or a few certain static metrics is an approach that is questioned by Dufour et al. [5]. Their research introduce a concept known as dynamic metrics which tells more about the behaviour of an application without problems with robustness.

3.1.2 Usage of hardware

Hauswirth et al. [19] introduces a method for understanding the behaviour of an application on several levels, not only Java, but operating system, other libraries, hardware and virtual machine level. This approach is called vertical profiling by the authors. The actual technique used is performance monitors, both in software and hardware. Their research also discusses the problem of perturbation.
Another approach on analyzing Java program behaviour using hardware monitors is researched by Sweeney et al. [36]. They modify the Jikes RVM [18] to let the thread multiplexers sample events for each thread. As many other studies, this one focuses most on cache performance but on a thread level.

### 3.1.3 This research

In most studies, when a JVM is being used explicitly to profile a workload either a JVMXI agent is used or by developing special code for reading performance counters from hardware. There is also research which base its profiling on byte code analysis and injection.

This study uses the special non-intrusive tools available in JRockit as an alternative to JVMXI or special instrumentation. Hardware analysis is performed by using the tools supplied by the hardware vendors.

### 3.2 Measuring Java performance

Georges, Buytaert and Eckhout have studied the non-determinism of measured performance of execution in virtual machines. Their paper describes existing performance analysis methodologies and points out statistical pitfalls. They conclude that when dealing with small performance differences one can often draw a faulty conclusion when trusting only one method without additional statistical tools. In addition, they present a statistically robust method for both startup and steady-state performance.

Gu, Verbrugge and Gagnon [12] have performed a study on relative factors in performance analysis of JVMs. In addition to discussing the general factors that affect performance of Java executions, they show that single digit micro benchmark improvements on a specific part of a program can give up to 10% variance in performance on a whole-program execution. With this in mind, they conclude that it is important to consider if the workload is substantially affected by cache hit rates.

Blackburn, Cheng and McKinley [21] compares three garbage collectors in the Jalapeno VM [18]. They describe their methodology: pre-runs to get a good optimization profile, one warm up iteration and several measurement iterations. A study on how SPECjbb2000 and SPECjvm98 behaves with different garbage collectors is performed for which the conclusion is that in Jalapeno, both benchmarks benefit from generational garbage collection.

Dofour, Hendren and Verbrugge [5] studies problems with measuring and assessing dynamic properties and scores of Java benchmarks. The issue of perturbation as well as unexpected program behaviour due to technical limitations in the measurement methodology is also discussed. Finally a discussion on how to present and aggregate measurements is presented.

Blackburn and Garner et al. [7] have done extensive research on the DaCapo benchmark suite. Topics such as experimental design, benchmarking methodologies and suitable metrics are discussed. They also present a technique called Principal Components Analysis for transforming high-dimension data to a lower dimension, typically a vast amount of metrics to a handful of components. It is used to analyze benchmark diversity. Their paper also includes useful practices and tips for general Java benchmarking.

Criterias for a good benchmark are presented by Bull and Smith et al. [26]. In their research they show why the criterias matter and how the Java Grande Forum benchmark suite ante up the criterias.

Another paper centered around the Java Grande Forum benchmarks is written by Daly et al. [8]. The research analyses the suite in an approach that aims to be platform independent. The method used involves byte code analysis on byte code compiled by different compilers. The article concludes that byte code analysis is insufficient to give a
detailed profile of program behaviour, yet it is possible to gain some knowledge of certain points of interest.

Gu et al. [44] describes the work done to the IBM JDK to increase performance and identifies some key performance aspects of JVMs. Among others, they name synchronization, allocation, memory management, run-time resolution, class libraries and graphics. They also present a methodology for finding performance bottlenecks.

Related work within the IBM JDK is presented by Dimpsey et al. [20]. Their research presents key areas of the JDK that affect mutator performance. This research focuses mostly on locks, I/O overhead, memory management and JIT compilation.

3.2.1 Aggregation

On the subject of measurements and infrastructure, Alexander et al. [45] have written a paper on the importance of storing and reporting performance metrics in a unified manner. They present a model known as arcflow, used for storing call-graph like data together with other metrics in an efficient way.

On the subject of aggregating measurements to form a score, or index, of some kind there are numerous benchmark suites with this feature. Two well known suites are SPECjvm and Futuremark’s [10] MARK-series. The SPEC suite has design information available and the 2008 version of this suite is free with source code included licensed with the SPEC license. The MARK suites comes with white papers describing how the score is calculated. SPECjvm98 computes scores compared to a reference run, SPECjvm2008 aggregates scores with the geometric mean and the MARK suites compute the geometric mean of sub-components scores.
Chapter 4

Analyzing workloads

To be able to aggregate measurements of workloads in different forms one would need to know how a particular load interacts with a virtual machine in general and especially with JRockit.

The reason for this is that the requirements state that one should be able to weight measurements and sub-measurements differently depending on what is important to the user.

For example one user could be very interested in XML parsing throughput, transaction response times and lock contention. Performing well in those aspects is of utter importance to this user and they should affect the magnitude of the users index the most.

The framework should then be able to find suitable workloads that tests a JVM with those performance metrics in mind.

A final goal for this work is having all the available applications, benchmarks and tools fully analyzed and their characteristics documented. Given the rather short time frame under which this thesis is being written a final completion of that goal is left for future work.

This chapter presents the general methodology for analyzing and documenting characteristics of workloads. A number of workloads are analyzed and presented as case studies, see 7.3.3, showing how characteristics of an application is correlated to application behavior and its affect on the JVM internals.

The theoretical work presented in this chapter is the backbone of the index, the results have been used to implement aggregation depending on characteristics. More than half of the work for this study has been put into this material.

4.1 Methodology

Workload characteristics have been gathered in the following way:

Each workload is measured with four different JVM setups:

“Tuned” - a fully configured run with all best known methods\(^2\).

“Minimal” - a run with minimal tuning, in most cases the JVM is given a heap size and a priority for the garbage collector.

“Out of the box” - (OOTB) no configuration what so ever. The application is fresh out of the box and the user is running it with a plain \texttt{java <app>} call.

“Control” - A fully tuned run, without profiling or logging.

\(^2\)Best Known Methods (BKM) is a term used to refer to the best JVM arguments, OS tuning and other such parameters.
All runs except the control runs were analyzed with the JRockit specific tools: JRockit Runtime Analyzer (JRA [40]) and verbose logs. The use of external tools were performed on the tuned, minimal and OOTB runs separately. See the following section for an introduction to these tools.

These measurements were run using the latest version of JRockit available to the public at the time of writing.

Most measurements took place on the following platforms:

**Windows 2003 SP2** On an Intel Tigerton 4 socket 16 Core machine with 32GB memory.

**Redhat Enterprise Linux Server 5.1** On an Intel Netburst 2 socket 4 Core machine with 8GB memory.

**Redhat Enterprise Linux Server 5.1** On an AMD Barcelona 2 socket 8 Core machine with 8GB memory.

**SUN Solaris 10** On a SUN-Fire T2000 Niagara single socket 6 Core 24 hardware-threads machine with 16GB memory.

Some benchmarks and applications requires special infrastructure, thorough configuration and tuning heavily dependent on the hardware and OS. These were run on similar hardware as mentioned above, although one or two generations older.


The reason for varying hardware and operating system is mainly to get a good picture of how metrics vary between platforms. Such variations are taken into consideration and when possible eliminated.

The following sections describe the tools used and the specific metrics gained from the resulting profile.

### 4.1.1 Profiling with JRockit

As mentioned earlier in 2.4, JRockit comes with its own custom tools and APIs for monitoring and profiling a Java application. As the work in this thesis is being done with JRockit in mind and Oracle as the ultimate customer of resulting products, a good idea would be to use these tools for the profiling work needed in this research.

**Verbose logs**

JRockit has the possibility to be quite verbose about what is going on under the hood. Like many other JVMs, it features a startup argument called `-Xverbose` [38].

When launching JRockit with this argument, one simply passes it as a command line argument in the following way:

```
java -Xverbose:component MyApplication
```

where component can be a either a single entity or a list. The components denote which parts the JVM should be verbose about.

The following components were used in the measurements for this thesis:

- **memory**: Basic garbage collection information.
- **memdbg**: Debug information regarding memory management.
- **opt**: Log entries for optimized methods. Contains what method, when it was optimized and how long time it took.
Example 4.1. An example of a verbose log entry

**gcpause**: Information regarding the explicit pauses that takes place during various garbage collections.

**compaction**: Entries with information on compaction.

The logging is output to the *standard output* if not specified with `-Xverboselog`. An entry to the log can look like example 4.1.

The entry in this case makes note of a garbage collection which took place 57.198 seconds after the JVM was started. This particular collection was a nursery collection and was done as a stop-the-world collection with several parallel workers. The collection took 21.743 milliseconds to complete.

As the reader may have noticed, one single entry may contain an overwhelming amount of information. A log can be interesting to analyze by ocular examination if you are looking for something particular. For larger trends or quantities the JRockit Performance Team makes use of custom built verbose-log analysis tools.

**JRockit Runtime Analyzer**

As mentioned in 2.4, JRockit has a built in monitoring and manageability interface which, among other things, is used for recording of statistics, run-time information and profiling.

JRA is built for non-intrusive profiling, depending on what extra features are turned on during profiling it should introduce little overhead. As can be seen in the appendices, the control run features only slightly better scores on some benchmarks versus a similarly tuned run with JRA.

Even by default, a JRA recording contains a vast amount of information. Some of the most useful highlights this thesis relies upon includes:

**General**: Collection of general information and metrics.

**Methods**: Runtime profile of time spent in methods.

**Garbage collection**: Information regarding garbage collections.

**Heap**: Statistics for heap size and composition.

**Objects**: List of objects and statistics about them from the execution.

**Optimizations**: List of runtime optimizations done to the executing code.

### 4.1.2 Hardware profiling

The information gathered from profiling with JVM-internal tells us about application performance in an environment. To some extent we can gain some knowledge of OS specific metrics, such as total system context switches or page faults.

