Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications

Martin Maas[⋆] Krste Asanović[⋆] Tim Harris[†] John Kubiatowicz[⋆]

[⋆] University of California, Berkeley [†] Oracle Labs, Cambridge

Abstract

Many distributed workloads in today's data centers are written in managed languages such as Java or Ruby. Examples include big data frameworks such as Hadoop, data stores such as Cassandra or applications such as the SOLR search engine. These workloads typically run across many independent language runtime systems on different nodes.

This setup represents a source of inefficiency, as these language runtime systems are unaware of each other. For example, they may perform Garbage Collection at times that are locally reasonable but not in a distributed setting.

We address these problems by introducing the concept of a *Holistic Runtime System* that makes runtime-level decisions for the entire distributed application rather than locally. We then present Taurus, a Holistic Runtime System prototype. Taurus is a JVM drop-in replacement, requires almost no configuration and can run unmodified off-the-shelf Java applications. Taurus enforces user-defined coordination policies and provides a DSL for writing these policies.

By applying Taurus to Garbage Collection, we demonstrate the potential of such a system and use it to explore coordination strategies for the runtime systems of real-world distributed applications, to improve application performance and address tail-latencies in latency-sensitive workloads.

1. Introduction

A large portion of workloads that are running in cloud data centers are written in managed languages such as C#, Go, Java, JavaScript/node.js, PHP/Hack, Python, Ruby or Scala. According to a recent survey [48], 4 out of the 6 most popular languages are managed languages, and many widely used cloud frameworks – such as Hadoop [66], Cassandra [1] or Spark [68] – are written in these languages.

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA
ACM 978-1-4503-4091-5/16/04.
http://dx.doi.org/10.1145/2872362.2872386

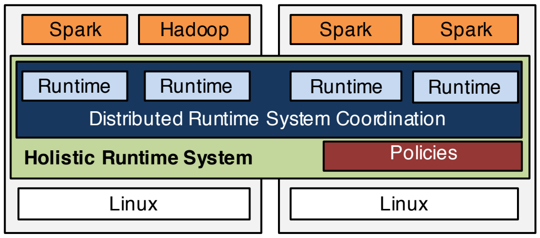


Figure 1: A Holistic Runtime System coordinates individual runtimes across machines based on a policy.

We see this trend continuing: Companies such as Twitter [17] or Facebook [8, 14] are writing most of their code in Scala and PHP/Hack. At the same time, cloud platforms such as Google AppEngine [7] or Microsoft Azure [13] are supporting managed language as explicit targets. Finally, many web startups write their code in languages such as Python, Ruby or JavaScript, as it allows them to iterate quickly.

Unfortunately, managed languages are known to cause performance overheads in some use cases [25, 38], and much work has been done to address these. However, most of this work looks at individual nodes. When running *distributed managed applications* and frameworks such as Hadoop or Spark at a rack or data-center scale, the application spans many processes across multiple nodes, each with its own runtime system. As we show in this paper, this causes problems beyond those encountered within an individual runtime system. Specifically, the language runtime systems on different nodes make completely independent decisions, such as about when to perform GC or how to JIT their code. This causes performance problems for many workloads, including both throughput-oriented and latency-sensitive jobs.

To address these problems, we propose what we call a *Holistic Runtime System* (Figure 1). A Holistic Runtime System is a distributed language runtime system that treats the runtimes underlying a distributed application as a distributed system itself and enables them to make decisions globally rather than individually. As such, it adopts some of the ideas from distributed operating systems [26, 44, 49, 53, 57] and applies them to language runtimes systems. This approach is motivated by today's low-latency data center networks and a trend towards rack-scale machines that enable a tighter integration of nodes in distributed systems. At the same time, data center workloads interact at ever smaller time-scales. These developments make it both feasible and necessary to integrate the runtime systems in a cluster more closely.

One design point would be a monolithic distributed runtime system with a single-system view [45]. However, this approach raises challenges with (1) compatibility with existing applications, (2) predictability by hiding the distinction between local and remote memory, (3) failure isolation, and (4) scalability bottlenecks such as those from distributed GC across a shared heap. We therefore propose an intermediate approach that retains the boundaries between individual runtime systems but enables coordination between them. Applications do not need to be aware that the runtime systems they are running across are part of the same logical entity. This enables running unmodified workloads, where the Holistic Runtime transparently applies policies for GC, compilation, etc. At the same time, workloads that know they are running on such a system can communicate with it to take advantage of its global knowledge and coordination capabilities.

To demonstrate the effectiveness of this approach, we built Taurus, a Holistic Language Runtime System for Java, with a specific focus on GC. Taurus is based on the OpenJDK Hotspot JVM and is a JVM drop-in replacement. It requires no modification of the application: once installed, running a Java application will automatically run it with Taurus, which joins it into the distributed system so that it can partake in global decision-making. Applications can then supply *policies* (written in a special DSL) to instruct Taurus how to coordinate language events (such as GC) between nodes.

We run two real-world workloads with Taurus. By using Taurus to coordinate GC, we show that it can reduce the execution time of a Spark PageRank workload by 21% and eliminate many GC-related stragglers for a Cassandra workload (reducing the 99.99%ile latency from 65.7 ms to 33.8 ms for reads, 40.7 ms to 10.1 ms for updates). At the same time, Taurus adds negligible overhead, scales to at least 180 nodes and is robust against failures (mostly isolating its faults from the applications and runtimes that it manages).

In this paper, we first make the case that the approach we take with the Holistic Runtime System – and Taurus in particular – is promising (Section 2). We then give an overview of the design of a Holistic Runtime System (Section 3), followed by a description of our policy DSL (Section 4) and the specific implementation details of Taurus (Section 5). We then evaluate Taurus on real-world workloads and microbenchmarks (Section 6). Finally, we present related work (Section 7) and draw our conclusions (Section 8).

2. Motivation & Case Studies

In this section, we make the case for using a Holistic Runtime System and why it is time to look at language runtime systems from a distributed systems perspective.

2.1 Opportunities of Managed Languages

The primary reason for using managed languages is arguably an increase in productivity, driven by features such as dynamic typing, class resolution at runtime, reflection and GC. The latter is particularly important: automatic memory management reduces the engineering overhead from managing explicit pointers and eliminates many sources of errors. Further advantages of managed languages include opportunities for dynamic optimization – data center workloads are often running for a long time and can therefore amortize JIT overheads. Many managed runtimes also compact memory during execution; this is important for long-running applications, since they can otherwise experience performance degradation from fragmentation and loss of locality.

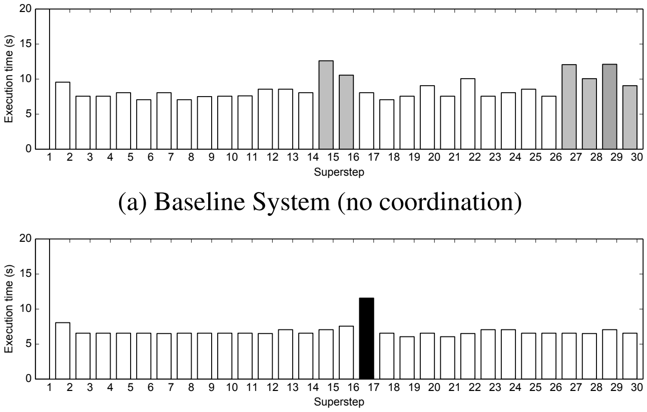
2.2 Problems for Distributed Applications

The advantages of managed languages often come at the cost of substantial overheads over native code. These overheads have been well-studied. The most significant challenge is often considered to be GC, with thousands of academic papers in this area. Other overheads include JIT compilation and profiling [25], type resolution [19], as well as overheads from boxed types, indirection and additional safety checks [38]. Many previous works address these challenges for individual runtime systems on a single machine, and tremendous progress has been made. However, when running distributed applications across multiple runtime systems, a largely orthogonal set of problems emerges. We divide these problems into four categories:

Lack of Coordination between Nodes. Runtime systems on different nodes are unaware of each other. This means that they make all decisions independently, e.g., when to perform GC. As we show in the next section, this can have a significant impact on distributed workloads, both batch workloads and interactive jobs. Similar problems have been reported in several real-world deployments [9, 15, 33].

