Notification and Prediction of Heap Management Pauses in Managed Languages for Latency Stable Systems

Daniar H. Kurniawan*, Cesar A. Stuardo*, Ray Andrew O. S.†, Haryadi S. Gunawi*

*University of Chicago †Emmerich Research Center

Abstract

An increasing number of high-performance distributed systems are executed on top of runtime environments like Java Runtime Environment (JRE), such as Cassandra [1], Hadoop [2], Spark [3], Hazelcast [4], Alluxio [5], Hive [6], and Re- thinkDB [7]. The developers are attracted to a managed language like Java as it offers services like Garbage Collection (GC), run-time type checking, reference checking, and cross-platform execution. However, many applications running on the JVM (e.g., big data frameworks such as Hadoop, data stores such as Cassandra) suffer from long garbage collection (GC) time. The long pause time due to Stop-The-World (STW) during Garbage-Collection (GC) not only degrades application throughput and causes long latency, but also hurts overall system efficiency and scalability.

To address these problems, we implement MITMEM that provides JVM support to cut millisecond level tail latencies induced by GC. MITMEM is a JVM drop-in replacement, requires no configuration and can run off-the-shelf Java applications with a minimum modification (e.g., adding 120 LOC to integrate with Cassandra). Our experiments indicate that the MITMEM-powered Cassandra successfully reduces the tail latency up to 99%.

1 Introduction

Since Google published the seminal “The Tail at Scale” article in 2013 [8], tail latency has grown to be an important concept and research topic [9, 10]. Likewise in the storage world, many varieties of products including hyper converged storage, key-value stores and databases, are advertised not only with traditional metrics such as throughput and average latency, but tail latency as well (e.g. Xms latency guaranteed at the Yth percentile) [11–14]. At the same time, an increasing number of high-performance distributed systems are executed on top of runtime environments like Java Runtime Environment (JRE), such as Cassandra [1], Hadoop [2], Spark [3], Hazelcast [4], Alluxio [5], and Hive [6]. The developers are attracted to a managed language like Java as it offers services like Garbage Collection (GC), run-time type checking, reference checking, and cross-platform execution. Moreover, the automated memory management enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. With all of those advantages, developers can build robust distributed systems with a simpler design that focusing on the performance, availability, and scalability.

IDC (International Data Corporation) predicts that the global datasphere will grow from 45 Zettabytes in 2019 to 175 Zettabytes by 2025 [15]. In all likelihood, those java-based systems will keep growing and fast evolving to keep up with the increasing demand for more efficient and better (scalability, availability, and performance) storage systems. Therefore, the role of Java Virtual Machine (JVM) is not only very important in our cloud ecosystems, but also becoming very critical. One issue of JVM that attracts most attention from the research community is its Garbage Collection (GC) problem. Although Java Memory Management is regarded as one of the language’s finest achievements, it is also one of the most crucial components that poses an open challenge to design a better automated memory management [16]. Over the course of GC evolution, there have been various GC algorithms being developed and perfected such as G1GC [17], ParallelGC [18], Shenandoah [19], ZGC [20]. The main difference among those algorithm will be explained in Background (Section §2). Most GC algorithms are employing Stop-The-World (STW) pauses, and they are facing the same performance issues as shown in many studies [21–23].

The fact that GC’s Stop-The-World (Figure 1) must pause mutators (threads that modify objects in heap), the GC duration contributes to a non-trivial portion of application execution time which can take up to one-third of the total execution time of an application [22]; and it can seriously injured the application performance as experienced by LinkedIn [24] and Instagram [25] which lead to poor scalability on multi-core systems with large memory. It can even account for half of the processing time in memory-intensive big data systems [21, 26]. Moreover, the exceedingly long garbage collection time hurts system throughput and incurs unpredictable and severely degraded tail latency in interactive services [27, 28].

There are various techniques proposed to reduce GC overhead [21, 26, 29], but such techniques will increase the burden on developers by requiring them to add additional logic (annotation) to their code base. Moreover, integrating it with the existing complex system forces the developer to rewrite the entire code base which negates much of the benefit of
using managed languages. The garbage collection impact is often unpredictable because the production workload can’t be easily simulated during testing. The system itself are also evolving by adapting new techniques and more features causing uncertain behavior at the runtime. As a result, the users must monitor the JVM’s performance continuously and adjust the GC parameters periodically as the workload and systems are continuously evolving.

To address these problems, we propose a fast-rejecting GC-aware mechanism called MITMEM. The MITMEM-powered JVM can be integrated with any systems without requiring modification to the existing code base. MITMEM proactively uses JVM’s detailed knowledge about its resources usage to avoid long tail latency induced by GC. When the JVM informs the application about a long service latency, applications can better manage impacts on the tail latencies. Depending on their Service-Level Agreement (SLA), the applications can choose not to wait and perform an instant failover to another replica. MITMEM enables the application to further utilize the nature of modern distributed systems that always put replication on its core design.

