DON’T GET CAUGHT IN THE COLD, WARM-UP YOUR JVM:
UNDERSTAND AND ELIMINATE JVM WARM-UP OVERHEAD IN DATA-PARALLEL
SYSTEMS

by

David Lion

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2017 by David Lion
Abstract

Don’t Get Caught in the Cold, Warm-up Your JVM:
Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems

David Lion
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

Many widely used, latency sensitive, data-parallel distributed systems, such as HDFS, Hive, and Spark choose to use the Java Virtual Machine (JVM), despite debate on the overhead of doing so. This thesis analyzes the extent and causes of the JVM performance overhead in the above mentioned systems. Surprisingly, we find that the warm-up overhead is frequently the bottleneck, taking 33% of execution time for a 1GB HDFS read, and an average of 21 seconds for Spark queries.

The findings on JVM warm-up overhead reveal a contradiction between the principle of parallelization, i.e., speeding up long running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks. We solve this problem by designing HotTub, a modified JVM that amortizes the warm-up overhead over the lifetime of a cluster node instead of over a single job.
Acknowledgements

First, I would like to thank my supervisor Professor Ding Yuan for his constant support and guidance. His unrivalled dedication to his students is largely responsible for any of my success or personal growth over the last few years. I would also like to thank him for accepting me as a graduate student and taking a chance on me when I did not deserve it.

Second, I would like to thank the many people involved in the OSDI paper this work is based on. The co-authors of the paper, Adrian Chiu, Hailong Sun, Xin Zhuang, and Nikola Grcevski for their hard work on the paper this thesis is based on. Especially, Adrian Chiu whose incredible dedication was paramount in the success of this work. Yu Luo, Serhei Makarov, Michael Stumm, Jenny Ren, Kirk Rodrigues, Guangji Xu, Yongle Zhang, and Xu Zhao for the useful and thought stimulating discussions. Especially, Yu Luo for setting up and maintaining the server cluster environment used in our experiments. The anonymous reviewers of the paper and its shepherd, Prof. Andrea Arpaci-Dusseau, for their insightful feedback. Working with all of these people has been a pleasure and I owe much to them. I would also like to thank my thesis committee members, Professor Michael Stumm, Professor Tarek Abdelrahman, and Professor Ben Liang for their feedback and insights.

Lastly, I would like to thank my friends and family for always supporting and encouraging me in my pursuits.
Contents

1 Introduction 1

2 Background 5

2.1 Java Virtual Machine 5

2.1.1 Class Loading 5

2.1.2 Interpretation 6

2.2 Systems 7

2.2.1 Hadoop Distributed File System (HDFS) 7

2.2.2 MapReduce and YARN 8

2.2.3 Hive and Tez 8

2.2.4 Spark 9

2.3 Blocked Time Analysis 9

3 Measure Warm-up Overhead 11

4 Analysis of Warm-up Overhead 13

4.1 Methodology 13

4.2 HDFS 15

4.3 Spark versus Hive 18

4.4 Summary of Findings 19

4.5 Industry Practices 19

5 Design of HotTub 22

5.1 Maintain Consistency 24

5.2 Limitations 26

6 Implementation of HotTub 28
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Breakdown of class loading time</td>
<td>17</td>
</tr>
<tr>
<td>7.1</td>
<td>Performance improvements by comparing the average job completion time of an</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>unmodified JVM and HotTub. For Spark and Hive we report the average times of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the queries with the, best, median, and worst speed-up for each data size.</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Comparing cache, TLB, and branch misses between HotTub (H) and an unmodified</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>JVM (U) when running query 11 of BigBench on Spark with 100GB data size.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The numbers are taken from the average of five runs. All page faults are</td>
<td></td>
</tr>
<tr>
<td></td>
<td>minor faults. “Rate diff.” is calculated as (U Rate - H Rate)/(U Rate),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>which shows the improvement of HotTub on the miss rate. Perf cannot report</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the number of L1-icache loads or memory references to know the corresponding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rates.</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Speed-up sensitivity to workload differences using the five fastest BigBench</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>queries on Spark with 100GB data.</td>
<td></td>
</tr>
</tbody>
</table>
## List of Figures

3.1 Intercepting mode changing returns. .......................... 12

4.1 JVM warm-up time in various HDFS workload. “cl” and “int” represent class loading and interpretation time respectively. The x-axis shows the input file size. .................. 15

4.2 The JVM warm-up overhead in HDFS workloads measured as the percentage of overall job completion time. ......................................................... 16

4.3 Breakdown of sequential HDFS read of 1GB file. ............... 16

4.4 Breakdown of the processing of data packets by client and datanode. ......................................................... 17

4.5 JVM overhead on BigBench. Overhead breakdown of BigBench queries across different scale factors. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has larger query time compared with Spark. .......................... 17

4.6 Breakdown of Spark’s execution of query 13. It only shows one executor (there are a total of ten executors, one per host). Each horizontal row represents a thread. The executor uses multiple threads to process this query. Each thread is used to process three tasks from three different stages. 18

5.1 Architecture of HotTub. ........................................... 22

5.2 HotTub’s client algorithm. ........................................ 23

7.1 HotTub successfully eliminates warm-up overhead. Unmodified query runtime shown against a breakdown of a query with reuse. There are 10 queries run on 2 scale factors for BigBench on Spark. Interpreter and class loading overhead are so low they are unnoticeable making up the difference. ................................................. 31

7.2 HotTub iterative runtime improvement. A Sequential 1MB HDFS read performed repeatedly by HotTub. Iteration 0 is the application runtime of a new JVM, while iteration N is the Nth reuse of this JVM. ................................................. 32
Chapter 1

Introduction

In recent years it has become increasingly common to use distributed systems to process large amounts of data. A distributed system is a software application that runs on a cluster of computers where the nodes communicate over a network backbone. A major advantage provided by these systems is their ability to process large amounts of data in an economically feasible manner. They can utilize easily accessible and affordable commodity hardware to scale their computation power by continuously adding nodes to the cluster. This is possible as the computation is parallelized across all nodes in the cluster, avoiding the need for strong and expensive individual pieces of hardware.

A large number of these data-parallel distributed systems are built on the Java Virtual Machine (JVM) [27]. These systems include distributed file systems such as HDFS [30], data analytic platforms such as Hadoop [29], Spark [67], Tez [65, 79], Hive [34, 80], Impala [13, 39], and key-value stores such as HBase [31] and Cassandra [15]. A recent trend is to process latency-sensitive, interactive queries [40, 68, 78] with these systems. For example, interactive query processing is one of the focuses for Spark SQL [10, 67, 68], Hive on Tez [40], and Impala [39].

Numerous improvements have been made to the performance of these systems. These works mostly focused on scheduling [2, 4, 33, 41, 59, 87], shuffling overhead [17, 19, 43, 48, 84], and removing redundant computations [64]. Performance characteristics studies [47, 49, 58, 60] and benchmarks [18, 24, 37, 83] have been used to guide the optimization efforts. Most recently, some studies analyzed the performance implications of the JVM’s garbage collection (GC) on big data systems [26, 51, 52, 62].

However, there lacks an understanding of the JVM’s overall performance implications, other than GC, in latency-sensitive data analytics workloads. Consequently, almost every discussion on the implications of the JVM’s performance results in heated debate [38, 44, 45, 46, 61, 72, 86]. For example, the developers of Hy-
pertable, an in-memory key-value store, use C++ because they believe that the JVM is inherently slow. They also think that Java is acceptable for Hadoop because “the bulk of the work performed is I/O” [38]. In addition, many believe that as long as the system “scales”, i.e., parallelizes long jobs into short ones, the overhead of the JVM is not concerning [72]. It is clear that given its dynamic nature, the JVM’s overhead heavily depends on the characteristics of the application. For example, whether an interpreted method is compiled to machine instructions by the just-in-time (JIT) compiler depends on how frequently it has been invoked. With all these different perspectives, a clear understanding of the JVM’s performance when running these systems is needed.

This research asks a simple question: what is the performance overhead introduced by the JVM in latency-sensitive data-parallel systems? We answer this by presenting a thorough analysis of the JVM’s performance behavior when running systems including HDFS, Hive on Tez, and Spark. We drove our study using representative workloads from recent benchmarks. We also had to carefully instrument the JVM and these applications to understand their performance. Surprisingly, after multiple iterations of instrumentation, we found that JVM warm-up time, i.e., time spent in class loading and interpreting bytecode, is a recurring overhead, which we made the focus of this study. Specifically, we made the following three major findings.

First, JVM warm-up overhead is significant even in I/O intensive workloads. We observed that queries from BigBench [24] spend an average of 21 seconds in warm-up time on Spark. Reading a 1GB file on HDFS from a hard drive spends 33% of its time in warm-up. We consider bytecode interpretation as an overhead because there is a huge performance discrepancy compared with JIT-compiled code (simply referred as compiled code in this thesis) [42, 76]. For instance, we find that CRC checksum computation, which is one of the bottlenecks in HDFS read, is 230x faster when executed by compiled code rather than interpretation.