If one wishes to be able to watch from a total system perspective, one need to resort to external tools and libraries. One interesting metric is how many processor *cycles* are spent in the actual JVM native code in comparison to the JITed code.
**VTune**

One such tool is the VTune Performance Analyzer [11] from Intel. It is a system-wide monitoring tool which uses special hardware features found in Intel processors to sample events about all executing software within a processor or a set of processors.

Since it uses hardware monitors it is non-intrusive and can run without instrumentation. If one wishes to correlate symbols seen in a profile to source code one needs a full set of debug symbols for the compiled executable.

VTune is also able to profile Java code running in a JVM, using a special JVMPI/JVMTI bridge. This is however not without some overhead, as is the case with most JVMPI interaction.

**Oprofile and CodeAnalyst**

Another similar tool for AMD processors is CodeAnalyst [2], which is a fronted tool for the Oprofile framework.

Oprofile is a Linux kernel enhancement for execution profiling. It reads hardware performance counters as well as software counters, aggregates the data and turns into usable statistics. According to the authors it executes with low overhead.

**4.1.3 Byte code analysis**

There exists a plethora of byte code analysis tools. Their purpose and depth of analysis varies, but some can relate byte code statistics to performance metrics.

**Static analysis**

Static byte code analysis is the process in which one parses all byte code of a given application and have heuristics go over the op codes found. It does not take into account what code is actually executed or if any optimizations can be done. One example of such a tool is the FindBugs [33] application.

Dowling et al. [41] have found that static byte code analysis rarely correlates to the dynamically executed counterpart. Their case studies included SPECjvm98 and parts of the Java Grande Forum benchmark. Since these benchmarks are considered to be diverse enough to be representative, this research does not use any static approaches to byte code analysis.

**Dynamic analysis**

Dynamic analysis requires the code actually be running. Some research has been done in this area, Dufour et al. [15] built a special JVMPI agent for Jikes RVM to be able to extract actual executed byte codes.

This approach comes with overhead, as with most JVMPI profiling.

This research opted to use some dynamic analysis of execute byte code and investigated several methods. None was found sufficient or modern enough to be used. The JRockit tools covered them with lesser overhead or they were simply out of date, built for older JVMs and JDKs.

**4.1.4 Overhead**

When measuring at run time one is guaranteed to experience overhead. By using instrumented code, features in the JVM, libraries in the OS or by reading hardware counters, you are in fact using cycles and resources that the application under measure could have used.
This is an absolute fact given the technology that is available at the time of writing.

Overhead is measurable in most cases. Sometimes this data is available in the profile. In other cases, typically benchmarks, the overhead is evident when you compare scores from a baseline without measurements.

One must be very careful when profiling and measuring Java applications. Research papers in this field [19] comment on this problem. However there is no general rule of thumb, it very much depends on the tool and application under measure.

The research in this thesis includes running control-runs without measurement and profiling for comparison. Different tools and features have been enabled in turns so that the actual overhead of each feature is visible and can be taken in to effect.

This approach also has the added benefit of characterizing the actual workload in some aspects such as locks. Applications with heavily contended locks tend to experience higher overhead from lock profiling than other non-contention heavy applications.

4.1.5 Perturbation

Another pitfall when profiling using any sort of instrumentation or extra features in a JVM is the issue of perturbation.

For example the profiling code executed within the JVM could change the cache behaviour of the total execution, as discussed by Hauswirth et al. [19].

In fact, without explicit knowledge of the profiling tools used, one cannot be certain that runtime characteristics such as cache hit rates are correct.

This research argues that since the profiling information in JRockit is gathered by a separate thread, using the same mechanisms as the hotspot sampler which will be running during any normal execution, cache perturbation is kept to a minimum.

On a multi-processor system it may also be contained within the cache-hierarchies of the processing unit it is executing within.

4.2 Analysis work flow

This section presents the work flow for a typical workload analysis. This work flow is used as a base for the work done in the appendices, see 7.3.3, where a number of case studies are presented.

4.2.1 Creating a hypothesis

A good scientific approach before applying any kind of research method is to create a hypothesis.

In the case of program behaviour analysis, the hypothesis consists of what one can believe that the program is supposed to do, what resources are used and how they are consumed.

Inspection

The very first approach is trying to get a basic picture by inspection. If the program comes with documentation, it is always a good place to start.

If the program comes provided with source code, one can assess the usage of the JDK library, in particular packages and classes which are known for stressing the JVM.

Some basic questions that should be answerable by simple tools, e.g. documentation, source code and operating system utilities:

- What does the authors state that their program(s) is supposed to do?
- Does the program consist of one JVM-process or several?
- Does the program need several concurrent JVM executions?
- Does it use any external Java or native libraries?
- Is network communication used?
- Does the program(s) open any files?
- What JDK level is required?

Simply executing the workload and getting it up and running answers some of these questions.

Getting more information regarding a workload could also be possible by searching for research papers on the subject, there are many cases where this would yield usable results [6, 7, 23, 26, 27, 34].

**Input / Output**

A program that spends a fair amount of its execution time performing I/O could either spend cycles in the operating system or stall on blocking operations. This results in a distorted behaviour if not taken into consideration.

**JDK level**

A workload could exhibit certain characteristics when written for a certain level of the JDK. New features introduced in later versions of the JDK can affect performance.

Interesting features of JDK 1.5.0 [30] from a performance point of view include the `concurrency` utils, `java.util.concurrent`, iterators, autoboxing and enhanced `Collection` classes.

**Anonymous programs**

In the very worst case, one will try to analyze a perfectly anonymous program. No information other than how to start it is known, thus one will have to rely on tools alone.

**4.2.2 Time spent**

The very first approach to system-wide profiling is to generate several system-wide profiles with a performance analysis tool suitable for the platform under test.

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU_CLK_UNHALTED.CORE</th>
<th>INSTR_RETIRED.ANY</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.exe.jit</td>
<td>698365</td>
<td>561447</td>
<td>85.61</td>
</tr>
<tr>
<td>jvm.dll</td>
<td>80356</td>
<td>43869</td>
<td>9.85</td>
</tr>
<tr>
<td>ntoskrnl.exe</td>
<td>23977</td>
<td>2016</td>
<td>2.94</td>
</tr>
<tr>
<td>Other64</td>
<td>5982</td>
<td>9769</td>
<td>0.73</td>
</tr>
<tr>
<td>ntdll.dll</td>
<td>3919</td>
<td>609</td>
<td>0.48</td>
</tr>
</tbody>
</table>

*Table 4.2. An example of basic code module breakdown in VTune.*

With these profiles one can gain knowledge of where (in what code) the CPU has spent its cycles. There should be three main modules. The JVM code (`jvm.dll` or `jvm.so`), the JIT compile code (`java.exe.jit`) and the kernel (OS, e.g. `ntdll.dll`) code, as displayed in table 4.2.
If there is a significant amount of cycles spent in other modules one might suspect a non-optimal running condition or heavy usage of Java to native (JNI) calls.

Depending on the distribution of cycles spent within the three main modules one already has some knowledge to what determines a workload’s performance.

### 4.2.3 JVM code

The JVM code module contains symbols from the actual compiled JVM code, e.g. the code generation and garbage collection systems.

These symbols generally belong to a certain .c file, e.g. `<system>/<submodule>/.../file.c`. Hence the JVM source code is structured in such a way that files belonging to a certain system lie together in the file structure.

This makes it quite easy to associate symbols seen in a VTune profile to source code files, which in turn are associated to a certain system within JRockit.

![Table 4.3. An example of an aggregation of compiled JVM code symbols.](image)

For this research such an association system for compiled JVM code was built, see table 4.3 for an example of its output. Note that OTHER denotes an aggregation of symbols, which if expanded would clutter the table beyond what is desirable for ocular inspection. Each aggregated symbol typically stands for a very small amount of cycles. The system can be set to aggregate groups with an amount of samples below a set limit.

### 4.2.4 JIT compiled code

The JIT compiled code, the Java application code compiled by the JVM, has its own module of interest in a “time spent”-analysis. Apart from the application, certain parts of the JVM might be built in Java. In JRockit, these Java specific parts are application driven and are not affected by heuristics. They can thus be considered be a part of the application. One example of such Java-JVM code is allocation and internal memory structure ordering.

Method profiles for the JIT compiled code can be extracted with the JRockit tools and can for example be found in a JRA recording.

Table 4.4 displays how such a method profile may look.

Such a table is interesting and important, especially for determining the most hot methods, however it can be quite cumbersome for a reader of this thesis to gain a complete
picture of the distribution of the JITed java code. For example one might be interested in how much of the executed JITed java code is the actual application, the JDK or other libraries.

<table>
<thead>
<tr>
<th>Method</th>
<th>Samples</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io.ByteArrayInputStream.read()</td>
<td>508</td>
<td>6.88%</td>
</tr>
<tr>
<td>java.util.HashMap.get(...)</td>
<td>239</td>
<td>3.24%</td>
</tr>
<tr>
<td>jrockit.vm.Allocator.allocSlowCase(...)</td>
<td>196</td>
<td>2.66%</td>
</tr>
<tr>
<td>jrockit.vm.ArrayCopy.copy_checks_done2(...)</td>
<td>193</td>
<td>2.62%</td>
</tr>
<tr>
<td>java.lang.String.toLowerCase(...)</td>
<td>157</td>
<td>2.13%</td>
</tr>
<tr>
<td>...sip.engine.connector.transport.UdpTransportModule$UdpWorker.run()</td>
<td>148</td>
<td>2.01%</td>
</tr>
<tr>
<td>java.lang.String.equals(...)</td>
<td>146</td>
<td>1.98%</td>
</tr>
<tr>
<td>java.util.HashMap.remove(...)</td>
<td>113</td>
<td>1.53%</td>
</tr>
<tr>
<td>...sip.engine.server.CallStateManager.getLockResult(...)</td>
<td>112</td>
<td>1.52%</td>
</tr>
<tr>
<td>...sip.engine.server.SipServletMessageImpl.getSessionImpl()</td>
<td>105</td>
<td>1.42%</td>
</tr>
</tbody>
</table>

Table 4.4. An example of a Method profile gathered from a JRA recording.