Interference within Nodes. Runtime systems on the same node do not coordinate. While co-scheduling of data center workloads is well-investigated and has been addressed by many projects [27, 30, 35, 43], most of these projects do not look at managed-language-specific issues such as GC interference or instruction cache pollution from multiple copies of the same code. Some distributed Java applications have 100s of instances on the same node – this can make these overheads substantial, and motivated the Multi-tasking Virtual Machine project [41] and JSR-121 [11]. However, MVM never achieved widespread adoption, presumably due to concerns about failure propagation and difficulty of deployment (two issues we are addressing in this work).

Slow Elasticity. Managed language runtimes take a long time to boot and reach their full performance, partly due to warming up the code cache. This introduces overhead when adding processes at a fine granularity, as well as time skew.



(a) Baseline System (no coordination)

(b) Coordinating GC (Stop-the-Universe)

Figure 2: Relation between GC and the superstep durations of Spark PageRank (shade represents the number of nodes performing GC during a superstep; white = no GC). As soon as one node performs GC, the superstep takes much longer.

Redundancy. Every runtime system has its own JIT. As distributed workloads often run the same executable on every node (or even multiple instances of the same executable), this causes wasteful re-JITing of shared code. Companies such as Microsoft are therefore starting to forego the JIT in favor of statically pre-compiled binaries [3, 10]. However, this loses benefits from profile-directed dynamic optimization (e.g., trace-driven compilation and dynamic inlining).

2.3 Case Studies

We will now show how the first problem materializes in the two main categories of workloads encountered in data centers: throughput-oriented "batch" workloads such as big data computations (where we care about the completion time) and latency-sensitive "interactive" workloads (where we care about the distribution of response times).

2.3.1 Batch Workloads

We illustrate the problems caused by a lack of coordination between runtime systems for batch workloads, using iterative computations in Apache Spark [68] as an example. Spark is a popular Scala framework for distributed computation. It can run a wide range of workloads, including graph computations [34], machine learning, database workloads [67] and stream computations [69].

Many Spark workloads are very memory-intensive. Spark is based on applying operators and transformations to large distributed data sets; as a result, it exercises the heap significantly. Spark, by default, uses the parallel scavenge/serial mark-sweep-compact stop-the-world GC in the Hotspot JVM, which has very high collection performance but frequently incurs short pauses to perform young-generation GC (on the order of 100s of ms) and every once in a while incurs very long pauses (up to multiple seconds) for full GC.

These long pauses cause difficulties during iterative computations. Figure 2 shows an example of a 30-superstep PageRank computation – the example from the original Spark paper [68] – on a 16-node cluster (Section 6.2 describes the setup in detail). During each superstep, Spark launches a number of tasks on each worker node. Once all tasks have completed, the nodes exchange results and can only continue after this exchange has completed. As a result, this step effectively acts as a barrier between supersteps.

In the absence of GC, every superstep takes a similar time (Figure 2a). However, when one node performs a full GC, it pauses for multiple seconds and all other nodes have to wait at the barrier for that node to become available again. While waiting, they cannot do any work themselves. Worse, when they continue, they will at some point incur a GC as well, and become the reason for other nodes to wait.

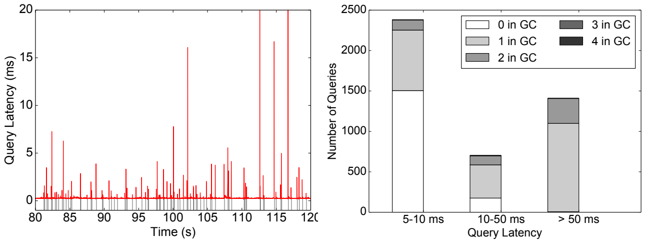
The root cause of this problem is that the runtime systems make independent decisions about when to perform GC. The memory on the different nodes fills up at a different rate, and therefore their GCs will occur at different times. But what if the system was globally coordinated? In that case, the best decision would be to let all nodes do GC at the same time: since nodes have to wait for a single node in GC, they can use that time efficiently by performing their own GC. We call this policy *Stop-the-Universe*. Figure 2b shows its effect: even on this relatively small workload, it leads to a speedup of 21% (excluding the initial loading of the graph from disk).

We note that this problem is common to all distributed iterative computations with global barriers. This includes many Machine Learning algorithms (e.g., Logistic Regression) and graph workloads (e.g., Shortest Path). The problem is somewhat reminiscent of inter-thread synchronization in parallel workloads and the OS Noise problem in HPC [63]. Both are often solved using gang-scheduling, and *Stop-the-Universe* can be seen as its language-runtime equivalent.

Similar problems have recently been confirmed for Naiad workloads [33]. This indicates that this class of problems applies to a wider range of workloads and runtime systems.

2.3.2 Interactive Workloads

We now demonstrate a different set of problems which is encountered by latency-sensitive workloads. This includes data stores such as Cassandra [1], client applications such as the SOLR search engine [60] or systems-level software such as ZooKeeper [39]. Data center applications are interacting at ever smaller time-scales (such as in algorithmic trading, real-time bidding for ads, or low-latency storage [52]). Further, applications are often composed of hundreds of services [29] and the expected latencies for individual services have decreased to micro-second granularities. In such a scenario, stragglers are a significant problem since a single straggler can cause an entire request to miss its deadline. GC can be a significant contributor to this problem – even minor-GC pauses at the order of milliseconds are problematic when services operate at micro-second granularity.



(a) Request latencies over time (b) Stragglers coinciding with GC

Figure 3: Correlation between stragglers in Cassandra read queries and GC. Grey lines in (a) are minor GCs and latencies of (potentially concurrent) request are averaged over 10ms intervals. The average request latency was 277 us.

We use the Cassandra key-value store as an example. Cassandra uses consistent hashing to replicate data across a subset of nodes. Requests can be sent to any node, which then assembles a quorum by contacting the replicas. While Cassandra uses a concurrent collector, it still experiences multi-millisecond pauses for young-generation GC (note that most concurrent collectors still introduce some form of jitter like this). To show the impact of these pauses, we ran a 4-node Cassandra cluster with the YCSB benchmark [28] for 10M queries (details can be found in Section 6.3). While the average latency for reads was 277 us, we occasionally incurred latencies that were over 100× larger. Figure 3a shows that these spikes mostly coincide with GC pauses (grey lines). Figure 3b shows that, in fact, most requests longer than 10 ms coincided with a GC pause, either in the node that received the request or in a node required to assemble a read or write quorum (concurrent work [31] presents related results). Note that these numbers might even understate the impact of GC, due to correlated omission in YCSB [65].

The fundamental problem is, once again, ignoring the language runtime system as part of the distributed system. Runtime systems make their GC decisions independently and collect as soon as their young generation fills up, stalling the application without warning. For example, two replicas holding the same entry can go down for GC at the same time, making it impossible to assemble a quorum. Further, services often have a choice where to send a request – for example, Cassandra requests can be handled by any node. However, when sending a request to a node that starts a GC pause before sending a response, we introduce a straggler.

Many of these problems could be avoided in a globally coordinated system. One approach is to expose the state of all runtime systems to the logic that directs requests to different servers (e.g., the load balancer). This makes it possible to avoid nodes that are close to GC – we call this *Request Steering*. Another strategy is to globally schedule GC such that there is always a sufficient number of replicas of any service available – we call this approach *Staggered GC*. Concurrent work [62] also confirms the efficiency of similar strategies.