In addition, the warm-up time does not scale. Instead, it remains nearly constant. For example, the warm-up time in Spark queries remains at 21 seconds regardless of the workload scale factor, thus affecting short running jobs more. The broader implication is the following:

There is a contradiction between the principle of parallelization, i.e., speeding up long running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks.

Finally, the use of complex software stacks aggravates warm-up overhead. A Spark client loads 19,066 classes executing a query, which is 3 times more than Hive despite Spark’s overall latency being shorter. These classes come from a variety of software components needed by Spark. In practice, applications using more classes also use more unique methods, which are initially interpreted. This results in increased interpretation time.

To solve the problem, our key observation is that the homogeneity of parallel data-processing jobs enables significant reuse rate of warm data, i.e., loaded classes and compiled code, when shared across different jobs. We designed HotTub, a modified JVM that eliminates warm-up overhead by reusing JVMs from prior runs. It
has the following advantages. First, it is a drop-in replacement of existing JVMs, abstracting away the JVM reuse, without needing users to modify their applications. In addition, it maintains consistency of an application’s execution, i.e., the behavior is equivalent to the application being executed by an unmodified JVM except for the performance benefit [28]. Finally, it has a simple design that does not require a centralized component, and it selects the “best” JVM that will likely result in the highest re-usage of loaded classes and compiled code.

Evaluating HotTub shows that it can significantly speed-up latency sensitive queries. It reduces Spark’s query latency on 100GB by up to 29 seconds, and speeds up HDFS reads on 1MB data by a factor of 30.08. In addition to warm-up time, the large speed up comes from more efficient use of cache, TLB, and branch predictor with up to 36% of miss rate reductions.

The following contributions of this thesis are heavily based on the work done in the paper appearing in the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) [50].

- It is the first analysis on the JVM’s performance overhead in latency sensitive, data-parallel workloads. We are also the first to identify and quantify the warm-up overhead on such workloads.
- It presents HotTub, the first system that eliminates warm-up overhead while maintaining consistency.
- It also implements a set of improved JVM performance counters that measure the warm-up overhead. In particular, it is the first to provide fine-grained measurement of interpretation time.

The contributions made by the many authors of the original paper were key to the success of this work. Adrian Chiu was responsible for the implementation of the JVM instrumentation for tracking interpretation time as well as helping with the performance study and the evaluation of HotTub. Hailong Sun performed the performance study on Spark using the provided instrumentation. Xin Zhuang aided in the performance study on Hive and the evaluation of HotTub on Hive. Nikola Grcevski was the expert consultant for the paper providing his years of experience on JVM design and implementation to ensure there was no oversight in the study’s methodology or the creation of HotTub due to our inexperience with the JVM. Ding Yuan aided Hailong in the performance study on Spark providing a scheduling simulator and provided guidance on the direction of the work. Adrian, Hailong, and Ding provided the most consistently to the overall discussion of the work.

The source code of HotTub and our JVM instrumentations in OpenJDK’s HotSpot JVM are publicly available [1].

This thesis has the following limitations. First, HotTub is less useful in long running workloads as the warm-up time is amortized by the long job completion time. In addition, HotTub does not completely comply with the Java Virtual Machine Specification [27] with regards to applications that use static variables whose initialization is timing dependent, which is rare and well-known to be a bad programming practice [1, 73].

This thesis is organized as follows. Chapter [2] provides background on the JVM and the data-parallel systems
being examined. Chapter 3 describes the instrumentations to the JVM used to measure its warm-up overhead. Chapter 4 presents the analysis of JVM performance. Chapter 5 and Chapter 6 describe HotTub’s design, implementation, and limitations. We evaluate HotTub in Chapter 7. We survey the related work in Chapter 8 before we conclude and discuss future work in Chapter 9.
Chapter 2

Background

This chapter provides background information on two areas that this thesis focuses on. The first area is the Java Virtual Machine or JVM, which is a runtime environment widely used in various applications and popular in open source data-parallel systems. The second area is the data-parallel applications that are focused on and tested against.

2.1 Java Virtual Machine

The Java Virtual Machine or JVM is a runtime environment designed to manage the execution of programs written to run on top of it. Many different high level programming languages such as Java and Scala, are compiled into Java bytecode, which is the instruction set of the JVM. It is the JVM’s responsibility to load and execute these bytecode instructions, running the user’s application. There are many features provided by the JVM to ease the development of applications and improve their performance. The problem is that many of these features incur overhead slowing execution down initially. The hope was that if an application is long running the overhead is amortized over the JVM’s lifespan and becomes acceptable. However, many applications especially in data-parallel systems break this assumption.

2.1.1 Class Loading

The JVM will dynamically load classes at runtime, first finding the file containing the class and then loading it into the JVM’s memory. A class file contains metadata about the class, such as its fields, their types, and constant values along with the bytecode instructions for its methods. Class files can be aggregated into packages called Java Archive (JAR) files. When executing a bytecode instruction the JVM may see a reference to a class not yet
loaded in the JVM. This reference will have the symbol of the required class. In the JVM a symbol is a special internal string and the symbol of a class is its name. If no class with a matching symbol has already been loaded the JVM must first find a matching class. The thread executing the bytecode instruction with the reference will have a class loader associated with it. A class loader is responsible for providing searching semantics to find the correct class file. The majority of classes are loaded with the default class loader known as the application class loader. The application class loader will search the application classpath, which is a list of local directories set by the user prior to execution. An example of a custom class loader could be to search and read files over a network or on a distributed file system such as HDFS. The JVM will load the first class found by a class loader by parsing the file and populating an internal data structure for the class. Now, any newly encountered unresolved reference with the class’ symbol will see the JVM has already loaded the class and will use it for resolution.

Dynamic class loading has its advantages. One can update libraries used by their application by simply replacing the library’s JAR or class files without needing to re-compile the application. When searching for the class the application will now find the updated files and use the new library assuming any used programming interfaces have remained compatible. This allows users to benefit from library upgrades very easily without any work on their end. Being able to load classes not on the local disk through custom class loaders is another benefit. Applications can avoid the need to distribute many files to all nodes on a cluster in order to run or minimize disk local disk usage.

However, this thesis will show that dynamic class loading comes at a price. Modern applications use many classes and have very large search paths to find the respective files. The loading overhead must be paid for each class in every run of the application. If an application’s runtime is short, but still loads many classes this becomes a large issue. This overhead also generally appears on the critical path. Developers intuitively only reference a class when it is needed and not ahead of time in a thread off the critical path. Since the reasoning required to load a class off of the critical path is confusing the first reference to a class generally ends up on the critical path.

2.1.2 Interpretation

There are three possible ways code is executed in the JVM. The bytecode from the class file can be ran in two ways, either with an interpreter or compiled and ran as machine code with a just-in-time (JIT) compiler. The third way is the user directly running machine code through a native library, but this is only used in very specific cases as it forfeits the advantages of using a higher level programming language. The JVM has background compiler threads that take requests to JIT-compile methods, so that the method no longer has to be interpreted. A request to compile a method is sent when a method is deemed “hot” enough by it having been invoked many times. It is also possible to just compile only part of a method known as on-stack-replacement (OSR), to improve the
performance of a loop body, but not the entire method. The idea is that performance will increase when the heavily used methods are executed natively rather than interpreted. The reason compilation only starts after reaching an invocation threshold is that compiling every seen method would delay the compilation of heavily used methods hurting performance. Also, compiler resources are scarce as they take away resources from the application, so doing it this way is more efficient. The problem is that running in the interpreter is slow and this penalty is paid until JIT-compilation can occur. The original idea was to amortize this cost over the lifespan of the JVM, but if the lifespan is very short this can be a large issue.

2.2 Systems

Many types of systems and applications can suffer from JVM warm-up overhead. This thesis focuses on the newer trend of data-parallel and distributed systems that are built to run on the JVM. Specifically, we study and evaluate three systems: HDFS, Hive on Tez, and Spark. HDFS is used to clearly illustrate the problem in a more controlled environment, while the other systems show the problem in latency-sensitive interactive queries that are becoming increasingly popular.

2.2.1 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System or HDFS is an open source distributed file system based off of the Google File System or GFS. As the name suggests, it acts as a singular file system, but is distributed across a multi-node cluster. It is designed to store large amounts of data, which cannot fit on a single disk, but allow access in a convenient way. The data stored on HDFS is spread across the many disks and nodes included in the system and is also replicated to make it more fault tolerant. Users can perform file system operations on HDFS similar to how they would perform them on a traditional file system on a single disk or partition. HDFS will then figure out the location of the requested data for the user and service the request. This provides a nice abstraction for the user as they do not need to concern themself with the distribution of the data, allowing them to access large amounts of data as if it was all simply stored on one disk.

