Don't Get Caught In the Cold, Warm-up Your JVM

Understand and Eliminate JVM Warm-up Overhead in Data-parallel Systems

David Lion, Adrian Chiu, Hailong Sun*, Xin Zhuang, Nikola Grcevski†, Ding Yuan

*University of Toronto, *Beihang University, †Vena Solutions*

Abstract

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 paper analyzes the extent and causes of the JVM performance overhead in the above mentioned systems. Surprisingly, we find that the warm-up overhead, i.e., class loading and interpretation of bytecode, is frequently the bottleneck. For example, even an I/O intensive, 1GB read on HDFS spends 33% of its execution time in JVM warm-up, and Spark queries spend an average of 21 seconds in warm-up.

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 new JVM that amortizes the warm-up overhead over the lifetime of a cluster node instead of over a single job by reusing a pool of already warm JVMs across multiple applications. The speed-up is significant. For example, using HotTub results in up to 1.8X speed-ups for Spark queries, despite not adhering to the JVM specification in edge cases.

1 Introduction

A large number of data-parallel distributed systems are built on the Java Virtual Machine (JVM) [25]. These systems include distributed file systems such as HDFS [28], data analytic platforms such as Hadoop [27], Spark [64], Tez [62, 76], Hive [32, 77], Impala [13, 36], and key-value stores such as HBase [29] and Cassandra [15]. A recent trend is to process latency-sensitive, interactive queries [37, 65, 75] with these systems. For example, interactive query processing is one of the focuses for Spark SQL [10, 64, 65], Hive on Tez [37], and Impala [36].

Numerous improvements have been made to the performance of these systems. These works mostly focused on scheduling [2, 4, 31, 38, 56, 84], shuffling overhead [17, 19, 40, 45, 81], and removing redundant computations [61]. Performance characteristics studies [44, 46, 55, 57] and benchmarks [18, 23, 34, 80] 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 [24, 47, 48, 59].

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 [35, 41, 42, 43, 58, 69, 83]. For example, the developers of Hypertable, 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" [35]. 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 [69]. 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 [23] 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 paper) [39, 73]. 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 new 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 [26]. 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.

This paper makes the following contributions.

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

This paper 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 [25] 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, 70].

This paper is organized as follows. Section 2 describes the instrumentations to the JVM used to measure its warm-up overhead. Section 3 presents the analysis of JVM performance. Section 4 and Section 5 describe HotTub's design, implementation, and limitations. We evaluate HotTub in Section 6. We survey the related work in Section 7 before we conclude.

2 Measure Warm-up Overhead

In this section we discuss how we instrument the JVM to measure its class loading and bytecode interpretation time with per-thread granularity. Section 3 describes how we use these instrumentations to study JVM overhead in data-parallel systems. We use OpenJDK's HotSpot

[1]`https://github.com/dsrg-uoft/hottub`

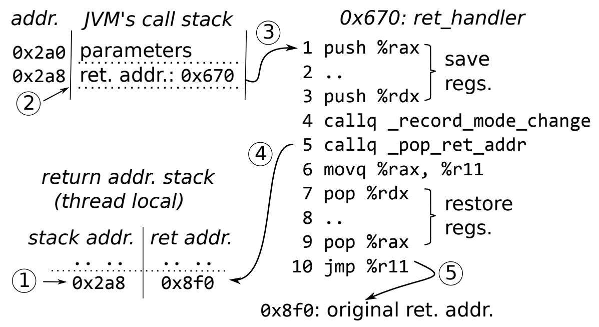

Figure 1: Intercepting mode changing returns.

JVM, version 1.8.0 build 25.66. HotSpot is the primary reference Java Virtual Machine implementation [47].

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.[2]

We instrument mode changing returns by replacing the original return address on the call stack with the address of our instrumented code. Figure 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 address with the address of our instrumented function `ret_handler` (`0x670`). (3) When B returns, it first jumps to `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. `ret_handler` is implemented in 15 assembly instructions.