Table 4.5 shows an example of an aggregation of the above shown table, where the distribution of the JITed java code should be easier to grasp.

<table>
<thead>
<tr>
<th>Run</th>
<th>JRockit</th>
<th>Workload</th>
<th>JDK</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKM</td>
<td>11.41%</td>
<td>68.31%</td>
<td>15.38%</td>
<td>4.41%</td>
</tr>
<tr>
<td>Minimal</td>
<td>8.84%</td>
<td>55.46%</td>
<td>25.12%</td>
<td>10.58%</td>
</tr>
<tr>
<td>OOTB</td>
<td>6.62%</td>
<td>56.48%</td>
<td>25.57%</td>
<td>12.33%</td>
</tr>
</tbody>
</table>

Table 4.5. Example of an aggregated JITed java code method profile, across differently tuned runs.

4.2.5 Operating system code

The time spent in the different part of the OS varies between applications, virtual machines and operating systems. Typical examples of OS code could be native I/O, thread management or memory management the OS needs to perform due to JVM or application behaviour.

No specific profile of exactly what part of the OS is being executed was gathered for this study. However, if one correlates time spent in the OS by other metrics, e.g. information gathered by inspection, one can in some cases assess what the OS is doing.

In other cases information on memory usage by the JVM and application could explain executed OS code, such as memory paging.

4.3 Metrics

A profile from the tools mentioned in 4.1 contains a huge amount of information, most of the times it can be quite hard to make good use of it. Sometimes this information is quite heavily bound to other metrics and in itself would not say much about anything.

During the characterization work that has undergone within this study several such “measurement-bound” metrics have been aggregated to form a dynamic metric. A dynamic metric as it is used in this thesis tries to mimic the work of Eeckhout et al. [15], however in some cases they are only dynamic to some extent.
For example certain “count metrics” found in a JRA recording say little about an application or the JVM when ripped from its context. Simply bounding it by a time-domain is sufficient for it to say something.

In order to be able to work with a large amount of metrics a good structure is needed. This thesis approaches the problem of having metrics taken from many system levels in a similar approach as in the work of Hauswirth et al. [19].

This section gives some examples of such metrics and how they can be used for analysis. The case studies in the appendices, see 7.3.3, feature some of them and some not mentioned here. Even though the metrics are divided into vertical groups, they can come from tools from any level.

For example typical OS metrics could be found in a JRA recording (context switching, paging, CPU load). Other metrics are harder to gather in the JVM and are reported from a tool of another level (cache hit rates, cycles per instruction).

### 4.3.1 Virtual Machine level

This section contains workload-controlled metrics tied to the inner workings of the JVM and their implications.

**Live data**

Live data, defined as the set of objects considered live throughout garbage collection, is typically measured in a size unit, i.e. megabytes, and varies over time depending on the workload. Some workloads have a semi-constant amount of live data, while others have a varying amount throughout its various states. Different characteristics may have different implications, depending on JVM.

**Semi-constant** live data: In a JVM which has the amount of live data as a tuning factors for heuristics, having the amount at a somewhat constant level stabilizes the behaviour of for example the garbage collector. The important factor here becomes the live data’s size in relation to the amount of memory the JVM can set aside for the Java world. Too much live data cripples the free parts of the heap, leading to more collections and with that, compaction.

**Varying** live data: For JVMs which tune behaviour depending on this metric, having it vary throughout program state could result in sub-optimal performance for these tuned components. In addition, the above mentioned statement about the live data’s relation to heap size also applies.

**Allocation rate**

Allocation rate is defined as the rate at which a Java workload’s threads can allocate objects on the heap. As the rate increases, the ability to allocate on the heap is stressed via several bottlenecks.

- **Allocation pipeline**: Whenever a thread allocates an object, in the very rudimentary multi-threaded case, each allocation would require a lock on the big heap, preventing writes from any other thread during the time an allocation takes place. Thus, having an efficient semi-lock free allocation schema is needed.

To avoid locking, threads could for example be given their own local area to allocate on. Efficient distribution of said areas is stressed as well as, in the case of free list allocators, the handling of free lists.
Other additions to the pipeline could involve advanced caches of above mentioned structures, stressing the maintenance and distribution of its elements.

- Allocating vs Collecting: As allocation rates increases, partly due to an efficient pipeline as well as systems having more threads and memory, the ability to maintain a sane heap structure is stressed.

More allocation could mean more collecting, the two metrics meet at a theoretical limit known as Ideal Allocation Rate, where more allocations in the same time frame would decrease application throughput or even be impossible due to stopping the world for collection.

Additionally, depending on how lock-reduced allocation distributes areas on the heap, higher rates could mean a higher need for compaction, additionally stressing the collector.

Dark matter

Dark matter is a size metric, indicating how much space that is unused and wasted due to being distributed in too small chunks between live objects. Any free chunk not small enough for the required amount that is to be allocated may be part of the dark matter.

A large amount of dark matter stresses the compacting schematics of the garbage collector, as well as leading to less usable space on the heap. Thus, leading to more garbage collection. Dark matter could also lead to sub-optimal cache behaviour, having the CPU fetch cache lines with holes of unusable data in them.

Large objects ratio

The ratio of allocated objects of two kinds: Objects considered large, for some definition of large, and all small objects. The reason it can be an interesting metric has to do with how free list allocators and the reduced-locking scheme works.

The areas where threads are allocating objects are of a certain size, typically not too large to avoid having no heap left to operate on. Said areas are often non-empty and even if they were, objects of sizes larger than these local areas have to be allocated directly on the common heap. Thus, the need to take the big heap lock arises.

Generally, in JVMs which operate with free lists and thread local areas, one would want this ratio to favor the small objects. Hence one would want to tune what the JVM considers a large object to make sure the big heap lock is taken seldom enough while maintaining the usefulness of the thread local areas.

A workload which would yield high ratios either stress the local area sizing heuristics or the allocation pipeline as a whole, with all the side effects that comes with that. See other related metrics above.

Locking information

Locking information, defined as counts of how often an object has been locked on, what kind of lock it ended up being and also native locking of internal JVM structures. The information is a collection of metrics indicating how a JVM is handling synchronized workloads.

These metrics indicates how a workload stresses the JVMs capability in the below listed locking mechanisms.

Convert locks: As described in chapter 2, certain lock types handles contention better. If a workload exhibits high contention it would stress the JVMs capability to convert
said locks to the correct type as well as converting it back to the original type if the locking behaviour is to change.

**Spin** on locks: If a thread is likely to hold a lock for a very short time, it could be beneficial to wait for the lock to be released while still being executed, or spinning on the lock. Thus saving the unnecessary overhead of context switching. A workload which exhibits very short and frequent locking times would be stressing the JVMs spin lock implementation.

**Lazy** lock: Certain locking behaviour favors lazy unlocking, a workload exhibiting this behaviour could be indicative of stressing a possible lazy unlocking implementation in the JVM.

Since locking information is such a huge amount of data, consisting of counts, types, timestamps and more, it is best shown as a large table. Generally locking information could vary greatly from workload to workload, therefore the discussion is hard to generalize and is best to further specialize in from situation to situation.

### 4.3.2 Operating system level

This level contain metrics that depend on workload characteristics which are out of control for a typical JVM, yet impact performance or behaviour.

**Context switches**

A multi-tasked operating system is in charge of switching processes in and out of the CPU, i.e. changing processing context. The process of doing so involves saving the register states and program counter address, thus taking time from the executing process.

A process with many threads, for example a Java application running in a JVM using native threads, has them scheduled by the OS. More threads running leads to shorter time slices to operate within as well as more time being spent copying registers to the stack.

Depending on how one measures this metric, this count could include switches made due to interrupts where the metric would grow when for example performing I/O.

A workload exhibiting high context switching could indicate, when the amount of hardware threads are taken into consideration, that the workload is stressing not only the OS’s capability to schedule and switch but also the JVMs internal thread managing structures.

**Memory paging**

Memory paging is, in this context, defined as loading pages into the *Translation Lookaside Buffer*. When accessing memory addressed in virtual memory, like most programs written for a modern OS does, the OS maps with the help of special CPU caches the virtual address to the actual physical one. A mapping is sliced in to fixed sized pages, consequently called exactly that.

The buffer has a fixed amount of slots, thus not all pages fits in the buffer. If the page is not in the TLB, the hardware has to iterate through the entire *page table* to find the matching mapping. This takes time.

If the workload exhibits memory behaviour that has the consequence that the JVM must address new or seldom accessed memory, the above mentioned scenario could occur.

In summary, this metric shows a workloads ability to stress JVM memory placement management as well as the OS’s ability to do the same.
Input / Output

This section contains a collection of metrics for I/O with varying implications. The I/O referred to is typically memory mapped file and network I/O. All metrics are of concern when they reach higher levels, the implications include:

Blocking I/O: Some I/O events include waiting for a task to complete. It can for example involve waiting for a moving mechanical arm or an incoming buffer to fill up. A smart thing to do on multi CPU systems would be to block the waiting process and scheduling other processes, thus stressing the I/O handling code.