HDFS is built on the JVM, so naturally it has the potential to suffer from JVM warm-up overhead. However, traditional intuition would suggest since it is a file system and most of its operations require I/O operations touching disks, its work should be bottlenecked by these I/O operations. Originally HDFS was designed to do long and large operations, such as reading huge amounts of data at once, to support this intuition. We decided to study the JVM warm-up overhead on HDFS to see if this intuition was true. It was also a way to see the severity of the problem as many people believe JVM warm-up to be small in general and traditional thinking says I/O operations should always be the bottleneck, so for JVM warm-up overhead to be a problem on HDFS many
assumptions would be brought into question.

2.2.2 MapReduce and YARN

One of the early popular data-parallel systems was MapReduce [20], which was the basis for the Hadoop MapReduce implementation. MapReduce is a programming paradigm that has two basic phases a user must implement. The map phase runs multiple mappers on many nodes simultaneously gathering data from HDFS and filters it outputting key, value pairs that are then stored on HDFS as intermediate files. The MapReduce system then shuffles the data so that pairs with the same key are sent to the same reducer node. In the reduce phase multiple reducers will read the intermediate files taking the pairs with keys that it was assigned by the shuffle. Each reducer will apply the user’s logic to those pairs and write the final output to HDFS. An important aspect of MapReduce to note is that the intermediate data is written to disk and the final reducer output is written to HDFS. Writing to disk and HDFS decreases performance, but improves fault tolerance. Another important aspect is that a reduce phase must follow a map phase, which hinders jobs that chain multiple map and reduce cycles together. Later on the Hadoop environment introduced YARN, which replaced the MapReduce component. YARN is meant to be a general cluster resource management system to allow applications beyond MapReduce to run on Hadoop. MapReduce was then implemented as a YARN application.

Since this thesis is concerned with JVM warm-up overhead the aspect of this system that is important is how it translates to running JVMs. In YARN each mapper and reducer is ran inside a JVM, which is known as a YARN container. Since each node in the system can run multiple mappers or reducers, the number of JVMs running is equal to the number of mappers or reducers currently running on that node.

2.2.3 Hive and Tez

Hive [34] was created to allow users to run SQL-like queries with Hadoop MapReduce. A user will first submit their SQL-like query to Hive, which will create a query plan to be executed on YARN. It is executed on YARN as a directed-acyclic-graph (DAG) of multiple MapReduce jobs. This proved to not be very performant on its own, but through the addition of Tez [79] the performance was improved.

With the development of Tez, Hive is now much more efficient and is less restricted by the MapReduce model. Tez acts as a library for Hive modifying the YARN job Hive outputs. Tez has implemented many optimizations from the database community and no longer strictly follows the MapReduce paradigm. Notably, it breaks the two aspects previously mentioned, as it is no longer required to save intermediate data to disk or for a reduce phase to follow a map phase. The resulting YARN job is then executed on YARN returning the result to the user. The YARN job created by Hive and Tez has two types of components. There is an application master that manages the
execution of the query and then many tasks that execute a piece of work at each stage of the query. These tasks can do things such as read from files on HDFS or joining information from other tasks together. All of these tasks will run in their own YARN container. Tez does try to reuse these YARN containers for different tasks and we will discuss this more in Chapter 4.5.

2.2.4 Spark

Spark will execute these queries in a similar way to Hive on Tez, but there is a fundamental difference in Spark’s architecture. On Spark each node will run a worker, which is a single JVM process. This worker is contacted by a master to spawn an executor on that node for a Spark job. Inside the executor each task for a job will run as a thread. Therefore, for a single job any used nodes will have a single JVM executing its tasks as threads. This is quite different from Hive on Tez, which will have multiple JVMs as YARN containers executing the tasks on each node. However, the problem remains that when the lifespan of the Spark executor, YARN container, or JVM in general is too short the JVM overhead can become a bottleneck.

2.3 Blocked Time Analysis

The goal of the performance study is to measure the contribution of JVM warm-up overhead to overall latency in data-parallel systems, but measuring individual components of these systems is very challenging due to concurrency. In particular we wish to measure the time spent performing class loading and interpretation, but isolating time amount of overall runtime made up of these pieces is difficult. Distributed systems such as Hadoop and Spark run on multiple nodes in a cluster, with each node running at least one process that will contain multiple threads. An simple, but incorrect, method to calculate the contribution of a component, such as class loading, would be to measure the time each thread spends in that component and then sum all measurements. Due to concurrency the summation could overestimate the contribution. For some period of time multiple threads may simultaneously be inside the measured component, but each thread will report that time individually causing the summation to over count as it will include the same period of real time multiple times.

Ousterhout et al. faced the same problem when systematically identifying the performance bottlenecks of distributed computation frameworks and developed blocked time analysis to solve it. They used this analysis to measured many contributions to total runtime, such as the time spent reading from disk, the time spent transferring data over the network, and the time a job is prolonged due to straggler tasks. Although their components of interest differed from ours blocked time analysis was made general to any component allowing us to apply it to JVM warm-up overhead. Since they also used Spark as their target system much of their methodology and solution was directly applicable for our problem as well.
The goal of blocked time analysis is to quantify how much more quickly a job would have completed if its tasks had never been blocked by the measured components. The resulting runtime is a best-case estimation where the improvement to the component is ideal and completely removes all time spent blocked (for example, a disk read happens instantaneously rather than having blocked). For each task the time spent blocked on a component such as a blocking read to disk, is measure by the task. By subtracting an individual task completion time with the time the task spent blocked on a component, the resulting time is the task completion time with a perfect speed up of that component as if it never blocked. However, simply doing this for each task and re-measuring end to end runtime as the start of the first task to the end of the last task is still problematic. Certain tasks may be waiting to be scheduled due to dependencies on previous tasks, but can now be scheduled earlier due to the improved completion time of these previous tasks. Thus, blocked time analysis simulates the scheduling of each task using their new completion time. The final end to end runtime of the simulated job shows the runtime of the job as if no time was spent in the component, but still accounts for concurrency by running the tasks in parallel.

One difference between Ousterhout’s use of block time analysis and our own is that the components of JVM warm-up overhead are not clearly blocking the same way that waiting on a disk read or network transfer are. As our goal is the removal of JVM warm-up overhead we treat time spent performing this overhead as time spent blocked. This makes the result of blocked time analysis the best-case optimization where warm-up overhead is entirely eliminated. In the case of class loading neither the CPU or JVM is blocked as they perform the work of class loading, but the JVM thread running application code is blocked until the class is loaded. In the case of interpretation time technically not even the application code is blocked. However, interpretation is so slow compared to native execution that treating it as blocked time is still a fairly accurate representation.
Chapter 3

Measure Warm-up Overhead

In this chapter we discuss how we instrument the JVM to measure its class loading and bytecode interpretation time with per-thread granularity. Chapter 4 describes how we use these instrumentations to study JVM overhead in data-parallel systems. We use OpenJDK’s HotSpot JVM, version 1.8.0 build 25.66. HotSpot is the primary reference Java Virtual Machine implementation [51].

Measuring per-thread class loading time is relatively straightforward. HotSpot already provides per-JVM class loading counters. We simply change the counter data structures to be thread local.

Measuring bytecode interpretation time is challenging. The JVM may be interpreting bytecode, executing JIT-compiled methods, or executing C/C++ compiled “native” code (referred as native execution). It requires us to instrument every mode change, i.e., transitions between interpreter execution and compiled/native execution. (We are not concerned with the transitions between compiled execution and native execution as our goal is to measure interpretation time.) If a mode change occurs via call, e.g., an interpreted method calls a compiled method or vice versa, it is straightforward to instrument, as it must first go through fixed program points known as adapters in HotSpot. Adapters are necessary because the interpreter uses a different stack layout from compiled or native execution. However, if a mode change occurs via ret, it is extremely difficult to instrument because the callee merely pops its stack frame, regardless of its caller. There is no one program point for ret that allows us to instrument the change back to the caller’s mode.

We instrument mode changing returns by replacing the original return address on the call stack with the address of our instrumented code. Figure 3.1 shows how it works in 5 steps: (1) when a mode changing call is executed, e.g., interpreter method A calls a compiled or native method B, we instrument this transition and also save the return address back to A (0x8f0) into a separate, thread local stack. (2) We replace the original return

---

1The interpreter can also directly jump into compiled code via on-stack-replacement (OSR) [23, 36], a technique that immediately allows a hot loop body to run with compiled code during execution of the method. OSR also has to go through adapters, which we instrument, as the stack layout needs to be changed.
address with the address of our instrumented function \texttt{ret\_handler (0x670)}. (3) When B returns, it first jumps to \texttt{ret\_handler}, which saves the registers that it is going to use. It records the mode change back to A, and pops the original return address (0x8f0) (step (4)). It then restores the saved registers, and in step (5) jumps to the original return address in A. \texttt{ret\_handler} is implemented in 15 assembly instructions.