2.4 What Does Software Do Today?

Problems such as the ones above have been reported for a wide range of applications and frameworks, including Hadoop [9], SOLR [60] and financial applications [4, 15]. They are often solved by rewriting parts of the application in a native language such as C++ and managing large data structures off-heap [16, 42]. Others use non-idiomatic Java (e.g., large byte arrays), split applications into smaller VMs or control allocation carefully to avoid GC pauses. In fact, the latest versions of Spark and Cassandra both use such techniques (note that our experiments use versions from before these changes). The problem with these approaches is that they lose many advantages of using a managed language in the first place, including safety and productivity.

There is also anecdotal evidence that some distributed applications treat GC as a failure mode like others, and restart the process when a full GC is necessary. The problem with this approach is that it is only viable if GC is rare. As we saw in Section 2.3, major GCs may occur every few minutes.

In the past, some applications also tried to control GC explicitly, using functions such as `System.gc`. This turned out insufficient, since it was difficult for application developers to make good decisions based on the knowledge available. Some runtime systems hence ignore such calls today.

We have also seen a commercial application that steers requests away from nodes paused for GC, similar to our proposed strategy above [15]. Implementation of such strategies is facilitated through language extensions such as C#'s GC Notification API [6] that allow applications to respond to upcoming GC pauses [62]. However, while explicitly designed for this use, we have not seen these APIs being widely used. Our hypothesis is that the mechanism is too low-level; programmers still need to solve distributed systems problems such as multi-node coordination or failures. Making the implementation of such strategies much easier is our key goal.

A final solution to the problem are concurrent garbage collectors such as C4 [61] or G1 [5]. These collectors avoid GC pauses; however, this comes at a performance cost from constantly having to trace and compact the heap, and performing extra book-keeping on every reference access to keep collector and mutator in sync (e.g., resulting in a large number of traps). This means that more memory bandwidth and CPU resources are used for GC to achieve the same GC throughput as a parallel stop-the-world collector. Furthermore, most concurrent GCs still introduce short pauses or jitter – with the very short request latencies required by many services today, this can still be problematic. In fact, we believe that it may be preferable to tolerate rare, predictable stop-the-world pauses over jitter from concurrent GC.

While all these solutions work, they often result in error-prone ad-hoc approaches, reduce productivity, redo a large amount of work, are not portable or yield poor performance. We believe it is time for a general solution to the problem, and propose Holistic Runtime Systems as such a solution.

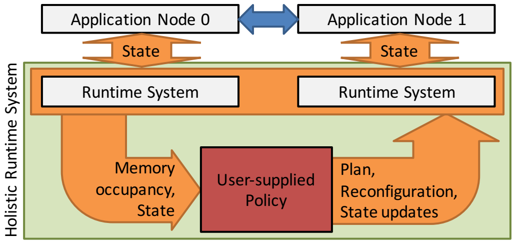


Figure 4: The Holistic Runtime System approach.

3. System Overview

We now give a high-level overview of the Holistic Runtime System approach and our implementation, Taurus. A Holistic Runtime System is a distributed language runtime system that coordinates a set of language runtimes across a cluster. It aims to be a general solution that enables developers to deploy and experiment with strategies to work around the problems from Section 2.3 while abstracting away many sources of errors and increasing productivity.

From the perspective of the application that is running across the runtime systems, nothing changes: the application processes are still isolated, and there is no shared heap. Traditionally, each runtime system would now make decisions independently, based on its settings. For example, the Hotspot JVM allows users to configure the GC, generation sizes, tenure rates, and many others. In a Holistic Runtime, these decisions are made globally for the entire distributed workload, through a configurable *policy* provided by the application. We show how policies are defined in Section 4.

Coordination works by dividing time into epochs of varying length. At the end of each epoch, a *leader* executes the policy. The policy considers the state of all the nodes and produces a *plan* that contains runtime events to be executed during the next epoch, as well as information to be shared between the nodes. This plan is then distributed to the other nodes and executed in a decentralized manner (reminiscent of the approach taken in systems such as Tesselation [27] for decentralized scheduling). At the end of the epoch, all nodes report back to the leader with any state updates, so that the next epoch can begin (Figure 4).

Holistic Runtimes enable a wide range of coordination patterns, while abstracting away the challenges of maintaining the distributed system, failure tolerance, time synchronization and interfacing to the runtime system. In Taurus, our prototype, we use these mechanisms to implement and investigate GC coordination. However, the same mechanisms could be used to coordinate JITs, share profiling data, reduce startup times or co-tune applications on the same node.

3.1 Design Decisions and Goals for Taurus

One of our primary goals for Taurus is compatibility. While it would be possible to design an entirely new system (and potentially language) from the ground up, it would be challenging to bring up workloads for it: many existing workloads require a fully standard-compliant runtime system, and even close approximations (such as Apache Harmony [2]) do not work reliably. We therefore decided to base our work on the OpenJDK Hotspot JVM, which is the reference implementation of Java and runs the vast majority of software.

Another important goal is usability. For Taurus to be useful, its deployment must be substantially easier than reimplementing coordination at the application level. We therefore designed Taurus to be a drop-in replacement; the goal is that the user only has to install a different `java` executable and everything else behaves the exact same way as before. Behind the scenes, this executable calls into Hotspot and brings up Taurus, which then connects the runtime systems.

A final goal is failure isolation. For the system to be adopted, it must also not substantially increase the probability of failure in any part of the system. For this reason, Taurus has fault tolerance built in and can tolerate node-failures by electing leaders and migrating state using a distributed consensus protocol (Section 5.3). This is an important argument for foregoing a model that supports a single-system view (which might reduce failure isolation between nodes).

We are also concerned about failure propagation into the JVM. We therefore avoid direct changes to the Hotspot code base but instead interact with the JVM through its management interface. Hotspot provides a rich interface to install libraries and performance monitors within the JVM, which can communicate to the outside world. Taurus itself is implemented as a co-process for Hotspot – this ensures that most errors in Taurus are staying outside the JVM process barrier, and failures do not bring down the JVM.

3.2 Components of the System

Figure 5 shows the high-level components of Taurus. With Taurus installed, every JVM instance is augmented with a *monitor* process at startup, to form a *Holistic Runtime Instance*. The monitor connects to the JVM's management interface, which allows it to monitor memory occupancy and other internals, and trigger JVM operations (Section 5.1). The monitor also opens a communication channel to the application space of the JVM. This allows the application to exchange information with Taurus (Section 3.4). Using this feature is optional and requires modifying the application.

On startup, the monitor instantiates a *client*, which exposes an RPC interface that other nodes can connect to (Section 5.2). Monitors also connect to a consensus layer that provides us with a set of replicated, consistent storage. This layer is used for features such as leader election or node discovery (Section 5.3). We use an implementation of the Raft consensus protocol [51] – we assume that this layer is available on startup, but it could also be launched automatically.

When launching a Holistic Runtime Instance, the application can select a policy, usually through a new set of special command line flags (we use the `-XX:HVM:flag=value` namespace, which is not used by Hotspot; adding new `-XX` arguments does not break J2SE-compliance).

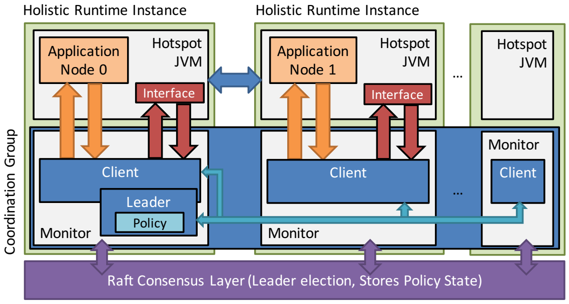


Figure 5: The high-level components of Taurus.

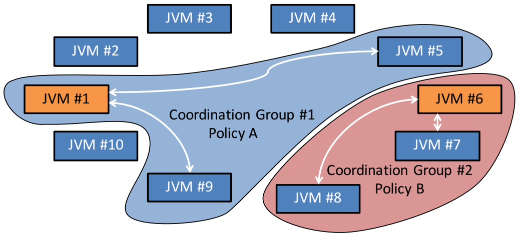


Figure 6: Cluster divided into multiple coordination groups. Orange nodes are the leaders, which execute the policy and distribute the results to the remaining nodes in the group.