Interrupts caused by I/O: Whenever an interrupt request is issued by any entity involved in an I/O task, the OS schedules the appropriate code for handling the event. This interrupts the running process and launches special code, thus taking time from the executing process. Thus increasing CPU load and stressing I/O handling code.

Typical metrics are amount over time, but could also be correlated metrics like interrupts cause by I/O over time.

A workload exhibiting high values of these kinds of metrics stresses the native libraries used by the JVM in terms of I/O as well as the operating system’s and the hardware’s capabilities in these areas. Going deeper into the system it could also be said to stress the event handling code and drivers, although that is in most generic cases of out reach for the JVM.

4.3.3 Hardware level

This group contains workload-controlled metrics which are out of control of both OS and JVM yet determines performance and characteristics.

CPU usage

CPU usage is a percentage metric which generally should come as a set of measurements over time and physical CPU.

Low CPU usage or unused CPUs could indicate, assuming that the workload is not idling and actually trying to behave like a benchmark, that the JVM cannot utilize the CPUs enough. Enough being a vague term, however either all CPUs are being utilized yet not to the fullest capacity or some CPUs are running under full load while others are idle.

It should be noted that both behaviors could be fully intended. However, the former situation could also indicate that some bottleneck halts the throughput so that the CPU is stalling for input. The latter could indicate that the JVM is not utilizing the CPUs for background tasks properly as the workload clearly is not using enough threads by design.

High CPU usage on the contrary can generally be considered good in a benchmarking context, in other situations where the system would need the CPUs for other tasks and the workload should be loading the CPU modestly the amount of load could indicate for example faulty generated code.

Cache hit rates

Cache hit rates are a collection of metrics containing cache hit rates in correlation to parts of executed code. High cache miss rates in the first level of cache during execution of Java code could indicate misaligned code or data. Similarly, miss rates being high in the native JVM code could indicate faulty compiler settings or badly engineered code or data structures.
Workloads that exhibit higher cache miss rates in selected pieces of code, typically hot code, is of importance. This could be considered stressful for the code generator or a trigger for badly aligned code or structures in the JVM code.

**Cycles spent**

Cycles spent is a metric indicating where, e.g. in what code, the CPU has spent it’s *cycles*. See the time spent sections above in 4.2.2.

In general, the metrics consist of tables of counts correlating to code memory addresses which in turn can be correlated to symbols in the compiled code.

**Cycles per instruction**

Cycles per instruction (CPI) is an aggregated metric, denoting how many cycles was spent executing a specific instruction on average, correlated to the actual code which was executed.

Typically, workloads that exhibit high cycles per instruction (CPI) could indicate JITed code being either generated or optimized in a sub-optimal way. Likewise, for native JVM code it could either be an engineering flaw or compiler artifact.

In contrast, if the code cannot be considered flawed, the workload can be considered challenging for the code generator in these specific spots of high CPI.
Chapter 5

Measuring performance

To be able to index the performance of a JVM it has to be measured. There are several ways of measuring and even more metrics to measure.

This chapter explains the principles, methods and infrastructure used for running the workloads and collecting their results. It also covers useful post processing methodology to aggregate sets of data to enable comparison. Finally it ends in some discussion on how to analyze said post processed results.

5.1 Methodology

The measurements are gathered in different ways, all however at run time:

- By the application alone, measuring it’s own operations and reporting them either subsequently as it is executing or at the end of the execution.
- A combination of the above and external measuring, either by the JVM output or another Java or native execution.
- By an external execution alone.

The report is usually delivered in the end, while in some cases the benchmark continually updates with measurements during the run.

Measuring application performance is referred to as a kind of benchmarking. When benchmarking in the Java world, there are some issues to consider apart from the usual benchmarking practices. This section covers these practices and issues.

5.1.1 Minimizing non-determinism

On todays systems, with multiple cores and operating systems running background tasks there will be sources of nondeterminism. One execution can appear to have been more efficient than a subsequent one if the latter execution was executed while the OS scheduled some background task.

There are even more fine grained issues, such as applications being placed differently in the physical memory, temperature variances in the hardware or extremely small voltage differences.

It is safe to say that with todays systems, one could very well run into nondeterminism if performing a non-trivial task.

Even with the pre-existing conditions in a system, running an application in a managed and dynamic environment such as a virtual machine can add additional nondeterminism to application behaviour. The causes for the nondeterminism are covered in chapter 2.
This section describes some practices that are best utilized to minimize nondeterminism during measurement. They were all used during the characterization work for this thesis.

**Hardware and system**

Ideally the system should be in the exact same state for all runs under measurement. Having the same state includes running the same, minimal amount, of system tasks and preferably rebooting the system in between measurements of varying kinds.

**Steady state**

As previously covered in chapter 2, JIT compiling JVMs generate and optimize the code that is to be executed. Some benchmarks include warm up time to eliminate nondeterminism which stem from this behaviour.

Unless the benchmark is designed to measure warm up, startup or code generation and optimization artifacts, it is generally a good idea to use the monitoring facilities provided by the JVM to ensure that the warm up period is long enough.

In general, one wants to measure warm up and steady state behaviour separately to isolate performance degradations.

### 5.2 Post processing results

The means of assessing relative performance between two or more JVMs include measuring performance, covered in the previous section, followed by extensive comparisons.

When one has measured a workload, the typical data is one or more sets of arithmetic numbers which together form a number of series of some length. In the case where the set is empty or of size one, comparison is trivial. However when data sets are of arbitrary or varying length one needs to post process the data to be able to compare it.

#### 5.2.1 Aggregation

The rest of this section lists and describes some useful mathematical tools for used for processing and aggregating measurements. While they are the absolute basis for analysis, they are of such importance that a thorough explanation of how and why they are used is needed.

**Arithmetic mean**

The arithmetic mean, or “average mean”, is an aggregation of arithmetic points which indicate what the average value of all the data points combined. In statistics, this is referred to as the expected value of an outcome or a sample taken from a stochastic variable.

The formula:

$$x = \frac{1}{n} \sum_{i=0}^{n} x_i$$  \hspace{1cm} (5.1)

When used on measured performance data points, one gets the medium value of all the data points. The arithmetic mean is the solution to the problem: “Consider a vector of values with a known sum, if one substitutes all elements of the vector with one value, what would that value have to be for the sum to remain the same?” [47].

The problem can be expressed as follows: \(\Sigma(2, 3, 4) = \Sigma(A, A, A)\). Then \(A\) is the average. The arithmetic mean is one of the three classic Pythagorean means [47], together
with the geometric and the harmonic mean, which are described further down in this section.

It is an excellent aggregation if the data sets are somewhat close to each other in dimension. It does however only tell where the mid point lie. If one or more of the including elements are far from the others in size, the average is skewed.

It is therefore best used together with some measurement of dispersion. The recommended way for measuring that is standard deviation.

**Standard deviation**

The standard deviation \( s \) is a measure on how much on average each point in a data set differs from the calculated mean. It is important to differentiate between the *population standard deviation* \( \sigma \), displayed in equation 5.2, and the *sampled standard deviation* \( s \), displayed in equation 5.3.

The former is the actual standard deviation for all of the members of the set in relation to the true mean where as the latter is an estimation of the standard deviation, based on a number of samples taken from the population.

\[
\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}} \tag{5.2}
\]

\[
s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}} \tag{5.3}
\]

When one is calculating the sampled standard deviation, the denominator is \( n - 1 \). The reason for this is that the denominator is the number of degrees of freedom in the data set vector, which has \( n - 1 \) elements.

In the case of the population standard deviation, the population has \( n \) elements and the degrees are therefore \( n \).

As a post processing tool, the standard deviation gives an indication on how much the measured data points vary in relation to each other on average.

Georges et al. [14] makes use of the arithmetic mean, the standard deviation and a third mathematical tool called *confidence intervals* in their study of statistically rigorous performance analysis methods.

**Geometric mean**

In the same way the arithmetic mean can be described as a solution to a problem, so can the geometric mean: “Given a vector of elements with a known product, if all the elements of said vector would individually be substituted with one value, what would that value be if the product is to remain the same?” [47].

Expressed as a similar equation: \( \prod(2, 3, 4) = \prod(G, G, G) \), then \( G \) is the geometric mean [47]. The formula is displayed in equation 5.4.

\[
G = \left( \prod_{i=0}^{n} x_i \right)^{\frac{1}{n}} = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n} \tag{5.4}
\]
The geometric mean is best used when aggregating data points of varying magnitude, since a change of $X\%$ to one of the elements affects the geometric mean as much as the same percentage change of another - even if the elements differ vastly in size.

If one is aggregating data points with skewed extremes or lows, the geometric mean dampens the effects of such artifacts [47]. As such, it is the primary choice when aggregating Java performance measurements from different environments.

**Harmonic mean**

The harmonic mean is defined as the inverse of the arithmetical mean of the inverse of a set of quantities. The formula is displayed in equation 5.5.

$$H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \tag{5.5}$$

It is best used for computing average rates of a set of rates. Consider the following example: 'A vehicle is traveling from Y to Y’ at the speed of X. Upon reaching Y’ it turns back to Y at the speed of X’. What is the average speed?'.

At first glance, one might be tempted to use the arithmetic mean. This however would yield the wrong answer, since traveling the same distance at that speed would take a different amount of time if $X \neq X’$. Instead, the harmonic mean must be used. The reason for this is that the quantities involved are *rates*, which (in this case) contains the changing variable in the denominator and thus skews the result.

In the world of Java performance, the harmonic mean is best utilized to compute average rates of transactions and other similar behaviour. Typically used when one is experiencing different growth rates during warm up or stabilization.

**Ad hoc aggregations**

Some workloads may deliver results suitable for specific ad hoc aggregation algorithms. A prime example of such a workload is SPECjbb2005.