[2]The interpreter can also directly jump into compiled code via on-stack-replacement (OSR) [22, 33], 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.

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 `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 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.

3 Analysis of Warm-up Overhead

This section 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.

3.1 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. It 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 [37, 65, 75]. 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. The versions we used are Hadoop-2.6.0, Spark-1.6.0, Hive-1.2.1, and Tez-0.7.0.

We benchmark Spark and Hive using BigBench [23]. It consists of 30 queries ranging over structured, semi-structured, and unstructured data that are modeled after real-world usage [23]. Its queries on structured data are selected from TPC-DS [79], which is widely used by SQL-on-Hadoop vendors like Cloudera [53], Hortonworks [54], Databricks [21, 66], 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. In addition, we found that at least 8 queries lead to heavy swapping at large data sizes, indicating that our system is not representative to run these queries.

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.* [55] 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

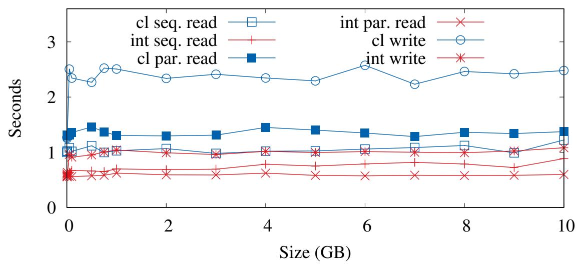

Figure 2: JVM warm-up time in various HDFS workload. “cl” and “int” represent class loading and interpretation time respectively. The x-axis shows the input file size.

time, and then simulates the scheduling of these tasks with the reduced execution time. 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 CPU resources. In addition, our instrumentation may not cover all threads. For example, some libraries can create their own threads which we do not instrument. We use our best effort to address this problem: we instrumented the JVM thread constructor to observe the creation of every application thread, and instrument those that at least load classes. However, there are still threads that are not instrumented.

3.2 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 2 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

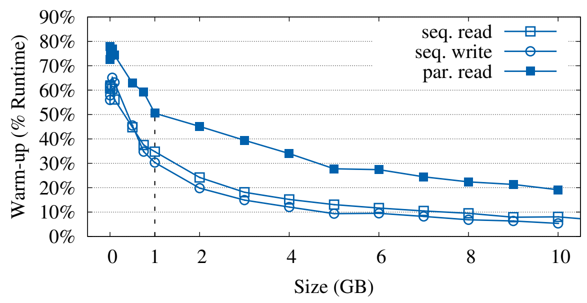

Figure 3: The JVM warm-up overhead in HDFS workloads measured as the percentage of overall job completion time.

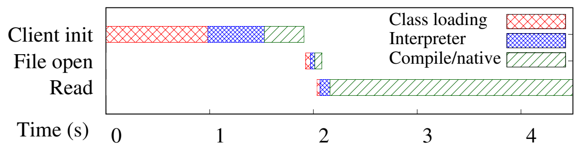

Figure 4: Breakdown of sequential HDFS read of 1GB file.

read because its parallelism allows the JVM to identify the "hot spot" faster.

Figure 3 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 [82] 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 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 5 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

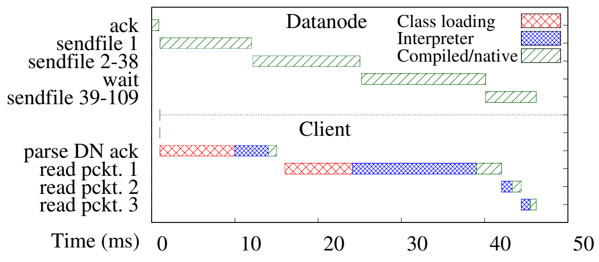

Figure 5: Breakdown of the processing of data packets by client and datanode.

	Read	Search	Define	Other	Total
Time (ms)	170	276	411	171	1,028

Table 1: Breakdown of class loading time.

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 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 5 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 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 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.