3.3 Policy Execution

Policies are the core of Taurus: they describe the strategies for coordinating the different runtime systems. Taurus's policies are written in a high-level DSL we describe in Section 4.

Policy execution in Taurus follows a two-level approach. Using the consensus layer for all coordination in the entire cluster would result in scalability issues, and prevent us from fully exploiting fast rack-level interconnects. Instead, Taurus allows multiple policies to be active within the same cluster and operate on what we call a *coordination group*.

A coordination group is a subset of runtime systems that is subject to a policy at a given time (Figure 6). Every runtime system can be a member of at most one coordination group, and policies can choose to add or remove unclaimed runtimes from their group. Coordination groups operate independently from each other and the consensus layer is only used to handle group membership, leader election and recovery (we assume such changes to be infrequent). This means that a Holistic Runtime System can manage a large set of machines while fine-grained coordination occurs for subsets of nodes, such as those within a rack or an application.

A policy is a pure function that takes as input the state of the coordination group (e.g., memory occupancy of all runtime systems, or user-defined state as described in Section 3.4) and produces a plan that contains runtime events for the next epoch and state updates. It also contains coordination group changes (e.g., runtime systems to add or remove).

Each coordination group has an elected leader; all other nodes are followers. The leader establishes a synchronized time base for the group and is responsible for executing the policy once every epoch. When a runtime system instance is launched with a policy selected, it will try to become the leader for this policy by contacting the consensus layer. If there is no leader yet, the node will become the leader, spawn off a *leader* thread, and start executing the policy. Otherwise, it will become a *follower* and connect to the existing leader.

By using this approach, we address the scalability issues since no global consensus is necessary within the coordination group; the only scalability bottleneck is the leader, which needs to receive one sample from each member of the group per epoch and distribute the plan (in Section 6.4.3, we show that this scales well to at least 180 nodes). Furthermore, failures in the coordination group are "softer" than failures in the global consensus layer. While they may lead to performance degradation, missing a few epochs will not cause a failure of the application running on Taurus – they can be fixed up throughout later epochs (work in cluster scheduling shows that such an *optimistic* approach can work well in a distributed setting [58]). The downside is that since we are not relying on the consensus layer within coordination groups, we need to be able to recover from both leader and follower failures (Section 5.4).

3.4 Communication with the Application

In addition to coordination at the runtime system level, Taurus can optionally communicate with the application itself. This is useful to implement policies such as the *Steering* policy from Section 2.3.2, where the runtime system needs to communicate to the application which nodes to avoid. The abstraction we chose is a shared set of key-value pairs visible to both the runtime system and the application space.

For this purpose, each monitor installs a globally visible class in the application space that provides thread-safe static methods `HVM.getKeyValuePair(k)` and `HVM.setKeyValuePair(k,v)`. To use them in an application, it suffices to add a .jar file to the classpath.

The executing policy receives as part of its input the full set of key-value pairs on all nodes of the coordination group, and can produce (as part of its plan) updates to key-value pairs on any node. This allows policies to implement a wide range of communication patterns with the application.

3.5 Reconfiguration

When launching a new policy, its coordination group only contains one node. The policy can query the list of available *runtimes* in the cluster, and whether they are a member of any other group. They can then select non-claimed nodes to add to their coordination group as part of the plan. These nodes join during the next epoch, and are considered during the next execution of the policy (conflicts are avoided by making policy executions atomic with respect to each other). Node terminations or failures are visible in a similar way: the policy can see a node that has timed out and can remove it from its coordination group (Section 5.4).

```
policy AutoAdd {
  run(Set<Runtime> s = runtimes) {
    foreach(Runtime r : s) {
      if (r.status == UNASSIGNED) {
        plan <— ReconfigureAddMember(r);
      }}}}
```

(a) Policy that automatically adds unassigned runtimes.

```
policy STU {
  extern double cutoff = 10.0;

  run(Set<Member> stu = members,
      Set<Member> collect = members) {
    if (!(stu.filter([Member m : (! m.busy) &&
        m.memory.old > cutoff]).empty())) {
      plan <— MajorGC(collect);
    }}}
```

(b) *Stop-the-Universe* Policy that performs a full GC for all members in `collect` if at least one of `stu` has reached a memory occupancy of `cutoff`.

```
policy Example {
  import STU(cutoff=90.0);
  import AutoAdd();

  run() {
    STU(members.filter([Member m:
        m.runtime.tag == "gc"]), members);
    AutoAdd(runtimes);
    plan <— EpochLength(200.0); //ms
  }}
```

(c) Composing policies: `Example` imports and calls into `STU` (with two dynamic parameters), then `AutoAdd`.

Figure 7: Examples of policies written in our DSL.

4. Policy Description Language

To facilitate the development of policies (and therefore adoption), we designed a DSL for policy descriptions (Figure 7).

Each policy declaration (which gives the policy a unique name) contains a `run` block that describes the policy function and defines policy parameters. This block contains a sequential program that can access any state visible to the system and builds up a plan. However, instead of a fully Turing-complete language, we do not provide general for or while loops but only `foreach` loops over finite sets; this ensures that policies *always terminate*, which prevents the leader getting stuck (a key advantage of using a DSL).

In addition to the primitive types `double`, `int` and `string` and two parameterized collection types `Set` and `Map`, the language provides two composite types to describe runtime systems (two types are required to distinguish between runtime systems that are under the control of the policy and those registered but not under the policy's control):

```
policy PingPong {
  state Map<Member,int> prev = 0;

  run() {
    foreach (Member m : members) {
      int pp = m.kv("pingpong").toInt(0);
      if (prev.get(m) != pp) {
        plan <— SetKV(m, "pingpong", pp+1);
        prev.set(m, pp+1);
      }
    }
    plan <— EpochLength(200.0);
  }}
```

Figure 8: Example policy using key-value pairs and policy state. The policy monitors a key-value pair and increases it when it sees a change; the application does the same.

- `Runtime`: A runtime that is part of the Holistic Runtime but not necessarily the current coordination group. This type is used for group management (e.g., adding/removing). It contains fields such as the runtime's server, command line, tags set at startup, and group membership.
- `Member`: A member of the policy's coordination group. This type is used to actually interact with the runtime system. It contains fields for memory occupancy of the different subspaces, GC statistics, whether the node is unresponsive (`busy`) and KV pairs.

Note that the `Members` are a subset of the `Runtimes`; given a `Member m`, the corresponding `Runtime` can be accessed with `m.runtime`. The sets of all runtimes and all members can be accessed using global keywords `runtimes` and `members`. Note that the `runtimes` set can become large, which is why monitors cache it instead of updating it from the consensus layer every epoch (comparing only a version number to check for updates).

The language allows filtering sets with a predicate. An example can be found in Figure 7b. In this case, a filter predicate is used to determine the set of members with a certain memory occupancy, and check whether it is empty.

We provide a special construct of the form `plan <- Action(...)` to add commands to the policy's plan. A plan is effectively the policy's return value and contains a set of commands to execute on each runtime system. Some commands take parameters, such as a subset of members or runtimes they apply to (e.g., adding a set of runtimes, performing GC on a set of members). By repeatedly using this construct, the policy creates the plan for the next epoch.

A member's key-value pairs are accessed through a `kv` field in `Member`, and updates to them are added to the plan (Figure 8). Key-value pairs are stored as strings and it is often necessary to convert values to or from primitive types such as integers. To prevent error conditions in the case of malformed strings, all such conversions need to be provided a default value that is used in case the conversion fails.

Finally, policies support state that is kept around between epochs using the `state` keyword (Figure 8). Note that all state variables also need a default value that is used in case of errors (e.g., if a key is not found in a Map, or on failure).

4.1 Configurability and Composability

We hypothesize that most workloads will require variations of a small set of basic policies, potentially with some application-specific extensions (Section 6.1). Composability is therefore an important feature in our Policy DSL: it allows us to build a repository of basic policies over time, and combine them into application-specific solutions. To achieve this flexibility, policies need to be configurable and composable.