For details on this workload, see case study one in appendix 7.3.3. The results are delivered as throughput over time during an initial set of warmup iterations, arriving at steady state iterations. Each iteration scales the amount of work up to stress workload scaling on multi CPU systems.

Due to how the load is scaled for each iteration, a typical 8 warehouse run could deliver a set of data similar to what is shown in table 5.1.

<table>
<thead>
<tr>
<th>Warehouses</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17448</td>
</tr>
<tr>
<td>2</td>
<td>37510</td>
</tr>
<tr>
<td>3</td>
<td>55530</td>
</tr>
<tr>
<td>4</td>
<td>68501</td>
</tr>
<tr>
<td>5</td>
<td>68067</td>
</tr>
<tr>
<td>6</td>
<td>67894</td>
</tr>
<tr>
<td>7</td>
<td>65695</td>
</tr>
<tr>
<td>8</td>
<td>65282</td>
</tr>
</tbody>
</table>

*Table 5.1. An example of typical SPECjbb2005 scores.*

How does one aggregate this set of data? In this particular case, one is interested in how the JVM continues to maintain the peak score after the workload is scaled up to and beyond the peak of hardware threads.

One variation could involve:
• Find the peak iteration
• From the peak to the last iteration, calculate a mean.

Other more advanced variations could involve calculating a regression line of the peak performance or including the warmup throughput in the score to account such behaviour. None the less, this aggregation is indeed ad hoc and most likely only suitable for this workload.

**Post processing example**

Consider the following example:

A performance engineer has been running a benchmark on two different JVMs and have obtained the following data series:

**Set A: Benchmark on JVM 1**

• Series 1: Iterations of a benchmark, with metric operations over time:
  180, 185, 175, 180, 181, length 5
• Series 2: Garbage collection pauses, with metric ms:
  20, 21, 22, 19, 20, ..., length 120

**Set B: Benchmark on JVM 2**

• Series 1:
  170, 175, 170, 230, 160, length 5
• Series 2:
  30, 31, 32, 5, 60, 20, ..., length 80

How does one compare the data? What conclusions can be drawn with certainty? By inspection one can assess that A1 seems to have higher scores, but B1 has larger diffusion and consequently has the highest maximum value.

<table>
<thead>
<tr>
<th>Serie</th>
<th>Arithmetic</th>
<th>Trimmed (^1)</th>
<th>Max</th>
<th>Min</th>
<th>Geometric</th>
<th>Harmonic</th>
<th>Std. deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>180,20</td>
<td><strong>180,33</strong></td>
<td>185,00</td>
<td><strong>175,00</strong></td>
<td>180,17</td>
<td>180,14</td>
<td>3,56</td>
</tr>
<tr>
<td>B1</td>
<td><strong>181,00</strong></td>
<td>171,76</td>
<td><strong>230,00</strong></td>
<td>160,00</td>
<td>179,46</td>
<td>178,08</td>
<td><strong>27,93</strong></td>
</tr>
</tbody>
</table>

Table 5.2. Examples of aggregation.

Table 5.2 shows a number of aggregations on the series respectively. Keeping in mind that the aggregation is to be done by machine automatically, e.g. to be used as a comparison for some part of a performance index, a heuristic to determine what aggregation is suitable is non-trivial.

If one does not know anything about the measurements, i.e. how the entity that was measured works, the aggregations does not tell anything apart from their definition: One serie is more scattered than the other and has lower average.

This clearly demonstrates the need for proper characteristics or workload meta data.

\(^1\)Trimmed: The arithmetic mean with the highest and lowest extremes removed from the set.
Meta data

Depending on how the measured workload works, measurements from it should be processed differently. Looking back at the example in the previous section, if the benchmark’s purpose is to demonstrate how much throughput can be achieved during the most optimal conditions (e.g. cache state, scheduling, contention et cetera) then $\text{max}$ can be a proper aggregation.

Similarly, if the benchmark’s purpose is to show how throughput varies over time to factor in system noise, some kind of mean aggregation can be suitable, depending on the magnitude of the measured data points and their variation.

As is shown in chapter 6, such a system of calculating aggregations depending on meta data was implemented with success.

Metric and context

Equally important as knowing how and why to process data points depending on meta data, metric can be important to consider. Consider the second series in each set, A2 and B2. They contain garbage collection pauses with metric time.

The entire vectors are left out on purpose due to size limitations, however let’s consider table 5.3 an aggregation of the fictional vectors. For the sake of clarity, we can imagine the vectors continuing with values close to the ones shown in the example, section 5.2.1.

<table>
<thead>
<tr>
<th>Serie</th>
<th>Total 2</th>
<th>Arithmetic</th>
<th>Max</th>
<th>Geometric</th>
<th>Harmonic</th>
<th>Std. deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>286.00</td>
<td>23.83</td>
<td>22.00</td>
<td>20.37</td>
<td>20.38</td>
<td>1.06</td>
</tr>
<tr>
<td>B2</td>
<td>251.00</td>
<td>31.38</td>
<td>60.00</td>
<td>26.69</td>
<td>19.50</td>
<td>14.74</td>
</tr>
</tbody>
</table>

Table 5.3. Examples of aggregation of garbage collection pause times.

Which series exhibits preferred behaviour? Again, it depends on the workload, the characteristics of it and the requirements it might have.

If the workload’s purpose is to maximize throughput, shorter total garbage collection time is preferred (B2) - given that both runs had equal time to execute within, which is perfectly reasonable to demand. On the other hand, if the workload has a requirement which state that no pause must be longer than 22.00 ms serie A2 would be preferred.

Similar arguments can be made with respect that predictable pause times which would require a low standard deviation.

Additional requirements

Nothing is said about the heap quality during these runs, longer and more garbage collections does not necessarily correlate to better heap structure with regards to such metrics as fragmentation or data locality.

Another specific requirement on these data sets can be for example be related to pause time frequency or scarcity. Consider the pauses spread out along a time line, with a sliding window of some size X ms (in time) sliding along the time axis. If the total amount of paused time within that window on any point of the axis exceeds some level Y ms the workload will have slipped below a certain quality threshold.

Such discussions would add further dimensions and would require additional measurements and post processing and are left out on purpose in this example. They are however typical examples of service level agreements and quality metrics of these kind of data.

---

2Total time spent paused due to garbage collection.
5.3 Analyzing results

When one has executed a number of benchmarks on a number of JVMs, by hand or via automatic systems, one will arrive at a set of results. This research has utilized a pilot project for storing results in a vast combined database. It was fitting, since the calculation of the index requires a complex structure and a large number of results.

For this discussion however, where one stores the results is irrelevant. Additionally, where the post processing takes place also does not matter. Database, application, third party application source or any other layer will do. For this study, the post processing takes place in the application layer.

5.3.1 Comparing results

Post processing gives the means to compare results, however one cannot wildly compare numbers from different domains as one pleases. There is a distinct order of what can be compared and how.

Levels of comparison

For the purpose of aggregating the comparisons, one have to consider the structure of how one gathers the results.

Each JVM generates results tied to that JVM. A JVM is considered unique to a build or version and these builds can behave differently depending on several variables. Below is the degrees of uniqueness a result is tied to. Note that the order of the levels in this context is arbitrary, in contrast to how levels of aggregation relate to each other.

**JVM**: As mentioned above, a JVM is unique to a version, build and similar systems. One could argue that JDK distinguishes JVMs as well, which is true considering the fact that far from all JVMs come in all the JDK flavors out there today. However for this study, JDK is seen as a part of the settings (see below).

**Platform**: A level made up from several categories. Most prominent are architecture and operating systems. Performance will naturally differ between these variables. However one must also account for things like CPU version and other hardware. To aggregate the latter factors to one entity, the variable physical machine was chosen. This has the added benefit of eliminating differences in hardware due to, for example, changed manufacturing processes or the likes even though a vendor may tag it as equal hardware.

**Workload**: Naturally one must distinguish workloads, as they are by existance non-comparable.

Comparison should only be done within equal values of these levels of comparison. This is the definition of fair comparison. If one wish to gain a wider picture of the performance, one will again have to rely on post processing or aggregation. See chapter 6 for more depth on this subject.

**Accuracy**

After post processing, comparing the aggregated results to each other with fair comparison gives an idea on how each JVM performed under each workload. But if one aggregated result is X% better than another, is it sufficient to say that the better JVM is that much better on that particular test?
The simple answer is no. While one can without a doubt state that the measured performance shows indication of one JVM being better than the other, it is wise to back such statements up with a measurement of diffusion.

Is it still X% better if the measurement of diffusion on the better result set is several magnitudes greater? One could argue that if all the measures have been taken to minimize nondeterminism, perhaps a workload which yields diffuse results is unsuitable for comparison.

None the less, this study treads no further into this discussion and leaves it as future work. All workloads used are carefully chosen with stable non-diffuse results in mind.
Chapter 6

Results

This chapter presents the results of this thesis and mostly discusses the software prototypes developed.

6.1 Analysis methodology

The main results are located in the appendices, see 7.3.3, due to the large amount of data and figures. They are also quite formally structured, so reading them all at once is not recommended. The reader is encouraged to take an in-depth look at them one by one. For the purpose of seeing the method at work, the first two case studies are an excellent start.

6.2 Index software

The second part of the thesis is the implementation part, the actual software that computes the index. The software developed for this thesis consists of the following parts:

- Analysis software.
- Data layer and aggregation model.
- Prototype index utilities.
  - Index mixer board and configuration interface.
  - Index visualizer and report builder.

The following sections describes the resulting software and how it interacts with the surrounding environment at the JRockit lab as well as featuring some example indices.