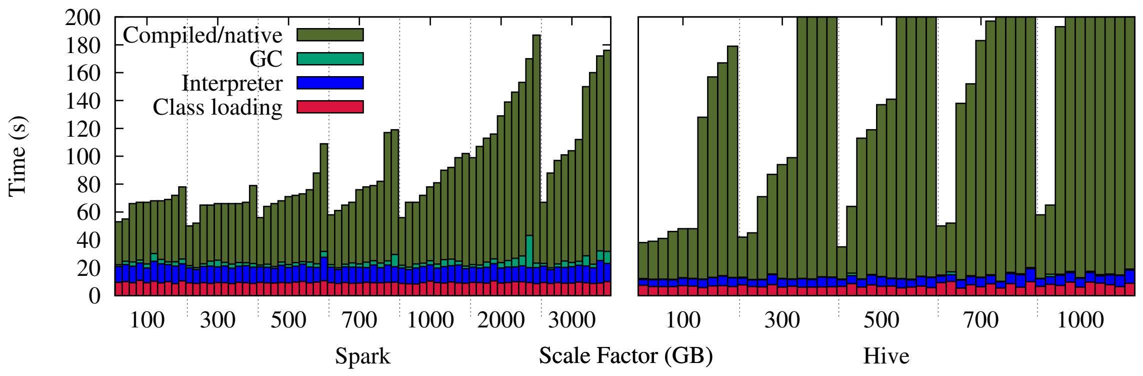

Figure 6: JVM overhead on BigBench. Overhead breakdown of BigBench queries across different scale factors. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has larger query time compared with Spark.

3.3 Spark versus Hive

Figure 6 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 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 7 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.

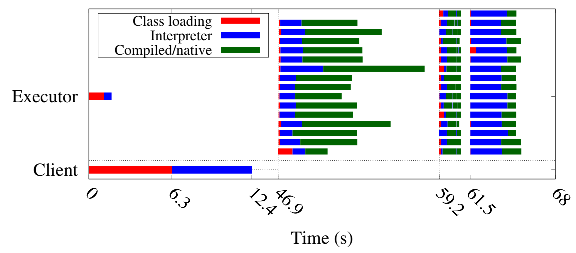

Figure 7: Breakdown of Spark's execution of query 13. It only shows one executor (there are a total of ten executors, one per host). Each horizontal row represents a thread. The executor uses multiple threads to process this query. Each thread is used to process three tasks from three different stages.

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.

3.4 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 [82]. Furthermore, Ousterhout *et al.* [56] 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.

3.5 Industry Practices

While JVM performance has been actively studied over the last 20 years, most of the improvements focused on GC [24, 47, 48, 59] and JIT [39, 71, 73, 72, 74] 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 [67, 68], 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.[3] 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 [52] 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) [6] (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 [5]. The latest version of ART (Nougat) [6] 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 [20], 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 aggressively 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

[3]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.

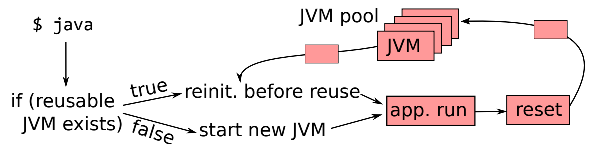

Figure 8: Architecture of HotTub.

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.

4 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 8 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

```
1  struct sockaddr_un add; // create unix sock.
2  char* sum = md5(classpath);
3  while (true) {
4    for (int i = 0; i < POOL_SIZE; i++) {
5     strcpy(add.sun_path,strcat(sum,itoc(i)));
6     if (connect(fd, add, sizeof(add))==0)
7       return reuse_server_and_wait(fd);
8     if (server_busy(i))
9       continue;
10    /* No JVM/server created. */
11    if (fork() == 0) // spawn new jvm in child
12      exec("/HotTub/java", args);
13    /* else, parent, go back to find server */
14   }
15 }
```

Figure 9: HotTub's client algorithm.

code 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 9. 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 9), 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.