We allow policies to be included into other policies through an `import` statement. This will include the policy and allows it to be called within the `run` block. Figure 7c shows an example of this. Policies are parametrizable with two types of parameters: dynamic and static parameters. Static parameters are defined at the time of import and do not change at runtime (these are the parameters defined as "extern" outside the run block). Dynamic parameters are given to the policy whenever it is called from within another policy. Figure 7b shows examples for both types of parameters: `cutoff` is static, while `stu` is dynamic. All parameters can have default values.

4.2 Policy Compilation

Our DSL is embedded into C++11, and we reuse many C++ features including numerical and logical operations, if statements and stream operators. A recursive-descent parser written in Python transforms our policy code into C++ code, which is then compiled into a dynamic library. The parser does not split the code down to individual tokens but only into pieces that can be directly transformed into C++ (e.g., the predicate within a filter). We then find and replace any DSL-specific keywords and idioms with their C++ equivalents (considering scopes, strings, etc.). Policies are transformed into classes, foreach loops into iterators and filter predicates into C++11 lambda expressions. We use C++11 move semantics to chain filters without copying data.

5. Implementation

After the high-level design, we will now present implementation details of Taurus. Taurus is entirely written in C++11, to avoid GC-induced pauses in the system itself.

5.1 JVM Interface

The monitor connects to Hotspot through three different interfaces (Figure 9). It queries memory occupancy information through `jstat`, which exposes the JVM's performance counters by writing them to a shared page that can be mapped by a different process (by default, Hotspot updates these counters every 50ms; as we require a finer granularity, we set the interval to 1ms instead). This means that the monitor can access this data without blocking on the JVM. Commands in the JVM (primarily triggering of Major GC) are performed through the `jcmd` interface, which allows calling into the JVM to trigger activities (it may stall if the JVM is unresponsive, such as in a GC pause). Finally, key-value pairs are exposed to the Java application through a *Java agent* that is installed into the JVM's application space at start-up. This agent connects to the monitor through an IPC mechanism provided by the ZMQ library [37], which is the same library we use for inter-node communication. The agent then updates key-value pairs in the `HVM` class (Section 3.4), which is accessible from the application. The agent is also responsible for triggering minor GCs: since Hotspot does not support this, the agent can force a minor GC by allocating unreachable objects until the young generation is full (using the `MemoryPoolMXBean` interface to determine how much memory it needs to fill).

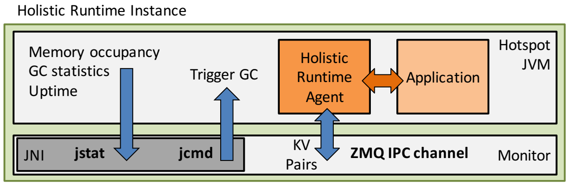


Figure 9: Interface between Hotspot and the monitor.

5.2 Inter-node Communication

Inter-node communication is implemented using a simple RPC protocol. We use ZMQ [37] for communication, since it gives us a higher level of abstraction than regular sockets (e.g., managing concurrency and high-level communication patterns), and supports low-latency communication over Infiniband. Our RPC protocol is using Protocol Buffers [64].

Adding a node is simple: The leader (prompted by its policy) sends a reconfigure request to the node's client, which will then send a request to join the leader's coordination group. The leader then confirms the join request and sends the plan of the currently active epoch to the client, after which the node is part of the group. Leader and client also repeatedly exchange timestamps through a separate connection, to determine the drift between them (no special hardware is needed for this). From then on, all timestamps are expressed relative to the leaders's clock.

5.3 Consensus Layer

We use LogCabin [12, 51], Stanford's Raft implementation, as our consensus layer. We run three LogCabin instances, and runtime systems can connect to any of them. LogCabin provides a small amount of consistent, highly replicated storage. We use this storage to track the set of instances currently registered with Taurus, as well as active policies. The set of instances has a version number; for performance, nodes cache the instances locally and only compare against the version number once per epoch.

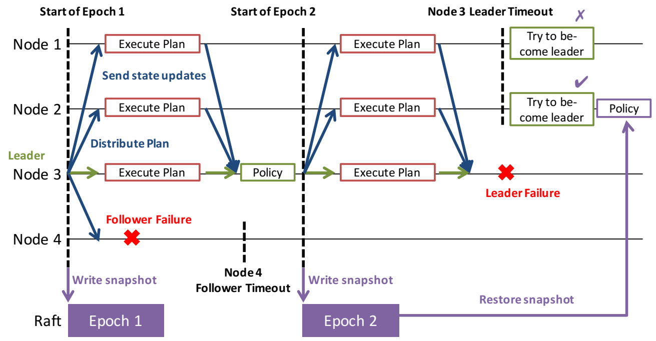


Figure 10: A sample policy execution in Taurus. When a follower fails, the policy sees it as unavailable. When the leader fails, another node becomes leader, restores the policy from the last executed epoch, and continues (marking all other nodes as unavailable for the next epoch, before stabilizing).

When launching a new JVM, its monitor will connect to LogCabin and create an entry for its runtime system instance. This includes information such as its command line options and address/port (the monitor selects a free port automatically on startup). Next, it will try to launch a policy and become the leader for it, if requested through a command line option. When the policy runs, it can detect other instances in the cluster and add them to its coordination group. (To help policies distinguish between multiple distributed applications, it is possible to supply a flag with a unique application ID to all JVMs belonging to one application; the first node with this ID will become the leader and runs the policy, which adds the other nodes.)

5.4 Execution & Failure Handling

Once a leader has brought up the policy, it enters a loop that consists of three stages, visualized in Figure 10:

1. Distribute the current plan to all nodes in the coordination group (at the beginning, the plan is empty).
2. Followers execute the instructions in the plan, then atomically take a sample of their state, and send it to the leader. If they cannot take a sample in time, they will send a response that they are "busy" (which can happen if the JVM is in a GC pause or if there is contention in the system).
3. The leader collects state updates from all followers. Once the epoch has ended, it marks all nodes from which it has not received an update as "unavailable", executes the policy, produces a new plan, and sends it to all nodes (including unavailable ones; failure is discussed below).

The length of the epoch is set by the plan itself and can adapt to the circumstances (e.g., if the workload is exhibiting irregular allocation rates, the policy can decide to decrease the epoch to react more quickly to changes). However, in this case policy authors have to be careful about control loops.

Leader failures are tolerated by storing the policy metadata (including the last plan) to the consensus layer every epoch (i.e., the epoch is made persistent before sending out a plan). All followers have a timeout by which they expect the next plan to arrive. If no plan arrives, they assume that the leader has failed and will attempt to become leader themselves by trying to write the entry of the current epoch. If it has been written before, it means that the original leader is either still alive and the message was delayed, or that some other node has become leader in its place. In that case, the node will continue as a follower. If the node succeeds in writing the policy entry, it becomes the next leader, pulls the policy data from the consensus layer, and starts executing.

This approach moves all the failure handling into the consensus layer, simplifying fault tolerance. At the same time, it puts little load on the consensus layer in the absence of failures (requiring only the leader to access it, once per epoch). This maintains the advantages of the two-level approach.

6. Evaluation

We now demonstrate how Taurus coordinates GC in real-world applications and characterize its coordination performance and overheads through microbenchmarks.

We performed most of our evaluation on a 16-node cluster connected with 40 GbE using Mellanox dual port MCX314A-BCBT cards. Each node has an Intel IvyBridge E5-1680V2 3.0GHz CPU with 8 cores (16 hardware threads) and 64GB RAM. In addition to our workloads, the cluster ran YARN, as well as HDFS and Tachyon file systems. To show generality and scalability of the system, we also run a scalability microbenchmark on a set of 200 `g1-small` instances on Google Compute Engine.