6.2.1 Infrastructure

Test configuration and execution has since long been in place at the JRockit lab. This research has with some success used the infrastructure for configuring, distributing, launching and executing tests. Minor adjustments and resource management have taken place to ensure new workloads and non-JRockit JVMs could be tested.

To easily manage the benchmark results from a vast platform matrix and multitude of JVMs, one needs an infrastructure to support execution and result management. Fortunately a pilot project for a central storage of test results have undergone in parallel with this study. All software prototypes that were developed for this study have been designed and implemented with this in mind and supports the pilot version of the project.
Consequently all results are stored together with test run information and data about hardware, OS and JVM configurations. The storage platform used was a relational database and most data accessing is done via SQL queries.

Complex data management can be made easier with tools, however the tools and test prototypes used in the pilot project only go so far or were at the time of writing missing or under development. For data that changes often, needs constant restoring and testing this study has used local XML schemas to store configurations and workload meta data.

6.2.2 Model

Once the workloads have been executed on the JVMs, platforms and configurations of choice the pilot projects tools reports the results to the central database.

A part from the problem of getting the whole matrix of results into the database, once that hurdle has been passed it is simply a question of extracting the results and modeling it to fit for aggregation and indexation.

Configuration

To be able to compute the index, one needs to create an index and store its configuration in a model. This configuration could be stored in any medium, however for this study an XML approach was used for convenience and agility.

The following components are stored in the schema:

- JVMs: Displayed name, release and build number along with a tag to mark it as a baseline or target.
- Hosts: Including hostname, architecture, OS and weight.

The hosts collection also includes a calculation descriptor, indicating how to aggregate measurements across different platforms.
- Workloads: The including workloads. Their displayed name and link to stored test-suite.

Workloads contains test-cases, with their displayed name, test unit and specific test. All cases also include a calculation descriptor.
- Performance aspect weights: A list of aspects, for example the ones in section 2.4, with their associated weights in percentage.

Additionally, as the work on the mixer board interface progressed new requirements were introduced. The need to store workload filters and database information arose, as well as similar implementation specific information.

Meta data

As each workload is aggregated and weighted depending on it’s characteristics, there is a need to store such information about information, or meta data, on each workload.

For this study an XML schema was chosen for the same reasons as the configuration, and the model is structured as follows:

- Workload: The top structure, identified with test-suite name.

Each workload includes any number of test-cases, identified with unit and case.

Each test-case includes one or more meta data, consisting of aspect identifier, JVM configuration regular expression and weight. A meta data descriptor determines what aspect a run of a workload belongs to depending on configuration and how the user wants a given test-case calculated.
This approach allows a test-case belonging to one workload to get aggregated into several aspects depending on how a belonging test-run was configured. Since a workload has several performance characteristics of different importance, a meta data descriptor must have an associated weight.

This allows one to run a specific workload with a specific configuration which would make an engineer consider it to belong to, for example a response times aspect. Subsequently one could run it with a configuration to maximize throughput and the model would consider that run to be a part of a throughput aspect.

**Stored computations**

The data layer software takes a configuration, described above, as input. It proceeds by loading the information specified from the database and stores it in a similar structure as described in the former section.

Each stored computation contains the information mentioned in section 6.2.2. In addition to the configuration, it stores the actual data along with the aggregations and comparisons for each level, as described below:

- **JVM:** The basic root structure for anything related to a particular JVM, including all structures below in this list.
- **Aspects:** All workload data associated to a JVM gets copied to its respective performance aspect depending on two things. Calculation and run information. Since meta data determines what aspect a run from a workload belongs to depending on how the run was configured as well as how the user wants it aggregated, a workload may appear in several aspects, with the associated runs spread out accordingly.
- **Workloads:** Workloads are the container structure for test-cases, and workloads themselves are contained within aspects according to the scheme described above.
- **Test-cases:** Each test-case belong to a given workload, as in the configuration scheme.
  - **Calculation:** Each test-case has an associated calculation descriptor indicating if the results should be considered better if they are higher or lower as well as which algorithm that is to be used for aggregation.
- **Hosts:** Within the test-case level, containers for runs belonging to a specific host, i.e. where the run was done, are stored.
- **Runs:** Within the associated host containers, runs from workloads are sorted in each container depending on what host they were executed on. Additionally, runs contain configuration information.
- **Data:** For each run, one or more data point is stored together with id, value and specific tags indicating what kind of data one is looking at.

When the data layer has filled the structure up with all data specified in the configuration, the following occurs to compute the necessary data sets:

1. All data for each run is aggregated to a single entity according to how one aggregates data for the particular test-case, as specified in the configuration. Each run-container stores the aggregated data.
2. For each aspect, workload, test-case and host the best run is picked according to the calculation descriptor. That run represents the test-case it belongs to within its aspect. The score is stored in each host-container.
3. The host-scores are aggregated according to how one aggregates across platforms, specified in the configuration. The host-scores may also be weighted if preferred.

4. For all baseline JVMs and throughout the entire matrix, all test-cases are compared to the other baselines according the calculation descriptors and the comparisons are stored in each test-case.

5. For all workloads, the test-cases are aggregated accordingly and the points are stored in each workload container.

6. For all aspects, all workload-scores are aggregated and weighted depending on the meta-data weights for each particular aspect.

7. For all baseline JVMs, all aspect-scores are aggregated. This forms the comparative index baseline.

8. For all target JVMs, data is aggregated as described in 1-7, except the comparison is done separately for each target against all baselines.

This way, expanding the index of new targets does not impact the scores of any other JVM. Consequently one should choose baselines carefully and try to keep it as static as possible, as adding a new baseline skews the entire index.

Additionally, one could include options to penalize differences between JVMs, with the option to undo said penalty. Consider the use-case where one is comparing JRockits only: a negative difference in this index setup means that one potentially has a regression in a newer build. If the newer build features an unknown regression, the engineer using the index-tool most likely do not want this regression to be hidden behind a feature which increases performance on some other workload.

**Weighting**

The index cannot be calculated without meta data describing the characteristics of each workload. This meta data is based on the conclusions of the workload analysis in the appendices, see 7.3.3.

For all workloads, each test case meta data is described as shown in the below example. The prototype uses an XML document, but the underlying medium could just as well be a database.

**Workload** : SPECjbb2005

**Case** : Benchmark score

**Configuration 1** : OOTB

**Weight 1A** : Code quality, 30%

**Weight 1B** : XML throughput, 10%

...

**Calculation** : SPECjbb ad hoc, higher is better.

**Weight 1X** : ...

**Configuration 2** : ...

...
Information on how to aggregate and analyze the results are also stored in the workload meta data.

The prototype is built in such a way that weighting can be done in an arbitrary way concerning levels of performance. The user could weight in the system, JVM module or Java level or any combination there of. One feature which was not implemented for the prototype is the concept of “super-aspects” which would essentially be a collection of aspects. This would enable the configuration tool to have far less aspects to weight yet still have the meta data to be as fine granular as possible.

6.2.3 Interfaces

For the purpose of utilizing the model as the prototypes intend, implementing the software can take many different routes. As this software could be implemented quite generically, the choice to separate the model, processing and GUI code was made. This enables one to utilize the different parts of the code in different external projects in an abstract way.

Data mining

The model is utilized by a set of data handling classes known as DataAccessors. The idea is to access data, via database, XML, or some other medium, through the accessors from any Java application.

Data processing

As a part of the data model utility classes, a library for processing and aggregating data sets was developed. The library handles the algorithms described in section 5.2.1.

Following the abstract model mentioned in the previous section, the library can of course be used for any external application wishing to operate on similar data.

6.2.4 Mixer table

Two prototype tools were developed to support index configuration, exploring and reporting. This section describes the configuration tool, or mixer table.

Figure 6.1 shows an example screenshot of this very mixer table.

The prototype extracts the needed configuration details from the database, allowing the user to select targets, baselines and hosts. The user then has the choice of letting the mixer table calculate the minimum set of workloads available on that platform matrix.

From the choice of available workloads, the user can copy workloads to an area referred to as the “Configured Workloads” area where settings such as calculation descriptor and displayed name can be set.

The mixing part can be seen to the right where the user can set aspect priorities, based on what is available in the meta data archive for the selected workloads.

6.2.5 Index Visualizer

The second prototype tool developed was the index visualizer and reporter. It serves as a front-end to load, compute, compare and store index data.

Figure 6.2 shows an example screenshot of the index visualizer.

The visualizer is an interactive graph/tree viewer, utilizing the prefuse [9] library to navigate, zoom, scroll, collapse and tooltip the structure as a tree.

The user has the ability to explore the entire structure, seeing what aspects, workloads, platforms, test-cases or runs are responsible for any number of interest. It also displays meta data such as configurations, weights and much more through tool tipping.
Reports
Apart from the database import, extracting and storing computations in raw XML the index visualizer also features the ability to export the computations to various formats including mark up languages such as GraphML and HTML.

The HTML report delivers a top down view of the index in the following way:

- The actual index, each JVM with its associated score.
- Aspect performance, each aspect showing how each JVM performs within it.
- Platform performance, each platform showing how each JVM performs in the individual workloads.

Each item is delivered a table, with more in-depth information and comparisons for each level.

6.3 Proposed index configurations
As a final result, this section presents two proposed index configurations.

The first part presents a general index, usable to compare JRockit to any JVM product capable of running the workloads. This should include most complete JDKs with at least a version of 1.6.0.

The second part features a JRockit only index that can be used to compare JRockit to older and newer releases. A typical use case for an index of this kind could for example be release progression.
As for available platforms, different JVMs ship on their various platforms and it could present at difficult task to include a multitude of platforms if one wishes the amount of baselines to be fair and rich of differences.

The platform problem may not be as challenging for the JRockit only index, as platform support has been fairly stable across JRockit development historically and the matrix should be of reasonable size even if one should have to exclude one or two platforms.