4.1 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 [25] when the initialization of static variables have *timing* dependencies. Consider the following Java example:

```java
Class A {
  static int a = 0;
  void foo () { a++; }
}
Class Bad { // bad practice
  static int b = A.a;
}
```

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 depedency is a well known bad programming practices [1, 70]. 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
Class A {
  static int a = 10;
  static int b = B.b;
}
Class B {
  static int a = A.a; // set to 10 in
      HotSpot; 0 in HotTub
  static int b = 12;
}
```

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 [25].

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.

4.2 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" [63]. 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) [51] 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.

5 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.[4] 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 constant, HotTub also has to update the mapping in this `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.

[4]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.

Completion time (s)	Unmod.	HotTub	Speed-up
HDFS read 1MB	2.29	0.08	30.08x
HDFS read 10MB	2.65	0.14	18.04x
HDFS read 100MB	2.33	0.41	5.71x
HDFS read 1GB	7.08	4.26	1.66x
Spark 100GB best	65.2	36.2	1.80x
Spark 100GB median	57.8	35.2	1.64x
Spark 100GB worst	74.8	54.4	1.36x
Spark 3TB best	66.4	41.4	1.60x
Spark 3TB median	98.4	73.6	1.34x
Spark 3TB worst	381.2	330.0	1.16x
Hive 100GB best	29.0	16.2	1.79x
Hive 100GB median	38.4	25.0	1.54x
Hive 100GB worst	206.6	188.4	1.10x

Table 2: 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.

6 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 Section 3.

6.1 Speed-up

Table 2 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.

Perf. counter	Executor						Client					
	U	H	U/H	U Rate	H Rate	Rate Diff.	U	H	U/H	U Rate	H Rate	Rate Diff.
L1-dcache-misses	171M	81M	2.1x	1.839%	1.994%	-8.416%	154M	21M	7.3x	6.254%	6.115%	2.218%
L1-icache-misses	40M	13M	3.1x	-	-	-	44M	6M	7.3x	-	-	-
page faults	543K	122K	4.4x	-	-	-	851K	227K	3.7x	-	-	-
dTLB-load-misses	4,431M	3,087M	1.4x	0.080%	0.051%	36.418%	2,999M	375M	8.0x	0.327%	0.295%	9.894%
iTLB-load-misses	704M	228M	3.1x	3.424%	3.294%	3.805%	755M	97M	7.8x	3.359%	3.054%	9.078%
branch-misses	1,158M	597M	1.9x	0.913%	0.646%	29.234%	974M	119M	8.2x	3.270%	2.971%	9.141%

Table 3: 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.

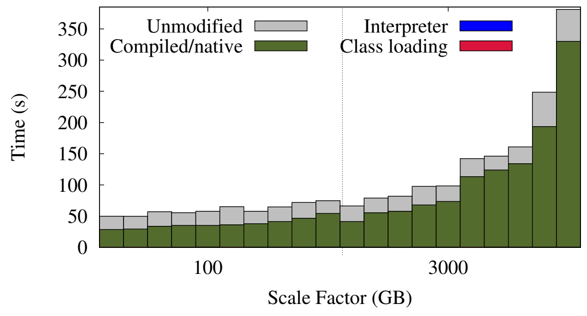

Figure 10: HotTub successfully eliminates warm-up overhead. Unmodified query runtime shown against a breakdown of a query with reuse. There are 10 queries run on 2 scale factors for BigBench on Spark. Interpreter and class loading overhead are so low they are unnoticeable making up the difference.

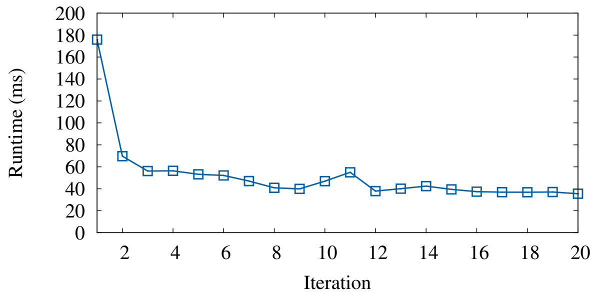

Figure 11: HotTub iterative runtime improvement. A Sequential 1MB HDFS read performed repeatedly by HotTub. Iteration 0 is the application runtime of a new JVM, while iteration N is the Nth reuse of this JVM.

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 queries that we used in our study. Note that in this experiment the systems we run are unmodified, unlike the ones we used in Section 3 that had to be instrumented. Therefore the unmodified systems' runtimes are slightly faster than the ones in Section 3.

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 10 shows Spark queries running on HotTub with nearly zero warm-up overhead.

Figure 11 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 warm by the end of the first run and require multiple iterations before reaching peak performance. Figure 11 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 3 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.

		Testing				
		q11	q14	q15	q09	q01
Training	q11	1.78	1.67	1.51	1.49	1.55
	q14	1.64	1.65	1.47	1.49	1.50
	q15	1.72	1.67	1.62	1.54	1.62
	q09	1.57	1.59	1.55	1.53	1.53
	q01	1.76	1.74	1.65	1.54	1.74

Table 4: Speed-up sensitivity to workload differences using the five fastest BigBench queries on Spark with 100GB data.

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 3 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.

6.2 Sensitivity to Difference in Workload

Table 4 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 $(mean - variance, mean + variance)$, 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.

6.3 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 $300\mu s$. 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.