We have to carefully handle a few edge cases where the return address is used for special purposes. For GC, the JVM needs to walk each Java thread’s stack to find live objects. Each frame’s return address is used to identify its caller. Therefore we cannot leave the address of \texttt{ret\_handler} on the stack; at the start of a GC pause, we restore the original return address. To quickly locate the original return address, we also save the address of the return address on the call stack (0x2a8 in Figure 3.1). Similarly, the JVM uses the return address to propagate exceptions to caller methods. Therefore we restore the original return address upon throwing an exception.

The instrumentation incurs negligible overhead. When both class loading and interpreter counters are enabled on a range of HDFS workloads we used, the overhead is always less than 3.3%.

Note that class loading and interpreter times overlap, but our counter identifies this overlap. Therefore whenever we report JVM warm-up overhead as a single number, it is the sum of class loading and interpreter time subtracted by their overlap. However, we found only a small portion (14.8% in HDFS workload) of them overlap because class loading methods are quickly being JIT-compiled due to their frequent invocations.
Chapter 4

Analysis of Warm-up Overhead

This chapter presents an in-depth analysis on JVM warm-up overhead in data-parallel systems. We first describe the systems used and our analysis method before presenting the analysis result. We also discuss existing industry practices that address the warm-up overhead and their limitations.

4.1 Methodology

We study HDFS, Hive running on Tez and YARN, and Spark SQL running with Spark in standalone mode. HDFS is a distributed file system that is the default file system for many data parallel systems, including Spark and Hive. Both Spark and Hive process user queries by parallelizing them into short tasks, and are designed specifically for interactive queries \[40\] \[68\] \[78\]. They differ in how they parallelize the tasks: each Spark job runs in only one JVM on each host (known as an executor), and utilizes multiple threads where each task runs in single thread (a JVM is a single process). In contrast, Hive on Tez runs each task in a separate JVM process known as YARN container. For each application the version used was the current stable version at the time of experimentation. These versions were Hadoop-2.6.0, Spark-1.6.0, Hive-1.2.1, and Tez-0.7.0.

We benchmark Spark and Hive using BigBench \[24\], which is made to evaluate systems which operate on large scales of data. It provides representative workloads that mimic what occurs in real-world practice, so that the performance of big data systems can be evaluated and compared. It consists of 30 queries ranging over structured, semi-structured, and unstructured data that are modeled after real-world usage. Its queries on structured data are selected from TPC-DS \[82\]. TPC-DS is an industry standard benchmark from the Transaction Processing Performance Council (TPC) influenced by both academic and industry experts. TPC-DS is widely used by SQL-on-Hadoop vendors like Cloudera \[57\], Hortonworks \[35\], Databricks \[22\] \[69\], and IBM \[12\] to drive their optimization efforts.
All experiments are performed on an in-house cluster with 10 servers. Four of them have 2 Xeon E5-2630V3, 16 virtual core, 2.4GHz CPUs with 256GB DDR4 RAM. The others have a single Xeon E5-2630V3 CPU with 128GB DDR4 RAM. Each server has two 7,200 RPM hard drives, is connected via 10Gbps interconnect, and runs Linux 3.16.0. The server components are long running and fully warmed-up for weeks and have serviced thousands of trial runs before measurement runs.

We run the queries on seven scale factors on Spark: 100, 300, 500, 700, 1K, 2K, 3K, and five scale factors on Hive: 100, 300, 500, 700, 1K. Each scale factor corresponds to the size of input data in GB (a scale factor 100 uses 100GB as input size, whereas 3K uses 3TB). For each scale factor, we repeat each query 10 times and take result from the fastest run in order to eliminate trials that might have been perturbed by other background workloads. In addition, we only analyze the 10 queries with the fastest job completion time out of the total 30 queries in BigBench, because of our focus on latency sensitive queries. (These queries are query 1, 9, 11, 12, 13, 14, 15, 17, 22, and 24.) BigBench is designed to be comprehensive, therefore many queries are long, batch processing queries instead of interactive queries. We acknowledge that JVM warm-up overhead will be amortized in long running jobs and are only interested in the short interactive jobs where it could be problematic. When testing all queries to determine which were the faster more interactive ones we found that at least 8 of the 30 queries lead to heavy swapping at large data sizes. This indicates that our cluster lacks physical memory and requires memory to be paged out to disk to continue execution. This could be remedied by either adding more memory to each node in the cluster or adding more nodes to the cluster. However, in its current state the cluster is not even capable of performing these 8 queries in a representative way as in practice the performance penalty of swapping would be unacceptable. It is possible with enough memory these queries may be short running interactive queries, but due to the swapping on our cluster they do not finish in the top 10.

We instrument each thread in the system with the per-thread class loading, interpreter, and GC performance counters to measure the JVM overhead of each parallel task. However, understanding the overall slow down of the entire job is non-trivial as the JVM overhead of multiple tasks can overlap. We borrow the blocked time analysis from Ousterhout et al. [58] to estimate the slow down to the entire job from per-task measurement. It works by first subtracting the time each task spends in the measured event (e.g., class loading) from its total execution time, and then simulates the scheduling of these tasks with the reduced execution time. See Chapter 2.3 for more details on blocked time analysis. We implemented a simple scheduling simulator with 500 LOC in Perl and simulated the original tasks with an accuracy of over 96%.

Limitations. Our measurement of JVM overhead is a conservative underestimate. First, we do not measure the effect of the background threads that are used to JIT-compile bytecode. Similarly, we only measure the stop-the-world GC pause, ignoring background GC activities. This background work will compete with the application for
4.2 HDFS

We implement three different HDFS clients: sequential read, parallel read with 16 threads, and sequential write. We flush the OS buffer cache on all nodes before each measurement to ensure the workload is I/O bound. Note that interpreter time does not include I/O time because I/O is always performed by native libraries.

Figure 4.1 shows the class loading and interpreter time under different work loads. The average class loading times are 1.05, 1.55, and 2.21 seconds for sequential read, parallel read, and sequential write, while their average interpreter times are 0.74, 0.71, and 0.92 seconds. The warm-up time does not change significantly with different data sizes. The reason that HDFS write takes the JVM longer to warm-up is that it exercises a more complicated control path and requires more classes. Parallel read spends less time in the interpreter than sequential read because its parallelism allows the JVM to identify the “hot spot” faster.

Figure 4.2 further shows the significance of warm-up overhead within the entire job. Short running jobs are more significantly affected. When the data size is under 1GB, warm-up overhead accounts for more than 33%, 48%, and 30% of the client’s total execution time in sequential read, parallel read, and sequential write. Sequential write suffers the least from warm-up overhead, despite its higher absolute warm-up time, because it has the longest run time. In contrast, parallel read suffers the most from warm-up overhead because of its short latency. According to a study published by Cloudera, a vast majority of the real-world Hadoop workloads
read and write less than 1GB per-job as they parallelize a big job into smaller ones. The study further shows that for some customers, over 60% of their jobs read less than 1MB from HDFS, whereas 1MB HDFS sequential read spends over 60% of its time in warm-up.

Next we break down class loading and interpreter time using the 1GB sequential read as an example. Figure 4.3 shows the warm-up time in the entire client read. A majority of the class loading and interpreter execution occurs before a client contacts a datanode to start reading.

Further drilling down, Figure 4.4 shows how warm-up time dwarfs the datanode’s file I/O time. When the datanode first receives the read request, it sends a 13 bytes ack to the client, and immediately proceeds to send data packets of 64KB using the sendfile system call. The first sendfile takes noticeably longer than subsequent ones as the data is read from the hard drive. However, the client takes even longer (15ms) to process the ack because it is bottlenecked by warm-up time. By the time the client finishes parsing the ack, the datanode has already sent 11 data packets, thus the I/O time is not even on the critical path. The client takes another 26ms to read the first packet, where it again spends a majority of the time loading classes and interpreting the computation of the CRC checksum. By the time the client finishes processing the first three packets, the datanode has already
CHAPTER 4. ANALYSIS OF WARM-UP OVERHEAD

Figure 4.4: Breakdown of the processing of data packets by client and datanode.

Figure 4.5: JVM overhead on BigBench. Overhead breakdown of BigBench queries across different scale factors. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has larger query time compared with Spark.

sent 109 packets. In fact, the datanode is so fast that the Linux kernel buffer becomes full after the 38th packet, and it had to block for 14ms so that kernel can adaptively increase its buffer size. The client, on the other hand, is trying to catch up the entire time.

Figure 4.4 also shows the performance discrepancy between interpreter and compiled code. Interpreter takes 15ms to compute the CRC checksum of the first packet, whereas compiled code only takes 65µs per-packet.

Break down class loading. The HDFS sequential read takes a total of 1,028 ms to load 2,001 classes. Table 4.1 shows the breakdown of class loading time. Reading the class files from the hard drive only takes 170ms. Because Java loads classes on demand, loading 2,001 classes is broken into many small reads. 276ms are spent searching for classes on the classpath, which is a list of filesystem locations. The JVM specification requires the JVM to

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Read</th>
<th>Search</th>
<th>Define</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>276</td>
<td>411</td>
<td>171</td>
<td></td>
<td>1,028</td>
</tr>
</tbody>
</table>

Table 4.1: Breakdown of class loading time.
load the first class that appears in the classpath in the case of multiple classes with identical names. Therefore it has to search the classpath linearly when loading a class. Another 411ms is spent in define class, where the JVM parses a class from file into an in-memory data structure.