### 6.3.1 General index

For the general index, the configuration has the following requirements:

- The included platforms should be a minimum set of the available platforms of all including baselines. For this index it is acceptable to make sacrifices and only include, for example, Linux and Windows 64-bit, or any other small amount of platforms which are readily available across the matrix.

- No internal measurements can be allowed, the results included may be external only.

As the index will be used to compare JRockit to the world of JVMs from a JRockit point of view, the index consists of a wide range of workloads covering all aspects of performance. It is however biased with weights on what JRockit is supposed to be specialized in.

The following workloads could be a part of such an index:

- **SPECjbb2005**: The holy grail of JVM benchmarks at the time of writing. Covers a large part of the throughput, compiler and some JDK performance aspects.
SPECjvm2008: A collection of benchmarks that is supposed to enter the SPEC scene as the new research and promotion benchmark for JVMs. Covers the compiler, throughput and memory performance aspects.

Sipstone Proxy 200: A pause time and latency bound SIP benchmark which covers some of those performance aspects. Has to be run at a representative rate of calls as one needs to have usable data from JVMs not suitable as well as letting the load be challenging for JVMs that manage lower rates too well. The results are measured by the applications involved and the metric is the latency of a message cycle. Thus it passes the requirement of only external measurements.

Depending on what JVMs one wants to include, there could be reason to include legacy workloads in the case that a reasonable amount of baselines are considered still important but ship with older JDKs.

Such a list could include:

- SPECjbb2000: The older version of SPECjbb, not requiring a JDK level of 1.5.0.
- SPECjvm98: Also an older version, unfortunately missing the modernizations and multi threaded features. Does however work on older JVMs.
- SciMark [31]: Classic scientific number crunching, now included in the SPECjvm2008 suite. This suite was last updated in 2004 and should work on older JVMs.
- Java Grande Forum Benchmark [8]: Benchmarks written to emulate large applications with, at the time of development for this suite, “large” requirements on memory, CPU and bandwidth.
- DaCapo [7]: A mini suite combining aspects of computing such as number crunching, xml parsing, chart plotting and Java language processing.

A general index like this example should cover the basic JVM performance aspects as well as including some of the specialization as well as critical workloads for JRockit. Typically the pause time or real time workloads could be biased if one is indexing JRockit Real Time. Another approach would be biasing the throughput aspect while still maintaining useful input from the other aspects.

The aspect weighting for this index may be set up as below, for a general overview. This in accordance to what JRockit is supposed to perform well at, however, with the added Startup aspect to balance the index.

- Throughput, 40%
- Response times, 50%
- Startup, 10%

Additional aspects, such as latency measurement or monitoring overhead cannot be measured on non-JRockit JVMs and is as such reserved for the JRockit index.

6.3.2 JRockit index

As for the JRockit index, the following requirements apply:

- All shipping platforms as a minimum common set. Since a newer release may support platforms that previous releases do not and vice versa.
The included baselines should be chosen to show progression as well as be considered to allow a large set of platforms.

Pause time and other internal measurements should be included, as it is one of JRockit’s distinguished features.

As we are comparing apples to apples, there is generally no need for biased aspects in the sense of specialization. Instead, one could bias the aspects based on a business standpoint. If any single workload is important for some outstanding reason, it can be argued that biasing the aspect(s) of performance related to that workload is acceptable.

What aspects should one focus on from a business standpoint? Refering to section 2.4, we already know that the primary aspects of JRockit are throughput, determinism and manageability. The next section(s) describes the aspects and including workloads, chosen from the appendices. Note that one workload could belong to several aspects.

**Throughput**

- SPECjbb2005 is a popular throughput benchmark at the time of writing. A workload that is heavily used for promotion and product launching and is as such a very important workload to perform well under for a competitive JVM.

- SPECjvm2008 could also be considered a throughput benchmark, as the nature of the including “micro” benchmarks consist of performing some operation over and over.

- SPECjAppServer2004 [35], a benchmark measuring application server performance, a so called multi-tier J2EE benchmark. This workload can generally be considered a throughput benchmark. Not of the traditional kind, but it does stress the JVM in a throughput oriented manner.

- JRockit internal engineering micro benchmarks. A large collection of small benchmarks, much like the SPECjvm2008 benchmarks yet smaller to the size. Most of the including benchmarks in this suite are considered throughput benchmarks.

**Determinism**

- SipStone Proxy 200: A workload with extreme requirements of response times in relation to throughput. Since the Proxy benchmark revolves around response times in call cycles, it is an excellent benchmark for measuring pause times and their determinism.

- WebLogic Event Server: Running the algo trading benchmark, this workload focuses on low latency transactions. The server maintains its event-state while processing events with extreme latency requirements. Deterministic pauses are required since unwanted delays breaks the latency requirements.

- SPECjAppServer2004: The benchmark is built around throughput versus response times and includes its own measurements of response times. It is therefor an important workload to include if one wishes to measure deterministic pause times.

**Manageability**

Manageability is an important aspect of JRockit, being able to deliver low-overhead manageability and monitoring services with low overhead. To measure these aspects, some of the above mentioned workloads are run with the manageability features turned on.
• SPECjbb2005 with a connected management console, listening on internal JRockit events and JMX beans.

• WLSS with a tool executing memory leak API calls, stressing the JVMs capability to display information about the allocated objects and the heap state while running.

• Selected SPECjvm2008 benchmarks while recording JRA-files. Different benchmarks are suited for different options. For example heavy-lock benchmarks in combination with lock profiling or transaction benchmarks with latency recording.

Additional aspects
An additional aspect of this index could include non-functional quality metrics:

• WebLogic Server startup times. Measuring the time it takes for a WebLogic server deployment to get to a running state, both newly created deployments and existing ones.

• Additional JRockit internal engineering micro benchmarks, measuring aspects such as optimization time and quality or time of class loading and unloading.

In conclusion, a JRockit index should include the following:

• SPECjbb2005: Same motivation as for the general index, see section 6.3.1 or appendix 7.3.3 for further details. The following test configurations could be included:

  Best Known Methods, configuring the JRockits for maximum throughput and peak performance.

  Manageability. Running SPECjbb2005 with JRA recording, JVMTI agents or attached consoles monitoring beans covers some parts of the manageability performance aspects.

  Determinism. Runs tuned for deterministic garbage collection shows the throughput/determinism trade off. It is included in the response times, real time and throughput aspects.

  OOTB. Naturally included in out of the box performance aspects.

• SPECjvm2008: base and peak runs with start up and steady state separated. Comparing the base, peak and included start up runs separately. Covering most aspects including start up and OOTB.

• JRockit internal engineering micro benchmarks: A subset of the vast amount of internal specialized micro benchmarks could cover a wide area of aspects. Typically one would want stable and interesting benchmarks, covering compiler and JDK performance.

• WebLogic Event Server: Running the algo trading benchmark. Using the workloads external measurements in combination with internal measurements such as pause times covers pause time and real time performance aspects.

• WebLogic Server Startup: As the Weblogic Server is bundled with JRockit, it is of importance that the server not only manages well on it (see SPECjAppServer2004 below), but utilizing this workload serves as a start up time performance aspect.
• SipStone Proxy 200: Including the same external measurements mentioned in section 6.3.1, one also has the possibility to compare internal measurements such as aggregations of pause times and how the pauses ante up to service level agreements.

• SPECjAppServer2004: It is unfortunately difficult to use when comparing to other JVMs as the execution includes binary searching for peak performance, thus using the benchmarks own metric is expensive time-wise, but doable. However internal measurements and quality of service agreements are very comparable if one is only measuring on JRockit.

The weights for a “plain” JRockit index may be set up as below. Based on the general index configuration, adding in specialized JRockit metrics.

• Throughput, 20%
• Response times, 30%
• Real time constraints, 30%
• Monitoring overhead, 10%
• Startup, 10%

Another example, an weighting which is based on typical server application loads, seen below. “Typical server application loads” is based on SPECjbb and SipStone characteristics.

• Throughput, 30%
• Response times, 40%
• Scalability, 10%
• Memory usage, 10%
• XML, 10%

6.4 Analysis of an indexation

How does one analyze the results of an indexation? The user ends up with one single number for each of the including JVM’s. One single number does not say much in terms of relative performance, only in relation to the other indices and the underlying baseline index does the result become meaningful.

Consider the case of an indexation\(^1\) where each JVM has an index of different magnitude. If properly weighted, the conclusion to draw from the index is of course that the JVMs with higher index are performing better. The second conclusion is that the highest scoring JVM is the best of what the index is supposed to represent, on the included platforms.

On the other hand, consider an indexation where all the included JVMs are indexed equally. Does this mean that the including JVMs are equally good on whatever the index is supposed to describe? The answer depends on what level of metrics is important to the user.

\(^1\)Ideally, referring to a final indexation in detail, with explicit actual data would be helpful for the reader. Unfortunately, as stated earlier, this is not possible due to time and infra structure constraints.
Given that the index is properly configured and that the wider picture is the one most important metric to the user, the index will indeed give an indication of either which JVM performs the best or, in the second above listed case, that all JVMs are equally good from a broad point of view.

There is however one major pitfall with aggregating such a vast amount of data to a set of numbers in relation. Suppose that one JVM regressed in one aspect of performance and gained substantially in another. From just looking at one number the end user may not notice. The same applies to the index of equally scored JVMs, the end user may miss the finer differences between the JVMs.

To combat this, the index analysis tool has the capability to explore the full composition and aggregation of the index as well as see how the comparison is made along with the resulting sub-indices.

As an answer to the question “Is one number enough to score a JVM?”, this research believes that yes, it can be enough as long as one puts the number in relation to the index. But if one wants the intricate details or if it is important that no single aspect has regressed, then one number is not enough and the user will have to explore the index using the tool.
Chapter 7

Conclusion and future work

This chapter ends with a discussion on what conclusions that are to be drawn from this study. Pointers for future derivative work from this thesis are also presented.