All nodes are running Linux 3.13.0. We run a snapshot of LogCabin from 4/10/15 with the `Segmented` storage module on a RAM disk (as we do not require persistence across machine failures). We use OpenJDK 1.7.0_75, ZeroMQ 4.0.5 and Google Protocol Buffers 2.6.1. Unless noted otherwise, we used the default settings for all applications.

6.1 GC Policies & Applying Taurus to Applications

To make efficient use of Taurus, it is important to apply policies that address the specific GC problems of a given application. Looking at a range of workloads, we discovered that most GC policies we found could be expressed as a combination of three fundamental strategies:

- *Schedule*: Trigger GC at suitable/convenient times.
- *Redirect*: If a task can be handled by multiple nodes, choose a node that is not unavailable due to GC.
- *Hand-off*: If a node has sole responsibility for a task or a piece of data, hand it to another node before GC.

We dub these fundamental strategies *S*, *R* and *H*. Looking at published literature and our own work, we can describe a wide range of policies in terms of these strategies:

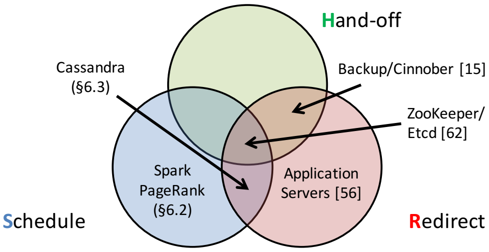


Figure 11: Mapping of GC policies to base strategies.

1. **Stop-the-Universe (S).** Synchronize GCs between all nodes in the distributed system. This strategy is useful for iterative batch workloads (Section 2.3.1).
2. **GC-when-Idle (S).** Trigger GC during a period of idleness. E.g., for in-memory computations, perform GC in the backing data store (e.g., HDFS) during computation.
3. **Steering (R).** In a system where multiple servers can handle requests (e.g., sharded search engines such as SOLR [60], or replicated services), steer requests away from nodes close to GC. Steering can happen within applications (e.g., picking replicas) or at a load-balancer.
4. **Staggering (S).** In a system that requires quorums of replicas for consistency, ensure that only one of the replicas is performing GC at any given time.
5. **Backup (SR).** Have two instances of a component, but use only one of them and trigger GC on the other [15].
6. **Leadership Transfer (H).** When the leader of a system is about to stall for GC, use the system's recovery mechanism to hand off leadership to another replica [62].
7. **Timeout Extension.** Stop connections or leases from timing out during GC pauses (a common problem [18]).

Applications can combine these policies to fit their needs. Figure 11 shows examples: Spark (Section 6.2) benefits from *Stop-the-Universe* (*S*), Cassandra (Section 6.3) uses *Steering* & *Staggering* (*SR*), a Zookeeper-like system [62] has been shown to benefit from a *Steering*, *Staggering* & *Leadership Transfer* (SRH), and replicated application servers [56] have been shown to benefit from *Steering* (*R*) alone.

We note that there is a connection between the fundamental policies and their implementation complexity: *S* oftentimes requires no changes to the application, *R* does require changes but those are minimal if it is possible to modify the replica selection algorithm already available in applications, while *H* can be more work, unless the application already has a hand-off mechanism as part of its failover handling.

Taurus enables and simplifies the implementation of these policies, and we hypothesize that they provide a basic set that can be combined to cover most applications. Using the two examples from Section 2.3, we now show how Taurus can be used to improve these workloads by implementing three of the fundamental policies above.

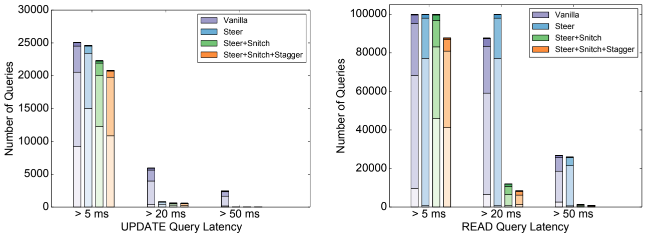


Figure 12: Cumulative latency distributions of the slowest 100K queries (99.9 percentile) of a representative run. Stacked bars represent the number of GCs during a request.

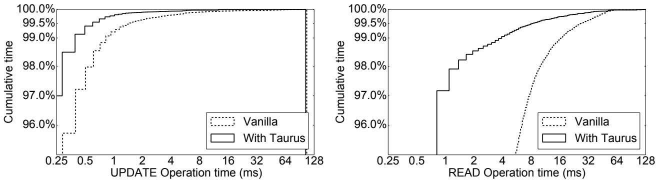


Figure 13: Fraction of total execution time (sum of all query latencies) spent in requests of at least a certain length.

6.2 Case Study: Apache Spark

As an example for a batch workload, we used the Spark PageRank workload from Section 2.3.1. As discussed there, this workload benefits from a Stop-the-Universe (STU) policy, and implementing it in Taurus is simple. In fact, all parts have already appeared in the paper: the improved PageRank workload in Figure 2b is the result of using the STU policy from Figure 7. We used Spark 1.1.1 on 16 nodes with 32GB heaps. No modification to Spark was required, and even this simple policy reduced execution time by 21%.

6.3 Case Study: Apache Cassandra

As an example for an interactive workload, we improve Cassandra tail-latencies (Section 2.3.2). We run a YCSB snapshot from 3/11/15 against Cassandra 1.0.6 with a replication factor of 3 and a cluster size of 8. We run workload A on a keyspace with 10M entries for 100M queries (50s warmup). We chose a 32GB heap with a 4GB young generation.

The average latencies were 173.6 us for updates and 522.8 us for reads, but some requests were over 100× slower: Figure 12 shows the distribution of the slowest 100K queries (>5ms latency), which is the tail from the 99.9 percentile. Our goal was to remove as many of the stragglers (>20ms) as possible. To this end, we applied multiple of the GC coordination policies from Section 6.1:

Request Steering (Steer). Requests in Cassandra can be sent to any node. A main source of stragglers are nodes that stall for GC while processing a request. To avoid this, we expose all nodes that are close to GC to the load balancer (in our case the YCSB client). We implemented a policy that monitors the young-generation occupancy of each node every 10ms epoch and maintains a key-value pair per node. When a node's occupancy reaches 80%, the key-value pair is set to 1. YCSB then checks these key-value pairs before dispatching each request. If the key-value pair is 1, YCSB sends the request to a different node (we hence maintain additional connections to each Cassandra node). This eliminates most update stragglers over 20ms but does not improve reads (Figure 12). This is because updates can be delayed while reads need to contact a quorum of replicas.

Snitch Steering (Snitch). In order to serve a read query, the node executing the query has to assemble a quorum of replicas. This quorum is chosen based on a *Snitch*, a feature in Cassandra that is normally used to describe the data center topology. We modified Cassandra's dynamic snitch to read the key-value pairs and select replicas that are not close to GC. Figure 12 shows this improves reads substantially.

Staggering (Stagger). Additional read stragglers stem from multiple GCs occurring at the same time. To address this, we modified the policy to ensure that only one node is performing GC at any time. Each epoch, we trigger a minor GC on the node with the highest memory occupancy over 80%, if any, and trigger no other GC before it has finished. For our 8-node cluster, this is sufficient. Larger clusters could stagger GC by triggering simultaneous GCs on nodes 3 steps apart in the Cassandra ring (to avoid stalling multiple replicas belonging to the same key).

The overall impact of all coordination techniques is shown in Figure 13. The y axis shows the fraction of total time spent in requests of at least the latency on the x axis (averaged over 10ms intervals). As our coordinated version is well to the left of the original (note the log scale), we eliminate a large fraction of stragglers. On a per-request basis, the 99.99%ile latency improves from 65.7 ms to 33.8 ms for reads (40.7 ms to 10.1 ms for updates) and the 99.999%ile from 128.6 ms to 54.6 ms for reads (67.7 ms to 21.0 ms for updates).

Note that we use an untuned configuration of Cassandra, with a large heap and young generation. In practice, Cassandra deployments are heavily hand-tuned. We argue that with a system such as Taurus, tuning becomes less important.