To this end, we introduce MITMEM, a JVM that employs a fast-rejecting GC-aware interface to support millisecond tail tolerance. MITMEM is based on the OpenJDK8 and is a JVM drop-in replacement. We materialize this concept within the JVM Hotspot module, primarily because GC are the major resource of contention for Java application. In a nutshell, MITMEM provides a GC-aware reply interface which will help the application avoid a long tail latency induced by GC. The biggest challenge in supporting a fast rejecting interface is the development of an always-running thread (we call this thread as SpecialThread) that will send rejection to the clients during STW pauses. Implementing the SpecialThread requires understanding the nature of object allocation and the underlying signal communication between Operating System (OS) and JVM. Furthermore, the SpecialThread needs to escape the safepoint check and be able to access its special heap during GC. Finally, the overhead should be negligible and the SpecialThread must be robust when dealing with multiple client connections.

To examine MITMEM can benefit applications, we study data-parallel storage such as distributed NoSQL systems. Examination show that many NoSQL systems do not adopt tail-tolerance mechanisms §2.1, and thus can benefit from MITMEM support. Evaluated on a 6 nodes cluster with synthetic workload, we show that MITMEM-powered Cassandra can cut the tail latency up to 99%. At the same time, MITMEM adds negligible overhead, and is robust against multiple client connections. In summary, our contributions are: design and working examples of MITMEM, and demonstration that MITMEM-powered Cassandra can leverage a fast-rejection mechanism to achieve significant latency reductions. We close with discussion, related work, and conclusion.

2 Motivation and Background

We first describe the lack of tail-tolerance mechanism in popular NoSQL systems. Second, we provide a brief background on garbage collection, garbage collectors, and ParallelGC. Next, we provide a concrete example of tail latencies in Cassandra due to garbage collection. And then, we review state-of-the-art solutions in the last decade that include the evolution of garbage collection algorithms and tail-tolerance mechanisms.

2.1 Lack of Tail-Tolerance Mechanism

In this section we want to highlight that not all NoSQL systems have sufficient tail-tolerance mechanisms. Based on our prior study [30], six popular NoSQL systems (Cassandra, Couchbase, HBase, MongoDB, Riak, and Voldemort) does not failover from the busy replica to the less-busy ones when there are I/O contentions for one second. As one of the most popular Java-based NoSQL systems [31, 32], Cassandra is severely impacted by JVM’s GC [24, 25]. Although Cassandra employs snitching, it is not effective with 1-second rotating burstiness. From the same study [30], Cassandra by default has a very high time out value, 12 seconds. Thus, an IO can stall for a long time without being retried which are the cause of tail latencies.

2.2 Garbage Collection

Java’s automatic memory management feature is handled by the garbage collection process. When a Java program runs, it creates many objects on the heap throughout its running time. The garbage collector will dispose any unused objects and delete them to free up memory. Two of the most widely adopted techniques used by various GC algorithms are Stop-The-World (STW) approach (implemented on ParallelGC, CMS, and G1GC) and concurrent approach (implemented on CMS, G1GC, ZGC, and Shenandoah). The thread that runs GC algorithm can be called collectors; the one that implements STW approach is called stop-the-world collector, while the concurrent collector is for the threads that uses concurrent approach. One GC algorithm could make use of both techniques as depicted by Table 1. The details of the state-of-the-art GC algorithm will be explained in §2.5.

STOP-THE-WORLD Stop-the-world approach is the oldest and simplest technique [22]. It also delivers the highest throughput compared to other approaches. As depicted by Figure 1, stop-the-world works by first completely stopping the mutators (threads that modify the object in heap, i.e. application threads or user program). Then starting from a root set of pointers (registers, stacks, local variables), it traces all live objects (the object currently used by the program). Each GC algorithm could have a different process/phase during the stop-the-world period. Generally, the algorithm will mark the live object and relocate them to reduce heap fragmentation. Once the process completed, the application can
Table 1: Summary of JVM’s garbage collectors design. Red-shades represents stop-the-world collectors; while green-shades represents concurrent collectors. ZGC and Shenandoah are released under experimental-gc category.

<table>
<thead>
<tr>
<th>Collector</th>
<th>Young Generation</th>
<th>Old Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Copy</td>
<td>Mark</td>
</tr>
<tr>
<td>CMS</td>
<td>Copy</td>
<td>Compact</td>
</tr>
<tr>
<td>G1</td>
<td>Conc Mark</td>
<td>Conc Sweep</td>
</tr>
<tr>
<td>ZGC</td>
<td>Conc Mark</td>
<td>Conc Compact</td>
</tr>
<tr>
<td>Shenandoah</td>
<td>Conc Mark</td>
<td>Conc Compact</td>
</tr>
</tbody>
</table>

be resumed and the mutators can allocate a new object to the freed spaces. The simplicity of its design makes STW approach popular, and it has been adopted in various managed languages such as Go, Ruby, and Oracle’s JVM (by default). However, since stop-the-world pauses the application for doing any task, it contributes to significant pause times that are proportional to the number of live pointers and the size of the heap. As a result, state-of-the-art GC algorithms that employ STW can have pause times of 10–40ms per GB of heap [22]. This long pauses are notorious for causing tail latency at Java-based systems [24, 25].