An important conclusion is that Java performance is indeed multi-faceted. One “benchmark suite” does not necessarily cover all performance aspects of a JVM. Performance can be summarized with a point of view - it is shown that it is possible to choose workloads almost arbitrarily, analyze them and summarize their performance given which of the performance aspects are important and to what degree.

The second major conclusion to be drawn from this project is concerning the risk of compressing a large amount of data to one single number. Two JVMs could be indexed to the same one number yet have vastly different performance in different aspects. It is therefore essential to not only rely on the one number, but to explore the structure and composition that makes up the final index. Hence the reason for the visualization tool, which can be seen in 6.2.5.

Since the work for this thesis has been divided into two main parts, each with distinct subparts, the rest of this chapter is divided to reflect that.

7.1 Theoretical work

The main part of this thesis has been a study focusing on developing an analysis method for characterizing Java workloads from a performance point of view, as well as applying said methodology to a number of workloads which forms a the case studies of this research.

7.1.1 Methodology

As such, the method does indeed characterize Java workloads in a general way, partly limited to the metrics and profiles used in the method. It covers artifacts of uncertainty such as platform differences, JVM tuning, overhead and perturbation.

Additionally it covers all levels of a modern physical system and differentiates metrics from different parts, which is also included in the concluding analysis.

On the other hand it is limited to JRockit tools due to how it is used in this study. While there are somewhat equivalent tools available at the time of writing, there are no guarantees as of the overhead or perturbation, nor that they cover all metrics and profiles that the JRockit tools can produce.

The study on the workloads is naturally biased towards how they behave in a JRockit environment, as they are executed and measured solely on JRockit. This limitation is fully acceptable for the purpose of using the characteristics to index JRockit. However, as a more general mean of characterizing a workload one would certainly need to expand the analysis to cover other JVMs and tools.
Analysis tools

The method of correlating hardware samples to JVM modules is a powerful tool for JRockit engineers and similar techniques has been performed by hand for certain analysis projects at Oracle.

The deep analysis, especially the tools used to correlate hardware sampled profiles to internal JVM mechanics are JRockit dependent as they require the very source code and debugging symbols.

It is quite simple to convert a VTune recording to the comma separated input needed to produce the tables of time spent in JRockit. However, for full production use a VTune "pack and go" importer would be preferred.

7.2 Measuring performance

7.2.1 Profiling

All analysis-data for this study has been gathered by hand on selected machines. More machines and platforms would have been preferred. While the underlying data is useful and representative, a full coverege would be optimal. Unfortunatley this was not possible due to lack of time.

Only to a small degree were the AMD and Sun hardware profilers utilized. Benchmark measurements are also not completely performed on all the different hardware. The limited factors are partly due to availability and system downtime but mostly due to time. Running benchmarks and validating the results has shown to take more time than first expected.

To efficiently characterize all workloads that are available to the JRockit team one would have to expand the automatic infrastructure to include a plethora of extra runs. Including hardware and software profiling, as well as control and OOTB runs.

7.2.2 Data gathering

While most data in the proof of concept has been run by the automatic task distribution system and executed within the test frameworks already existing at Oracle much time has gone into modifying frameworks and systems.

Not only will one have to modify above mentioned entities if one wishes to run JVMs not fitting within what is presumed in the system, but adding new workloads for automatic execution is not trivial.

In a perfect world, the index system should have access to most modern competitors as baselines with every workload executed on them. Additionally, meta data for all workloads should also be existing in the database or be easy to add.

As for the infrastructure and storage, the thesis served as a utilizer of the pilot project. The software prototype proves that the relational structure works and is easy to use.

7.3 Software

The software produced for the implementation part of this thesis provide a fully functional proof of concept prototype, working together with the internal JRockit result database and infrastructure.
7.3.1 Infrastructure

As mentioned earlier, the software utilizing the pilot projects proves the strengths and usefulness of the project as a whole. However, some limitations, mainly tools, led to the use of XML files in the prototyping. These XML files could perhaps be better fitted in the same database as everything else.

The results used as index-data for this thesis is also quite lacking in quantity. Mainly due to that it comes from a multitude of frameworks, some which could be considered legacy software.

As the infrastructure project moves on, including all workloads and their results in the new database this index will become increasingly useful.

One major dependency however, is the meta data required for all tests and their considered usage. Some tool for quickly inserting meta data into the database or XML files would be needed if the workload amount is to be fully covered.

7.3.2 Mixer table

The mixer table serves its purpose as a prototype. The user is able to configure a complete index configuration in a non-limited way.

As with most GUI tools, there are a million and one things one might want to add and change. Perhaps one even want to integrate the mixer table and visualizer into the same application and make both of them more dynamic.

7.3.3 Index Visualizer

The purpose of the visualizer is more debugging and demo related than anything else. The reports are most likely what the final user will be interested in. The DataAccessor could naturally be used to compute indices in any other tool that has the option to include Java.

The visualizer is however useful to zoom in on issues with missing or mis-configured runs, as well as an efficient tree-base structure browser, mainly utilized by an experienced user willing to tweak an index configuration.

Perhaps one wants to include some kind of limited support for the top tier DataAccessor reports in a future analysis tool.

Reports

The reports and exports serve as an external final report without dependency on the JRockit lab and software. As such, the HTML report can be expanded to cover any table found in the structure with little effort.

A really powerful export would be a fully-fletched excel sheet with built in formulas showing how the aggregation takes place, as well as color highlighting and macro support for interactive hierarchy. This feature was left out due to time constraints.
References


The appendices containing workload analyses were not published due to the sensitive nature of the material with regards to JRockit as a commercial product.
Appendix A

Requirements and use cases

This appendix presents the requirements and use cases on the different parts of the final software, one of the products of this thesis.

The performance index system must be able to:

- Aggregate performance measurements to one index.
- Compute an index, using some given profile, of non-JRockit JVMs.
- Compute an index including JRockit-specific metrics, making the index non-comparable.
- Given a ready-made profile, the only initial input must be to execute the application.

The profile generator system must be able to:

- Display available workloads, JVMs and platforms.
- Be able to change how to aggregate measurements of each workload individually.
- Customize names and tags of included entities.
- Change the priority of the index, given aspects and their ties to workload meta data.

This section features two use cases, used as basis for program design.

Release progression indicator

Adam is a performance engineer in a JVM team and is working on investigating the performance of the latest build on the current stabilization project. One of the release criterias is that the product must not have any regressions. There are however some circumstances which would allow a regression in a benchmark, depending on among other things severity and impact.

He has encountered several regressions during the release project. Some have been fixed, some have not. He now wishes to get a quick overview of the performance of this particular JVM in regards to a number of aspects.

Adam loads up the index visualizer with a pre-made release progression index. The visualizer loads the data, aggregates it and finally compares it to a number of pre-computed baselines.

His suspicions are confirmed, some of the regressions impact a number of workloads in some of his aspects of interest. On the other hand he notices that other workloads have gained performance from earlier mentioned fixes. The effect being that the total index has risen just slightly. This enables Adam to slip some regressions in to the release with confidence that the total performance is unharmed.
Product management

Eva is a product manager in a JVM team and is working on getting a clear picture of the startup performance of their product relative their closest competitors. She fires up the index creator tool, quickly marks some of the latest releases as targets and the competitors JVMs as baselines. For this study the only platform of interest is Linux, so she choses a couple of good machines for that OS.

Eva can now get a tree-view over the workloads that have been run on this matrix at the press of a button. She chooses a couple of good startup benchmarks and sets the aspect relevance of “startup” to a high value. The other included aspects are set to really low values.

Finally she saves the configuration and loads it into the index visualizer and report generator which shows her an interactive tree view of the index and its sub-components with the option to export to various document formats.
Appendix B

Nomenclature

Block : (in this context) a block of memory of a given size.

Concurrency : A notion describing a set of entities executing at the same time, possibly interacting with each other.

Cycle : The time period for a machine to process and execute a machine-code instruction from its memory.

Domain : A collection of applications and configuration files used to specify what and how is to be executed in an application server.

Dark matter : Memory space on the heap too small to be used for allocation, typically found in between used blocks of memory.

Finalizer : Special method which is run before the object is disposed in a garbage collection.

Heap : A large memory area allocated to a program, referred to as the heap.

Hot code : Code which is often executed.

JRE : Java Runtime Environment. A JVM and the libraries required for running a Java SE workload.

Live data : Data on the heap which is reachable from other data also considered live.

Lock contention : The event where several threads are trying to acquire the same lock.

Lock granularity : How often in a segment of code a certain synchronization to a lock is done.

Mutex : Mutual exclusion lock.

Online Transaction Processing : A set of systems that manages transactions, typically entering and receiving given sets of information.

Op code : Operation Code, part of a language instruction set that specifies a specific operation.

Quality of Service : A metric of performance together with a requirement, e.g. 99% of all frames processed must be on par with some standard.

Race condition : A condition where two or more executing threads give arise to different program states depending on how they are scheduled.
Reference: An address to a specific entity in the memory address, e.g. an address to an object on the heap.

Register: (in this context) a hardware register in the CPU which stores information in various sizes for very fast access.

Service Level Agreement: An agreement such that a system must uphold a certain quality of service given some circumstances.

Steady state: A state of the JVM in the terms of optimized methods and allocated data.

Translation Lookaside Buffer: Abbreviated TLB, a CPU cache used to store virtual address translation entries. Implemented in hardware for speed, it stores mapping between a virtual memory space and the actual physical.

Virtual invocation: A call to a method using dynamic binding at runtime, e.g. the actual method which is executed depends on what subclass is actually instantiated in a class hierarchy.