### 4.3 Spark versus Hive

Figure 4.5 shows the JVM overhead on Spark and Hive. Surprisingly, each query spends an average of 21.0 and 12.6 seconds in warm-up time on Spark and Hive respectively. Similar to HDFS, the warm-up time in both systems does not vary significantly when data size changes.

**Software layers aggravate warm-up overhead.** The difference in the warm-up times between Spark and Hive is explained by the difference in number of loaded classes. The Spark client loads an average of 19,066 classes, compared with Hive client’s 5,855. Consequently, Spark client takes 6.3 seconds in class loading whereas the Hive client spends 3.7 seconds. A majority of the classes loaded by Spark client come from 10 third-party libraries, including Hadoop (3,088 classes), Scala (2,328 classes), and derby (1,110 classes). Only 3,329 of the loaded classes are from Spark packaged classes.

A large number of loaded classes also results in a large interpreter time. The more classes being loaded leads to an increase in the number of different methods that are invoked, where each method has to be interpreted at the beginning. On average, a Spark client invokes 242,291 unique methods, where 91% of them were never compiled by JIT-compiler. In comparison, a Hive client only invokes 113,944 unique methods, while 96% of them were never JIT-compiled.

**Breaking down Spark’s warm-up time.** We further drill down into to one query (query 13 on SF 100) to un-
understand the long warm-up time of Spark. While different queries exhibit different overall behaviors and different runtime, the pattern of JVM warm-up overhead is similar, as evidenced by the stable warm-up time. Figure 4.6 shows the breakdown of this query. The query completion time is 68 seconds, and 24.6 seconds are spent on warm-up overhead. 12.4 seconds of the warm-up time are spent on the client while the other 12.2 seconds come from the executors. Note that a majority of executors’ class loading time is not on the critical path because executors are started immediately after the query is submitted, which allows executors’ class loading time to be overlapped with the client’s warm-up time. However, at the beginning of each stage the executor still suffers from significant warm-up overhead that comes primarily from interpreter time.

**Hive.** Hive parallelizes a query using different JVM processes, known as containers, whereas each container uses only one computation thread. Therefore within each container the warm-up overhead has a similar pattern with the HDFS client shown earlier. Hive and Tez also reuses containers to process tasks of the same query, therefore the JVM warm-up overhead can be amortized across the lifetime of a query.

### 4.4 Summary of Findings

Our analysis reveals that the JVM warm-up time is commonly the bottleneck of short running jobs, even when the job is I/O intensive. For example, 33% of the time in an HDFS 1GB sequential read is spent in warm-up, and 32% of the Spark query time on 100GB data size is on warm-up. The warm-up time stays nearly constant, indicating that its overhead becomes more significant in well parallelized short running jobs. In practice, many workloads are short running. For example, 90% of Facebook’s analytics jobs have under 100GB input size [3, 9], and majority of the real-world Hadoop workloads read and write less than 1GB per-task [85]. Furthermore, Ousterhout et al. [59] show a trend of increasingly short running jobs with latency in the hundreds of milliseconds. This shows that both the data size and latency of data-parallel workloads are trending to be smaller. We also observe that multi-layered systems exacerbates the warm-up overhead, as they tend use more classes and methods, increasing class loading and interpretation times.

### 4.5 Industry Practices

While JVM performance has been actively studied over the last 20 years, most of the improvements focused on GC [26, 51, 52, 62] and JIT [42, 74, 76, 75, 77] instead of warm-up. One reason is that it is assumed that workloads are using long-running JVMs. For example, traditional JVM benchmarks, such as DayTrader [7] and SpecJBB/SpecJVM [70, 71], all assume the use of long running JVMs. This study has shown that this paradigm has changed on data-parallel systems, and efforts to address warm-up overhead should be increased moving
forward.

Nevertheless, there exists some industry practices to address warm-up overhead. Despite the study showing clear overhead issues in Spark and Hive on Tez, both in fact already implement measures to reduce JVM warm-up overhead at the application layer. Both reuse the same JVM on each host to process the different tasks of each job (query in our case), thus amortizing the warm-up overhead across the life-time of a job. Spark runs one JVM or executor on each node that has the same life time as the job. Hive on Tez runs each task on a separate JVM (i.e., YARN container) and will try to keep reuse containers for new tasks. (Container reuse is the key feature introduced in Tez compared to Hadoop MapReduce.) A client could be designed to take multiple jobs from users and run them seemingly as one long job, which would allow multiple jobs to continue to use the same JVM. However, it requires domain expertise to determine whether such reuse is safe. One must consider what static data should be re-initialized or which threads need to be killed. The use of third-party libraries further exacerbate the problem as they may contain stale static data and create additional threads. This is perhaps the reason that these systems do not allow JVM to be reused across different jobs unless the client is specifically designed to process multiple jobs. Nailgun maintains a long-running JVM, and allows a client to dynamically run different applications on it. However, it does not take any measure to ensure that the reuse is consistent and the burden is on the users to decide whether a reuse is safe. In fact, naively reuse (unmodified) JVM does not even work when running the same Hive query twice, as the second run will crash because some static data in YARN needs to be properly reinitialized.

There are also a few solutions that change the JVM to address the warm-up overhead. The most advanced ones are perhaps on mobile platforms. The previous version of the Android runtime (ART) (Android Marshmallow) would compile the entire app when it is first downloaded to gain native performance, but it suffered from the various limitations including large binary size and slow updates. The latest version of ART (Nougat) uses a new hybrid model. It first runs an app with an interpreter and JIT-compiles hot methods, similar to OpenJDK’s HotSpot JVM. However, ART also stores profiling data after the app’s run, allowing a background process to compile the select methods into native code guided by the profile. These methods now no longer suffer the warm-up overhead the next time this app is used. ART also statically initializes selected classes during the compilation and stores them in the application image to reduce class loading time.

The Excelsior JET, which is a proprietary JVM, compiles the bytecode statically into x86 native code before running the application, similar to older versions of ART. This eliminates both class loading and interpreted overhead, but this is at the cost of losing the performance benefit provided by profile-guided JIT compiler.

Other programming methods exist to reduce warm-up time. One can try to make JIT-compile more aggres-

---

1 The container reuse in Tez is less predictable and cannot be taken advantage of by a smart user unlike with Spark, as there is a threshold for how long a container will be kept.
sively by changing the threshold with -XX:CompileThreshold. It is also possible to trigger class loading manually before classes are actually needed by either referencing the class or directly loading it. This is only useful if done off of the critical path. An example is that Spark’s executor is created before it actually receives any work allowing it to load classes. Similarly, one could potentially trigger JIT-compilation manually by invoking a method many times. Not only is this only useful if done off the critical path, but there are also other limitations. One has to ensure that the invocation has no side effects to the program state. Furthermore, one must also be wary of the parameters and path the method takes because the JIT-compiler is heavily guided by run-time profile, and unrealistic invocations could result in code less optimized for cases a developer cares about.
Chapter 5

Design of HotTub

The design goal for HotTub is to allow applications to share the “warm” data, i.e., loaded classes and compiled code, thus eliminating the warm-up overhead from their executions. We considered two design choices: explicitly copy data among JVMs, or reuse the same JVMs after properly resetting states. We began implementation of the first design, trying to save class metadata and JIT-compiled code to disk for reuse in the next JVM process, similar to Android runtime [6]. We were able to share loaded classes, but eventually rejected this design because it is too complicated to maintain the consistency of all the pointers between the JVM address spaces. For example, the JIT-compiler does not produce relocatable code; a compiled method may directly jump to another compiled method. To maintain consistency, we either have to allocate all the loaded classes and compiled methods at the exact same addresses, which is inflexible, or fix all the pointer values, which is impractical as we have to interpret every memory address in compiled code. We chose the “reuse” design, which proved to be simpler, and we can leverage existing JVM features, such as garbage collection, to properly throw out stale data.

Figure 5.1 shows the architecture of HotTub. It is a drop-in replacement – users simply replace java with HotTub and can run their Java application with normal command. Running java will spawn a HotTub client, which attempts to contact a warmed-up JVM, known as a HotTub server, to run on. We refer to a reusable JVM as server because it is designed to be long running. After a server has completed a run, it will send the return code

![Figure 5.1: Architecture of HotTub.](image-url)
to the client allowing the client to return normally to the user. The server will then run garbage collection, and reset the JVM state in preparation for the next client.

Next we discuss HotTub’s client algorithm as shown in Figure 5.2. First, an important consideration is the reuse policy, i.e., which applications are allowed to share the same JVM. In order to gain the most benefit from an existing JVM it is ideal to run as similar a workload as possible on it. An application that performs similar logic and traverses the same code paths will reuse the same classes and compiled code. However, if the new application is significantly different from the previous one, then these benefits are reduced. In HotTub, a client first computes a checksum of the classpath and every file containing classes, which are generally JAR (Java Archive) files, on the classpath. Only the servers with the same checksum are candidates for reuse. While this limits reuse potential, it ensures large overlap of warm data. It also ensures that clients always uses the same classes, avoiding inconsistency problems.