7 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 Section 3.5. Our work distinguishes itself as it 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 [24, 47, 48, 59]. Gog *et al.* observe that objects often have clearly defined lifetimes in big data analytics systems [24]. Therefore they propose a region-based memory management [78] system where developers can annotate the objects with the same lifetime. Maas *et al.* [47, 48] 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 Taurs 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 [30], 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, 85].

Performance studies on data-parallel systems. A handful of works have thoroughly analyzed the performance of data-parallel systems [44, 46, 55, 57]. However they did not study the JVM performance impact. Ousterhout *et al.* comprehensively studied the performance of Spark [55], 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.* [57] compared Hadoop MapReduce with DBMS, and found that data shuffling is often the bottleneck for MapReduce. Jiang *et al.* [44] analyzed the performance of Hadoop MapReduce. McSherry *et al.* [50] 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 [82], where the JVM warm-up time is even more significant.

Other works on improving data-parallel systems performance focused on scheduling [2, 4, 31, 38, 56, 84], high performance interconnect [17, 19, 40, 45, 81], optimization for multi-cores [16, 49, 60], and removing redundant operations [61]. Our work is complementary as it focuses on JVM-level improvements.

8 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.

Acknowledgements

We greatly appreciate the insightful feedbacks from the anonymous reviewers and our shepherd Andrea Arpaci-Dusseau. We thank Yu Luo, Serhei Makarov, Michael Stumm, Jenny Ren, Kirk Rodrigues, Guangji Xu, Yongle Zhang, and Xu Zhao for the useful and thought stimulating discussions. We thank Yu Luo for setting up and maintaining the server cluster environment used in our experiments. His help was invaluable. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and an NSERC USRA award. Hailong Sun is supported by National Key Research and Development Program of China (2016YFB1000804) and National Natural Science Foundation of China (61370057).

References

[1] A case against static initializers. `http://sensualjava.blogspot.com/2008/12/case-against-static-initializers.html`.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, 2010.

[5] Android ART Just-In-Time (JIT) Compiler. `https://source.android.com/devices/tech/dalvik/jit-compiler.html`.

[6] Android runtime (ART). `https://source.android.com/devices/tech/dalvik/index.html`.

[7] Apache Geronimo DayTrader Benchmark. `http://geronimo.apache.org/GMOxDOC20/daytrader.html`.

[8] A. W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4).

[9] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, 2002.

[12] Big SQL 3.0: Hadoop-DS benchmark-Performance isn't everything. `https://developer.ibm.com/hadoop/blog/2014/12/02/big-sql-3-0-hadoop-ds-benchmark-performance-isnt-everything/`.

[13] M. K. A. B. V. Bittorf, T. Bobrovytsky, C. C. A. C. J. Erickson, M. G. D. Hecht, M. J. I. J. L. Kuff, D. K. A. Leblang, N. L. I. P. H. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

[14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, 2004.

[15] Cassandra. `http://cassandra.apache.org`.

[16] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[17] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, 2010.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[19] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, 2012.

[20] Excelsior JET - Java Virtual Machine (JVM) and Native Code Compiler. `https://www.excelsiorjet.com/`.

[21] Exciting performance improvements on the horizon for spark sql. `https://databricks.com/blog/2014/06/02/exciting-performance-improvements-on-the-horizon-for-spark-sql.html`.

[22] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, 2003.

[23] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[24] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand,

and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[25] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java®Virtual Machine specification - Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/.

[26] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, 1989.

[27] Hadoop. https://hadoop.apache.org.