6.4 Microbenchmarks

We now characterize the coordination performance, scalability and overheads of Taurus using microbenchmarks.

6.4.1 Coordination Granularity

We are first interested in the granularity of coordination that Taurus enables, specifically the minimum sustainable epoch length. We ran Taurus on a Java program that sleeps for one second at a time in an infinite loop, and ran it for different policies and epoch lengths (Figure 14). We report how often clients fail to report back by the end of the epoch; this indicates that the epoch was too short. In all cases, we let the system reach a stable state before performing our measurements. Error bars here and later are the σ of 5 runs.

8-Null and 16-Null are a policy that does nothing and only checks for new runtimes (on 8 and 16 nodes). In both cases, no epochs were missed until going below 2ms, indicating the minimum sustainable epoch length is 1-2ms (for the scale we looked at). Since the minimum jstat sampling rate of the Hotspot JVM is 1ms, this is sufficient.

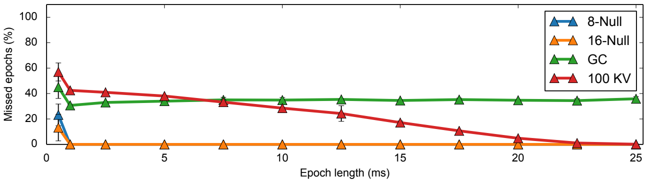


Figure 14: Missed epochs depending on the epoch length.

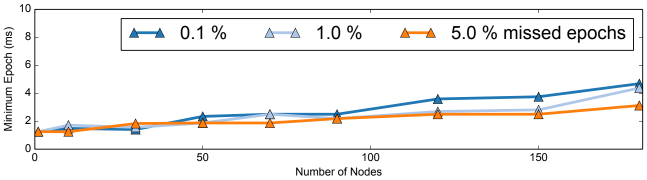


Figure 15: Scaling to 180 Google Compute Engine nodes.

We also experimented with two other policies: `KV` sets 100 key-value pairs every epoch on each node – this stresses communication and increases the minimum epoch length to 20ms. We also ran a policy that triggers a full GC every 10s (`GC`), on a workload that always allocates data. As expected, this misses a constant fraction of epochs due to GC pauses.

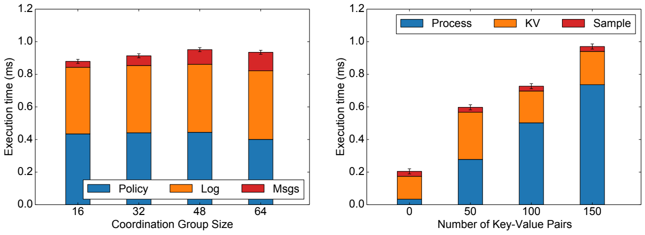
6.4.2 Scalability

To demonstrate that Taurus scales to a large number of nodes in a more realistic data center deployment, we ran the same benchmark on a set of up to 180 Google Compute Engine `g1-small` instances. We performed a run with a policy that constantly measures the number of missed epochs and adjusts their length in a binary search fashion until a target fraction of missed epochs is reached (note that in a large deployment, there are always going to be some missed epochs).

Figure 15 shows that even in such a deployment without Infiniband, Taurus performs well and can achieve epoch times below 10ms. With ping latencies of around 300us (and LogCabin running on a separate set of nodes), we believe these results to be reasonable. As we saw in Section 6.3, a 10ms epoch is sufficient even for fine-grained coordination in latency-sensitive systems. In addition to helping us determine the minimum epoch length, this experiment also shows an example of a policy that automatically adjusts the epoch length (a good strategy to make policies more portable).

6.4.3 Performance Breakdown

We are now interested in how different components of the execution contribute to the epoch length. Figure 16 shows the different contributors. For the leader, we ran a policy that reads out all nodes' GC statistics while varying the coordination group size (with four JVMs per node). For the followers, we chose the KV policy from Section 6.4.1, varying the number of key-value pairs.



(a) Leader Execution (b) Follower Execution

Figure 16: Impact and scalability of the different components of policy execution, for both the leader and followers.

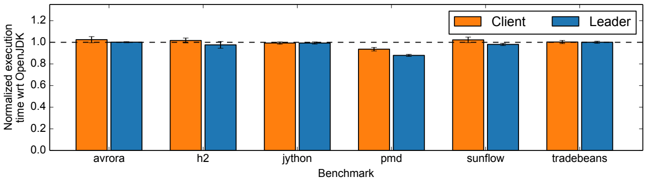


Figure 17: Overhead of running Dacapo with Taurus.

The leader's execution limits the coordination granularity as much as the epoch length. The bottleneck appears to be the access to LogCabin: snapshotting the epoch takes about 400us on average (with a LogCabin instance on the leader's node). The policy takes a similar amount, as it needs to perform one access to the LogCabin cluster as well, in order to check version numbers. Sending out plans is a small fraction, and scales linearly to the coordination group size.

For followers, the execution time is currently dominated by receiving, decoding and applying the instructions of the plan (which grows with the number of key-value pairs). It is likely that a better plan format could reduce these overheads. Neither sampling the JVM nor communicating Key-Value pairs to the JVM appear to be a substantial bottleneck.

6.4.4 Overhead

To determine the overhead of Taurus, we used the Dacapo benchmarks [22] (we chose the subset that worked correctly with OpenJDK 7). We compare the performance degradation both for followers and leaders. Taurus does not introduce substantial overheads, at most 3.0% (Figure 17). Given that JVMs are known to be sensitive to changes in the environment and that we introduce many such changes (e.g., adding an agent, using management interfaces, changing the sampling interval), we believe most of the differences in performance to be noise (e.g., the speed-up for pmd).

7. Related Work

There has been a large amount of work on GC – Jones and Lins [40] provide a comprehensive survey. Recently, there has been renewed interest in GC in the Big Data setting [31, 33, 45, 46, 54]. Proposed solutions range from approaches similar to our request steering [15, 31, 55, 56, 62] to region-based memory management for avoiding GC [33, 50], to specialized collectors [32, 61]. Our approach is different in that we provide a general mechanism to implement many of these approaches, instead of a new approach in itself.

The Holistic Runtime System has similarities to previous work on Distributed JVMs [20, 21, 47, 70, 71]. However, these JVMs are targeted at monolithic applications, not distributed workloads. There are also projects that share some of our goals: Forseti [24] investigates holistic heap sizing. The MVM [41] looked at running multiple applications in the same JVM. A^2-VM [59] cooperatively schedules Java applications across machines, making the JVM and its services resource-aware to enable cluster-wide thread scheduling based on policies (which bear some resemblance to Taurus's policies). Finally, Terracotta [23] deploys Java applications across JVMs through clustering. The last two differ from Taurus in that they provide a platform for writing distributed workloads rather than a transparent support layer.

Taurus also has similarities with work on cluster schedulers [36, 58]. While they schedule and coordinate workloads as well, they do so at a much coarser granularity. However, it is possible that a system such as Taurus could eventually be integrated into the cluster scheduler, to reuse its available information and failover mechanisms.

8. Future Work and Conclusion

We presented the design of a Holistic Runtime System to coordinate distributed applications in data centers. We introduced Taurus, a prototype of such a system that can be used to coordinate GC. Taurus is a JVM drop-in replacement, runs unmodified real-world applications, requires no modifications to the underlying runtime system and provides a simple DSL to implement policies. Our goal is to enable developers to implement their own policies and use Taurus as a research vehicle for exploring coordination strategies.

We believe that Taurus can be used beyond GC. Specifically, it could be used for distributed monitoring and profiling, coordinating code generation and reducing interference between JVMs. It would also be possible to integrate Taurus with other runtime systems than the JVM, to coordinate workloads across different programming languages.