In addition, the client appends an integer between 0 and POOL_SIZE to the checksum (line 5 in Figure 5.2), creating an ID to use as an address to contact a specific server. The client tries to connect to each ID in this range, and reuses the first connected server. If connect fails because the server is currently busy the client tries the next server. If connect fails because no server exists, or all servers are busy, the client forks a child process to create the server (line 11-12). The reason that we need to fork and exec java in the child, instead of directly exec java without fork, is that the user could be waiting for the java command to finish. Forking allows the parent to return after the application finishes, while the child process, which is now a warmed-up JVM, waits for reuse.

This design has a number of benefits. First, it is simple. The clients and servers on each node do not require a central coordinator, avoiding a potential bottleneck or central point of failure. In addition, it selects the longest running server that will likely result in the highest reusage of warm data. This is because the longest running server has had the most time to warm-up, JIT-compiling the most methods and loading the most classes. Finally, reusing the same JVM process across applications also offers caching benefits – between consecutive application
runs the warm data stays in CPU caches because its memory address remains the same, and the OS does not need to flush the TLB.

## 5.1 Maintain Consistency

The main challenge to HotTub’s design is to ensure that the application’s execution on HotTub is consistent with the execution on an unmodified JVM. Data on stack and heap does not impose inconsistency problem, because at the end of a JVM application’s execution, all of the application’s stack frames have naturally ended. HotTub further garbage collects the heap objects with root references from the stack, therefore all of the heap objects that are application specific are also cleared. The remaining items, namely the loaded classes, compiled code, static variables, and file descriptors, need to be shared between reuse. Next we describe how HotTub maintains their consistency between reuse.

### Class consistency.
HotTub must ensure that any class it reuses is the same as what would be loaded by an unmodified JVM, as classes could potentially change in between runs, or during runs. Maintaining class consistency also ensures the consistency of compiled code as it is compiled from the class bytecode. The checksum mechanism used by the client only ensures the consistency of classes on the application classpath, which are loaded by the default class loader. While this accounts for the majority of loaded classes, an application can also implement a custom class loader, which has user-defined searching semantics, or dynamically generate a class.

Fortunately, any classes loaded by a custom class loader will not impose inconsistency issues for HotTub because a custom class loader must be instantiated by user code. This makes reuse impossible as every run will create a new instance of the class loader with no data from the previous run, causing it to load any class normally. Similarly, classes that are dynamically generated are loaded by custom class loaders in practice and are not an issue for consistency. However, there is no performance gained from reusing any classes that are loaded by custom class loaders as they are simply not reused.

### Static variable consistency.
At the end of application execution, static variables have values from the previous execution. Therefore HotTub needs to reinitialize them first to their default type value and then reinitialize them with their class initialization code. HotTub uses a simple policy. When the server is about to be reused, it reinitializes the static variables all at once by invoking the static initializer, namely `<clinit>`, of each class.

HotTub needs to maintain the correct order of the invocations to `<clinit>` of different classes. For example, class A’s initialization may depend on class B having already been initialized. HotTub maintains the correct order by recording the order of class initializations when they are first initialized, and replaying the initializations in the same order before each reuse.

Unfortunately, reinitializing all the static data before the start of application is not consistent with the JVM
specification [27] when the initialization of static variables have timing dependencies. Consider the following Java example:

```java
1 Class A {
2    static int a = 0;
3    void foo () { a++; }
4 }
5 Class Bad { // bad practice
6    static int b = A.a;
7 }
```

According to the JVM specification, the value of variable b in class Bad depends on when class A gets initialized. For example, if foo() has been called 3 times before Bad is referenced, then b will be initialized to 3. HotTub will initialize it to 0 in this case.

However, it is worth noting that static initialization that has timing dependency is a well known bad programming practices [1][73]. It makes programs hard to reason about and difficult to test. Furthermore, multi-threading makes the matter worse as foo() in the previous example can be executed concurrently. In our experiments, we carefully examined the static initializers of the experimented systems, and none of them use such practice.

Another potential issue is when there exists a dependence cycle in the class initialization code of multiple classes, HotTub could lead to inconsistent behavior. For example, consider the following code snippet:

```java
1 Class A {
2    static int a = 10;
3    static int b = B.b;
4 }
5 Class B {
6    static int a = A.a; // set to 10 in HotSpot; 0 in HotTub
7    static int b = 12;
8 }
```

There exists a circular dependency between class A and B in their static variables. Assume class A begins initialization first. Under HotSpot, it will first initialize A.a to 10 (line 2), and starts to initialize class B because A.b depends on it. When executing line 6, HotSpot detects that there is a circular dependency, and it will proceed to finish the initialization of the current class (class B), setting B.a to 10 and B.b to 12, before continuing the initialization of class A. HotTub, however, will run each static initializer from beginning to the end before moving on to the next one. Therefore in this case, it will first initialize class B because B’s initialization finished before A’s in the initial run. Since class A has not been initialized yet, HotTub will set B.a to the initial value of A.a, which is 0, leading to an inconsistent value on B.a. Note that circular dependence in static initializers is also a known bad practice, and the JVM specification explicitly warns about its dangers and discourages its use [27].
CHAPTER 5. DESIGN OF HotTub

**File descriptor consistency.** HotTub will close the file descriptors (fd) opened by the application at the reset phase so that they will not affect the next client run. The only remaining open fds are those opened by the JVM itself, mostly for JAR files. HotTub also closes stdin, stdout, and stderr at the end of an application’s execution in the reset stage. After the client selects a server JVM for reuse, the client first sends all fds it has opened to the server, including stdin, stdout, and stderr, so that the server can inherit these file descriptors and have the same same open files as the client.

However, it is possible that a file descriptor opened by the client conflicts with an open file descriptor used by the server JVM. For example, if the user invokes HotTub with the command $ java 4>file, HotTub cannot reuse a server with fd 4 in use. Therefore when selecting a server for reuse, HotTub also checks if the server has open fds that conflict with a client’s redirected fd, and only reuse servers that do not have such a conflict.

**Handling signals and explicit exit.** HotTub has to handle signals such as SIGTERM and SIGINT and explicit exit by the application, otherwise it will lose the target server process from our pool. If application registers its own signal handler, HotTub forwards the signal. Otherwise, HotTub handles signals and application exits by unwinding the stack of non-daemon Java threads and killing them. Java “daemon” threads are not normally cleaned at JVM exit as they simply exit with the process. However, for consistency, HotTub must kill these threads. This sets the JVM to the same state as if the application finishes normally. The server then closes connection to client, so the client exits normally. However, if the application calls _exit in a native library, HotTub cannot save this server process from being terminated.

### 5.2 Limitations

HotTub cannot handle SIGKILL. Therefore, if the user sends `kill -9` to a HotTub server we will lose it for future reuse. However, it is most likely that the user only wants to kill the client `java` process, which will not cause us to lose the server because the server and client are in different processes.

Unfortunately, this use of separate processes can raise problems if a user expects the application to run in the same process as `java`. For example, YARN terminates a container by first sending SIGTERM to the process identified by a PID file, followed by SIGKILL. This would not cause HotTub to violate consistency, as the server will be killed and the client subsequently exits on a closed connection. However, this will disable HotTub from reusing the server. Therefore we had to modify the management logic in YARN to disable “kill -9”.

The use of HotTub raises privacy concerns. HotTub limits reuse to the same Linux user, as cross user reuse allows a different user to execute code with the privileges of the first user. However, our design still violates the principle “base the protection mechanisms on permission rather than exclusion” [65]. Although we carefully clear and reset data from the prior run, an attacker could still reconstruct the partial execution path of the prior
run via timing channel. For example, by measuring the execution time of the first time invocation of a method the attacker can infer whether this method has been executed, and thus JIT-compiled, in the prior run. In our current implementation we are not zeroing out the heap space after GC. This allows malicious users to use native libraries to read the heap data from prior runs.

HotTub cannot maintain consistency if the application rewrites the bytecode or compiled code of a class on the classpath after it has been loaded, and does not write the modifications back to the class file. In such cases, the in-memory bytecode or compiled code will be different from the checksum computed by HotTub. It is difficult to detect the bytecode rewriting because the application can always bypass the JVM using a native library to modify any memory locations in the address space. However, modifying the bytecode of a loaded class is undefined behavior as the JVM may be already using a compiled method, thus the changes to bytecode will have no effect. In practice we have never encountered such cases. Note that the HotSpot JVM performs its own form of bytecode rewriting, which is not a problem for HotTub as this is only done for performance optimizations and preserves the original semantic.