[28] Hadoop Distributed File System (HDFS). http://hadoop.apache.org/docs/stable/hdfs_design.html.

[29] Hbase. http://hbase.apache.org/.

[30] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, 2005.

[31] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, 2011.

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

[33] U. Hölzle and D. Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, OOPSLA '94, 1994.

[34] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

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

[36] Impala – Cloudera. http://www.cloudera.com/content/www/en-us/products/apache-hadoop/impala.html.

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

[38] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[39] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, 2003.

[40] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

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

[42] 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.

[43] StackOverflow: Why do people still say Java is slow? http://programmers.stackexchange.com/questions/368/why-do-people-still-say-java-is-slow.

[44] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2).

[45] MapReduce-4049: Plugin for generic shuffle service. https://issues.apache.org/jira/browse/MAPREDUCE-4049.

[46] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4).

[47] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[48] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[49] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, Massachusetts Institute of Technology, 2010.

[50] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[51] Microsoft Common Language Runtime (CLR). https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx.

[52] Nailgun: Insanely fast Java. http://www.martiansoftware.com/nailgun/background.html.

[53] New benchmarks for sql-on-hadoop: Impala 1.4 widens the performance gap. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/.

[54] New benchmarks for sql-on-hadoop: Impala 1.4 widens the performance gap. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/.

[55] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, 2015.

[56] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[57] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, 2009.

[58] Performance comparison - c++/java/python/ruby/jython/jruby/groovy. http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/.

[59] Project tungsten: Bringing spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[60] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, 2007.

[61] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[62] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[63] J. H. Saltzer. Protection and the control of information sharing in Multics. *Commun. ACM*, 17(7).

[64] Spark. http://spark.apache.org.

[65] Spark will offer interactive querying of live data. https://www.linux.com/news/spark-20-will-offer-interactive-querying-live-data.

[66] Spark SQL performance test. https://github.com/databricks/spark-sql-perf.

[67] Specjbb2015. https://www.spec.org/jbb2015/.

[68] SPECjvm2008. https://www.spec.org/jvm2008/.

[69] StackOverflow: Is Java really slow? http://stackoverflow.com/questions/2163411/is-java-really-slow.

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

[71] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-time Compiler. *IBM Syst. J.*, 39(1).

[72] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4).

[73] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, 2001.

[74] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, 2003.

[75] Spark and Tez are successors of MapReduce. http://blogs.gartner.com/nick-heudecker/spark-tez-highlight-mapreduce-problems/.

[76] Tez. https://tez.apache.org/.

[77] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2).

[78] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[79] Transaction Processing Performance Council (TPC) Benchmark$^{\text{TM}}$DS (TPC-DS): The New Decision Support Benchmark Standard. http://www.tpc.org/tpcds.

[80] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014.

[81] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[82] What do real-life apache hadoop workloads look like? `http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/`.

[83] Why Java will always be slower than C++. `http://www.jelovic.com/articles/why_java_is_slow.htm`.

[84] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, 2008.

[85] B. Zorn. The measured cost of conservative garbage collection. *Software – Practice & Experience*, 23(7).