Acknowledgements

We would like to thank Michael Armbrust, Peter Kessler, Kay Ousterhout, Philip Reames, Mario Wolczko, Reynold Xin and the anonymous reviewers for their feedback on this and earlier versions of this work. Thanks are owed to Peter Bailis for pointing out the snitch mechanism in Cassandra, and to Daniel Goodman for coining the term "Stop-the-Universe".

Precursor work was done at Oracle Labs, Cambridge. Research at UC Berkeley was partially funded by DARPA Award Number HR0011-12-2-0016, DOE grant #DE-AC02-05CH11231, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Martin Maas, the first author, is dedicating this paper to his newlywed wife Yucy. "Taurus" is her zodiac sign.

References

[1] “The Apache Cassandra Project.” [Online]. Available: http://cassandra.apache.org/

[2] “Apache Harmony.” [Online]. Available: http://harmony.apache.org/

[3] “ART vs Dalvik - introducing the new Android runtime in KitKat.” [Online]. Available: http://www.infinum.co/the-capsized-eight/articles/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat

[4] “Credit Suisse Case Study.” [Online]. Available: http://www.azulsystems.com/customers/creditsuisse

[5] “G1: One Garbage Collector To Rule Them All.” [Online]. Available: http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

[6] “Garbage Collection Notifications.” [Online]. Available: https://msdn.microsoft.com/en-us/library/cc713687(v=vs.110).aspx

[7] “Google App Engine: Platform as a Service.”

[8] “Hack: a new programming language for HHVM.” [Online]. Available: https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/

[9] “HDFS Issue 7244: "Reduce Namenode memory using Flyweight pattern".” [Online]. Available: https://issues.apache.org/jira/browse/HDFS-7244

[10] “Inside .NET Native (Channel 9).” [Online]. Available: http://channel9.msdn.com/Shows/Going+Deep/Inside-NET-Native

[11] “JSR-000121 Application Isolation API Specification.” [Online]. Available: https://jcp.org/aboutJava/communityprocess/final/jsr121/

[12] “LogCabin (GitHub).” [Online]. Available: http://github.com/logcabin/logcabin

[13] “Microsoft Windows Azure.” [Online]. Available: http://www.windowsazure.com/

[14] “On Garbage Collection.” [Online]. Available: http://hhvm.com/blog/431/on-garbage-collection

[15] “Predictable Low Latency: "Cinnober on GC pause-free Java applications through orchestrated memory management",” Tech. Rep. [Online]. Available: http://www.cinnober.com/sites/cinnober.com/files/news/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf

[16] “Project Tungsten: Bringing Spark Closer to Bare Metal.” [Online]. Available: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[17] “Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers.” [Online]. Available: http://www.infoq.com/articles/twitter-java-use

[18] “ZooKeeper SessionExpired events,” in *Apache HBase Reference Guide*. Apache HBase Team. [Online]. Available: http://hbase.apache.org/book.html

[19] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, “Checked Load: Architectural Support for JavaScript Type-checking on Mobile Processors,” in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[20] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill, “Kaffemik - a Distributed JVM Featuring a Single Address Space Architecture,” in *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium*, 2001.

[21] Y. Aridor, M. Factor, and A. Teperman, “cJVM: A single system image of a JVM on a cluster,” in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999.

[22] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The DaCapo Benchmarks: Java Benchmarking Development and Analysis,” in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2006.

[23] J. Bonér and E. Kuleshov, “Clustering the Java Virtual Machine using Aspect-Oriented Programming,” in *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[24] C. Cameron, J. Singer, and D. Vengerov, “The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes,” in *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, 2015.

[25] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, “The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[26] D. Cheriton, “The V Distributed System,” *Commun. ACM*, vol. 31, no. 3, pp. 314–333, Mar. 1988.

[27] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz, “Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous

Adaptation," in *Proceedings of the 50th Annual Design Automation Conference*, 2013.

[28] 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*, 2010.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[30] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[31] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong, "Understanding the Causes of Consistency Anomalies in Apache Cassandra," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, 2015.

[32] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[33] I. Gog, J. Giceva, M. Schwarzkopf, K. Viswani, D. Vytiniotis, G. Ramalingan, M. Costa, D. Murray, S. Hand, and M. Isard, "Broom: sweeping out Garbage Collection from Big Data systems," in *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems (HotOS 2015)*, 2015.

[34] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[35] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling Parallel Runtime Systems," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[36] 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*, 2011.

[37] P. Hintjens, "ZeroMQ: The Guide," Tech. Rep., 2010. [Online]. Available: http://zguide.zeromq.org/page:all

[38] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the Software Stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007.

[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.

[40] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sep. 1996.

[41] M. Jordan, L. Daynès, G. Czajkowski, M. Jarzab, and C. Bryce, "Scaling J2EE Application Servers with the Multi-tasking Virtual Machine," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2004.

[42] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder, "Impala: A modern, open-source SQL engine for hadoop," in *Seventh Biennial Conference on Innovative Data Systems Research*, 2015.

[43] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[44] E. D. Lazowska, H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal, "The Architecture of the Eden System," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 1981.

[45] M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz, "The Case for the Holistic Language Runtime System," in *First International Workshop on Rack-scale Computing (WRSC '14)*, 2014.

[46] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz, "Trash Day: Coordinating Garbage Collection in Distributed Systems," in *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems (HotOS 2015)*, 2015.

[47] M. Maas and R. McIlroy, "A JVM for the Barrelfish Operating System," in *2nd Workshop on Systems for Future Multi-core Architectures (SFMA '12)*, 2012.

[48] L. A. Meyerovich and A. S. Rabkin, "Empirical Analysis of Programming Language Adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[49] S. Mullender, G. van Rossum, A. Tananbaum, R. van Renesse, and H. van Staveren, "Amoeba: a distributed

operating system for the 1990s," *Computer*, vol. 23, no. 5, pp. 44–53, May 1990.

[50] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[51] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[52] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[53] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *Computer*, vol. 21, no. 2, pp. 23–36, Feb. 1988.

[54] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[55] A. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, "Adaptive GC-Aware Load Balancing Strategy for High-Assurance Java Distributed Systems," in *16th International Symposium on High Assurance Systems Engineering (HASE)*, 2015.

[56] A. O. Portillo-Domínguez, M. Wang, D. Magoni, P. Perry, and J. Murphy, "Load balancing of Java applications by forecasting garbage collections," 2014.

[57] M. Schwarzkopf, M. P. Grosvenor, and S. Hand, "New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, 2013.

[58] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th European Conference on Computer Systems*, 2013.

[59] J. Simão, J. Lemos, and L. Veiga, "A2-VM : A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling," in *On the Move to Meaningful Internet Systems: OTM 2011*, ser. Lecture Notes in Computer Science, 2011.

[60] D. Smiley and D. E. Pugh, *Apache Solr 3 Enterprise Search Server*. Packt Publishing Ltd, 2011.

[61] G. Tene, B. Iyengar, and M. Wolf, "C4: The Continuously Concurrent Compacting Collector," in *Proceedings of the International Symposium on Memory Management*, 2011.

[62] D. Terei and A. Levy, "Blade: A Data Center Garbage Collector," *arXiv:1504.02578 [cs]*, Apr. 2015, arXiv: 1504.02578.

[63] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications," in *Proceedings of the 19th Annual International Conference on Supercomputing*, 2005.

[64] K. Varda, *Protocol buffers: Google's data interchange format*, 2008.

[65] N. Wakart, "Correcting YCSB's Coordinated Omission problem," Mar. 2015. [Online]. Available: http://psy-lob-saw.blogspot.com/2015/03/fixing-ycsb-coordinated-omission.html

[66] T. White, *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[67] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and Rich Analytics at Scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[68] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[69] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, 2012.

[70] W. Zhu, C.-L. Wang, and F. Lau, "JESSICA2: a distributed Java Virtual Machine with transparent thread migration support," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[71] J. N. Zigman and R. Sankaranarayana, "dJVM-A distributed JVM on a Cluster," Australian National University, Tech. Rep., 2002.