HotTub currently only targets the Java Virtual Machine runtime. Other runtimes such as Microsoft’s Common Runtime Language (CLR) also exhibits similar warm-up overhead properties. Similar to class loading done by the JVM, CLR must load portable executable (PE) file, which is similar to a Java classfile, that contains metadata required at runtime such as type definitions and member signatures. To the best of our knowledge CLR operates similar to typical JVMs, where an interpreter will execute bytecode or an intermediate language, until a JIT-compiler can produce native code for the method. Having these properties should exhibit similar warm-up overhead of class loading and interpretation in CLR, so implementing HotTub for CLR could potentially produce similar speed-ups.
Chapter 6

Implementation of HotTub

The client is implemented as a stand-alone program with 800 lines of C code, and the server is implemented on top of OpenJDK’s HotSpot JVM by adding approximately 800 lines of C/C++ code. We use Unix domain sockets to connect a client with servers. A nice feature of Unix domain socket is that it allows processes to send file descriptors. Therefore the client simply send its open file descriptors, including stdin, stdout, stderr together with other redirected ones, to the server. This avoids sending the actual input and output data across processes. Next we discuss the implementation details of HotTub.

Threads management. HotTub does not use any additional thread in JVM for its management task. Instead, it uses the Java main thread. At the end of the application execution after the main() method finishes, we do not terminate the Java main thread. Instead we uses it to perform the various reset tasks including (1) kill other Java threads, (2) set all static variables to their default type value, (3) garbage collect the heap, (4) optionally unload native libraries. It then waits for a client connection. When it receives another client connection, it reinitializes the static variables, sets up the file descriptors properly, sets any Java properties, sets any environment variables, and finally invokes the main() method.

Complication arises when the JVM receives a signal or a thread calls System.exit. In these cases, we need to use the thread that receives the signal or calls System.exit to clean up the all Java threads.

Static reinitialization. HotTub handles a few technical challenges when implementing the replay of class initialization. One challenge is with enumeration classes. For each class in Java there exists a java.lang.Class object that contains information about the class, such as the methods and fields it contains, so that it can be queried for reflection inspections. For enumeration classes, this object contains a mapping of each enumeration constant string name to its object. Because after reinitialization there will be new objects allocated for each enumeration

\[1\] Theoretically we should unload native libraries because HotTub does not verify their consistencies. However we observe that native libraries typically do not impose inconsistency issues, e.g., they typically do not use static data. Therefore we make unloading them optional.
constant, HotTub also has to update the mapping in this \texttt{java.lang.Class} object in each class.

Another challenge is the JIT-compiler’s inlining of static final references, i.e., a compiled method could directly reference the address of a static final object. However, after reinitialization, a new object will be created so that the old reference is no longer valid. HotTub solves this by disabling the inlining of static final references.
Chapter 7

Performance of HotTub

We conduct a variety of experiments on HotTub to evaluate its performance on the following dimensions: (1) speed-up over an unmodified JVM repeating the same workload; (2) speed-up when running different workloads; (3) management overhead imposed by HotTub (e.g., reset, client/server management). All experiments are performed on the same environment and settings as described in Chapter 4.

7.1 Speed-up

Table 7.1 shows HotTub’s speed-up compared with unmodified HotSpot JVM. We run the same workload five times on an unmodified JVM and six times on HotTub. We compare the average runtime of the five unmodified runs with the five reuse HotTub runs, excluding the initial warm-up run. For Spark and Hive, we run the same 10

<table>
<thead>
<tr>
<th>Workload</th>
<th>Completion time (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unmod.</td>
<td>HotTub</td>
</tr>
<tr>
<td>HDFS read 1MB</td>
<td>2.29</td>
<td>0.08</td>
</tr>
<tr>
<td>HDFS read 10MB</td>
<td>2.65</td>
<td>0.14</td>
</tr>
<tr>
<td>HDFS read 100MB</td>
<td>2.33</td>
<td>0.41</td>
</tr>
<tr>
<td>HDFS read 1GB</td>
<td>7.08</td>
<td>4.26</td>
</tr>
<tr>
<td>Spark 100GB best</td>
<td>65.2</td>
<td>36.2</td>
</tr>
<tr>
<td>Spark 100GB median</td>
<td>57.8</td>
<td>35.2</td>
</tr>
<tr>
<td>Spark 100GB worst</td>
<td>74.8</td>
<td>54.4</td>
</tr>
<tr>
<td>Spark 3TB best</td>
<td>66.4</td>
<td>41.4</td>
</tr>
<tr>
<td>Spark 3TB median</td>
<td>98.4</td>
<td>73.6</td>
</tr>
<tr>
<td>Spark 3TB worst</td>
<td>381.2</td>
<td>330.0</td>
</tr>
<tr>
<td>Hive 100GB best</td>
<td>29.0</td>
<td>16.2</td>
</tr>
<tr>
<td>Hive 100GB median</td>
<td>38.4</td>
<td>25.0</td>
</tr>
<tr>
<td>Hive 100GB worst</td>
<td>206.6</td>
<td>188.4</td>
</tr>
</tbody>
</table>

Table 7.1: Performance improvements by comparing the average job completion time of an unmodified JVM and HotTub. For Spark and Hive we report the average times of the queries with the, best, median, and worst speed-up for each data size.
Figure 7.1: HotTub successfully eliminates warm-up overhead. Unmodified query runtime shown against a breakdown of a query with reuse. There are 10 queries run on 2 scale factors for BigBench on Spark. Interpreter and class loading overhead are so low they are unnoticeable making up the difference.

Table 7.2: Comparing cache, TLB, and branch misses between HotTub (H) and an unmodified JVM (U) when running query 11 of BigBench on Spark with 100GB data size. The numbers are taken from the average of five runs. All page faults are minor faults. “Rate diff.” is calculated as (U Rate - H Rate)/(U Rate), which shows the improvement of HotTub on the miss rate. Perf cannot report the number of L1-icache loads or memory references to know the corresponding rates.

queries that we used in our study. Note that in this experiment the systems we run are unmodified, unlike the ones we used in Chapter 4 that had to be instrumented. Therefore the unmodified systems’ runtimes are slightly faster than the ones in Chapter 4.

The results shows that HotTub significantly speeds up the total execution time. For example, HotTub reduces the average job completion time of the Spark query with the highest speed-up by 29 seconds on 100GB data, and can speed-up HDFS 1MB read by a factor of 30.08. Amongst nearly 200 pairs of trials, a job running in a reused HotTub JVM always completed faster compared to an unmodified JVM.

Enabling our performance counters, we observe that indeed HotTub eliminates the warm-up overhead. In all the experiments, the server JVM spends less than 1% of the execution time in class loading and interpreter. Figure 7.1 shows Spark queries running on HotTub with nearly zero warm-up overhead.

Figure 7.2 shows how the runtime decreases over the number of reuses. While the significant speedup comes from the first time reuse, it also shows that for this particular short running job the JVM will not be completely
Figure 7.2: HotTub iterative runtime improvement. A Sequential 1MB HDFS read performed repeatedly by HotTub. Iteration 0 is the application runtime of a new JVM, while iteration N is the Nth reuse of this JVM.

warm by the end of the first run and require multiple iterations before reaching peak performance. Figure 7.2 also shows that it takes 12 iterations before the JVM becomes fully warmed-up. This further suggests that short running jobs cannot even reach max JVM performance by the end of its execution, which further emphasizes the necessity of reusing JVMs on short jobs. Long running jobs, however, will likely fully warm up the JVM before their execution ends on first run.

To understand HotTub’s performance behavior in detail, we further compare the hardware performance counters. Table 7.2 shows the result. HotTub already significantly reduces the number of memory accesses because the classes are already loaded and bytecode compiled. For the Spark executor there are almost half as many cache references reported by perf and the Spark client shows an even higher reduction, up to 9x. The reduction in accesses appears large, but is consistent with the 1.74x speed-up experienced by running this query on HotTub. It further reduces the number of various cache misses. The reduction comes from multiple sources. First, HotTub results in less memory accesses and smaller footprint because some data is no longer needed to be accessed (e.g., bytecode that are already compiled will not be accessed). Second, because the applications run in the same JVM process, warm data does not need to be reallocated between the runs, therefore the cached data can be reused and the number of cold misses get reduced. The OS also does not need to flush the TLB between the runs. Finally, the reduction of instruction cache and iTLB misses is likely afforded by eliminating interpreter execution and JIT-compiler’s instruction cache usage optimization. For example, JIT-compiler will arrange the basic blocks in the order of frequently taken branches.

The numbers in Table 7.2 also show that HotTub reduces the accesses and misses in the Spark client much more than the executor. This is likely due to the nature of the work each component does. The executor is processing large amounts of data, taking the majority of time and memory references, even in a reused run, while
the client performs much less work. Since the warm-up overhead reduction is constant, it follows that the executor should be less affected, while the client will be heavily affected as it spent more of its time performing warm-up.

### 7.2 Sensitivity to Difference in Workload

Table 7.3 compares HotTub’s performance sensitivity to the workload differences between training and testing runs. We warm up the JVM by repeatedly run a single “training query” four times, and apply it once on the “testing query”. Note that we cannot apply the test run more than once for this experiment because the JVM will then be warming up with the testing query. We repeat this process five times and take the average runtime of the testing queries, and then report the speed-up of this average runtime over the runtime of unmodified JVM running the testing query. The result shows that, for our tests, HotTub can achieve at least 1.47 speed-up.

Query 11 and 1 observe the largest speed-up when the training and testing queries are the same. The other queries observe best speed-up when running on a JVM trained from a different query. This is due to the large variance observed in our experiment. All of the measured runtime of testing queries fall into the range of \((\text{mean} - \text{variance}, \text{mean} + \text{variance})\), where mean and variance are of the five measured runs where the training and testing queries are the same. This also indicates that different queries use many similar classes and code.

### 7.3 Management Overhead

Compared with an unmodified JVM, HotTub adds overhead in three phases: when a client connects to a server, when the server runs class initialization, and when the server resets the static data. The first two are on the critical path of the application latency while the third merely affects how early this JVM can be reused again. The overhead for connecting to a server when there are no servers in the pool is 81ms. Once servers are available for re-use, the connection overhead drops to 300µs. The overhead added to the critical path from class reinitialization is, on average, 350ms for Hive on Tez containers, 400ms for Spark executors, and 720ms for Spark clients. The time taken to reset static data is dominated by garbage collection and only takes no more than 200ms because the application’s stack frames have ended, thus there are few roots from which GC has to search from. Root objects
are objects assumed to be reachable, such as static variables or variables in live stack frames.

HotTub also adds overhead on the memory usage of the system as the server processes remain alive after the application finishes. The number of servers to be left alive on a node can be configured by the user, for our evaluation we arbitrarily chose 5. An unused server takes up approximately 1GB of memory.
Chapter 8

Related Work

We discuss prior studies on the performance of the JVM and data-parallel distributed systems. Commercial and industry solutions have been discussed in Chapter 4.5. Our work distinguishes itself as it is the first to study the performance implications of JVM warm-up overhead on data-parallel systems.

Performance of garbage collection (GC). Recently, a few proposals are made to improve the GC performance for big data systems [26, 51, 52, 62]. Gog et al. observe that objects often have clearly defined lifetimes in big data analytics systems [26]. Therefore they propose a region-based memory management [81] system where developers can annotate the objects with the same lifetime. Maas et al. [51, 52] observe that different JVMs running the same job often pause to garbage collect at the same time given the homogeneous nature of their executions, therefore they propose a system named Taurus that coordinates the GC pauses among different JVM processes. Our work is complementary as we focus on studying the JVM warm-up overhead, while Broom and Taurus only focused on GC. HotTub can also be integrated together with Broom and Taurus to provide comprehensive speed-up of the JVM runtime. Comparing the design of Taurus and HotTub also reveals interesting trade-offs. Taurus does not modify the JVM itself, therefore users will have less reliability concerns in deployment. However, its capability to control the JVM is restricted to the interfaces JVM exposed. Taurus requires the JVMs in the network to coordinate via a consensus protocol, whereas HotTub uses a simpler design that makes it standalone and does not require network communication. Consequently HotTub can also benefit non-distributed applications.

Other papers studied GC performance on non-distributed workload. Appel [8] uses theoretical analysis to argue that when physical memory is abundant, GC can achieve high performance comparable to other memory management techniques. Hertz et al. further validated this via more thorough experimental evaluation [32], but they also found that the performance of GC deteriorate quickly when free memory becomes scarce. Others have compared the performance of general GC algorithms (e.g., generational GC) versus customized ones and
concluded that general algorithms achieve good performance in most cases \[11, 14, 88\].

**Performance studies on data-parallel systems.** A handful of works have thoroughly analyzed the performance of data-parallel systems \[47, 49, 58, 60\]. However they did not study the JVM performance impact. Ousterhout *et al.* comprehensively studied the performance of Spark \[58\], and revealed that network and disk I/O are no longer the bottleneck. Interestingly, they found that CPU is often the bottleneck, and a large amount of CPU time is spent in data deserialization and decompression. However, because they only analyzed Spark itself, they did not further drill down to provide a low level understanding of such high CPU time. Using the same workload, our study suggests that the class loading and bytecode interpretation are likely the main cause of deserialization and decompression. Pavlo *et al.* \[60\] compared Hadoop MapReduce with DBMS, and found that data shuffling is often the bottleneck for MapReduce. Jiang *et al.* \[47\] analyzed the performance of Hadoop MapReduce. McSherry *et al.* \[54\] surveyed the existing literature on data-parallel systems and their experimental workload, and concluded that many of the workloads use small data input sizes that can be well handled by a single threaded implementation. This has similar implications as the other studies on real-world analytic jobs where most jobs are short running because of the small input size \[85\], where the JVM warm-up time is even more significant.

Other works on improving data-parallel systems performance focused on scheduling \[2, 4, 33, 41, 59, 87\], high performance interconnect \[17, 19, 43, 48, 84\], optimization for multi-cores \[16, 53, 63\], and removing redundant operations \[64\]. Our work is complementary as it focuses on JVM-level improvements.
Chapter 9

Concluding Remarks

We started this project with the curiosity to understand the JVM’s overhead on data-parallel systems, driven by the observation that systems software is increasingly built on top of it. Enabled by non-trivial JVM instrumentations, we observed the warm-up overhead, and were surprised by the extent of the problem. We then pivoted our focus on to the warm-up overhead by first presenting an in-depth analysis on three real-world systems. Our result shows the warm-up overhead is significant, and can be exacerbated as jobs become more parallelized and short running. We further designed HotTub, a drop-in replacement of the JVM that can eliminate warm-up overhead by amortizing it over the lifetime of a host. Evaluation shows it can speed-up systems like HDFS, Hive, and Spark, with a best case speed-up of over 30.08X.

9.1 Future Work

This work has identified and analyzed the problem of JVM warm-up overhead in data-parallel systems, but fully exploring the different solutions to this problem remains as future work. The presented solution, HotTub, is just one way to address this problem and had many of its design decisions limited by time and engineering effort. Alternative approaches that utilize ahead of time compilation to different degrees or store profiling information to disk for future JVMs can address the problem of warm-up overhead with different benefits.

Ahead of time (AOT) compilation is the traditional model of compilation used in many programming languages such as C and C++. Using AOT compilation could eliminate both class loading and interpretation time, but at the cost of certain features enabled by a dynamic environment. If the class data is directly in the compiled binary then the cost of search and defining a class, or dynamic class loading, is removed. The problem is that in many of these systems not all of the classes are known or available before runtime. A custom classloader could potentially load a class over the network. If this feature is to still be supported either a library of the class must
be capable of being loaded at runtime, or classes loaded through this feature must be loaded and interpreted as they normally would. Another problem is that by using AOT compilation the benefits of JIT compilation are lost. Many JIT compiler optimizations utilized by the JVM can improve performance significantly. For example, speculating the result of a virtual call can avoid performing the lookup and only require checking the runtime type matches the expectation. A JIT compiler could also take advantage of values specific to the hardware the program is currently being executed on, such as optimizing for cache sizes. AOT compilation would sacrifice these special optimizations, but does allow code to be immediately run natively avoiding the expensive cost of interpretation. This method was used in Android [6] (Marshmallow) and is used by the The Excelsior JET [21], which is a proprietary JVM. The implications of using this method are not well documented, and likely vary depending on the application. From what was seen in the study on JVM warm-up overhead it is very likely that AOT compilation would be a great improvement for latency-sensitive applications as it directly eliminates the warm-up overhead of both class loading and interpretation, which was costly.

The new version of the Android [6] (Nougat) uses a new hybrid model. It uses both an interpreter and JIT compiler like typical JVM runtimes, but it stores profiling data to disk after an application run. This allows a background process to compile hot methods to native code with the profile guided optimizations that JIT compilers take advantage of. It will also initialize select classes and store them in the application image to avoid loading the class at runtime. Similar to the AOT compilation model a complete comparison of this model against the others has not been evaluated on non-mobile platforms. Implementations of this model for server computers to run data-parallel systems could see great improvements in performance, eliminating warm-up overhead while retaining the advantages of JIT compilation.
Bibliography


[34] Hive. [http://hive.apache.org]


[38] hypertable: why we chose CPP over Java. [https://code.google.com/p/hypertable/wiki/WhyWeChoseCppOverJava]


[40] Interactive query with apache hive on apache tez. [http://hortonworks.com/hadoop-tutorial/supercharging-interactive-queries-hive-tez/]


[44] Quora: In what cases is Java faster than C. https://www.quora.com/In-what-cases-is-Java-faster-if-at-all-than-C

[45] Quora: In what cases is Java slower than C by a big margin. https://www.quora.com/In-what-cases-is-Java-slower-than-C-by-a-big-margin


[73] Static initializers will murder your family. [http://meowni.ca/posts/static-initializers/](http://meowni.ca/posts/static-initializers/)