**Concurrent** This technique is fully adopted in the JVM’s newest GC algorithms such as Shenandoah [19] and ZGC [20]. Implementing concurrent approach is the straightforward solution to reduce the pause time caused by STW collectors because it enables the GC processes/phases to run concurrently with application threads. The concurrent collector is using techniques such as read and write barriers to detect and fix concurrent modifications to the heap while tracing live data and/or relocating objects. These techniques minimize the pause times significantly compared to the stop-the-world approach [16]. However, concurrent garbage collectors have lower-throughput, higher implementation complexity, and edge cases that still require GC pauses. Oracle’s JVM, for example, has combined concurrent collectors with parallel collectors in G1GC, but the pause times is higher than ParallelGC (which only uses parallel collectors) [16]. This because G1GC could cause too much STW when allocating a humongous object. In the other case, Oracle also developed two experimental GC algorithms, Shenandoah [19] and ZGC [20], that fully utilize concurrent approach and they almost have no pauses [16]. The drawback is that they have higher complexity than parallel collectors, especially when doing reference checking. Generally, the concurrent collectors introduces more load to the processing unit by implementing a smarter algorithm on each GC’s phase. This project will not cover any CPU-related contention, and our solution will focus on solving the tail latency problem induced by stop-the-world pauses.

Figure 1: ParallelGC Phases. There are three phases: init phase (VM thread), parallel phase (GC threads), and final phase (VM thread). These phases are executed within Stop-The-World barrier, and mutators (i.e. application threads) are getting paused during that period.

2.3 ParallelGC

As depicted by Figure 1, ParallelGC collection involves three phases: initialization phase, parallel phase, and final synchronization phase [22]. In the initialization phase, VM thread suspends all mutators (i.e. application threads) before waking up the GC threads. After the GC threads become live, the VM thread sleeps and waits for the final phase. Collection is performed in the parallel phase, in which the GCTaskManager creates and adds GC tasks into the GCTaskQueue from where multiple GC threads can fetch and execute them in parallel. The global task queue enables ParallelGC to have dynamic task assignment among GC threads. ParallelGC is also regarded as generational garbage collection since it divides the heap into multiple generations: young, old, and permanent generation. The young generation is further divided into one eden space and two survivor spaces (from-space and to-space). When the eden space is filled up, a MinorGC is performed. Referenced objects in eden and from-survivor space are moved to the to-survivor space, and unreferenced objects are discarded. After a MinorGC, the eden and the from space are cleared, and all the objects that survived in the to-space will have their age incremented. After surviving a predefined number of MinorGCs, objects are promoted to the old generation.

After a few cycles of MinorGC, the old generation will be full and then MajorGC is triggered to free up the old generation space. Both minor and MajorGCs obtain tasks from GCTaskQueue except that GCTaskManager prepares different GC tasks for them. Among GC tasks, steal task happen very often and it is placed in GCTaskQueue after normal GC tasks. Steal task is needed as an attempt to balance the load between GC threads GC threads that have fetched steal tasks will try to steal work from other GC threads. When all GC threads complete the parallel phase and suspend themselves, the VM thread is woken up, entering the final synchroniza-
2.4 Tail Induced by GC

Let us consider the following scenario. When the application receives the request, and if the code is written in a managed language such as Java, the request is usually converted from byte stream to an object (e.g. “R = new Request(bytes)”) which can stall if the runtime is garbage-collecting the heap [21–23, 27, 33, 34]. If not, the application can continue to process the request. Figure 2 shows the requests being stalled by Java garbage collection (GC) roughly 8% of the time due to other bulk requests that (de)allocate memory intensively. Given this observation, it is hard to guarantee extreme stable latencies in Java-based storage systems.

2.5 State of the Art

The last decade has witnessed many novel solutions proposed to tame the tail latency problem, which we classify into four general categories: GC algorithm advancement, application-level modification, speculation, and replica selection (Table 2). We review their pros and cons in four axes: “application simplicity” implies no intrusive changes to the application; “efficiency” denotes no extra load (i.e. no speculative backup requests); “reactivity” means rapid reaction to latency perturbation in millisecond windows; and finally “flexibility” indicates the adaptability of the systems under various workloads (data size and throughput) changes.

**GC-algorithm advancement** happens rarely because of the complexity. As depicted by Table 1, JVM offered various GC algorithm to choose, each has its own design choices and caveats. For example, CMS suffers from heap fragmentation problem; G1GC performs bad when dealing with big objects allocation, it also has total pause longer than ParallelGC under the same workload; ZGC and Shenandoah add more load to the processing unit as it employs complex reference checking than other algorithms.

**APPLICATION-LEVEL MODIFICATION** re-architects tail-prone applications with a better computation and data management. For example, numerous key-value storage designs have been proposed for reducing contention between user and management operations or for handling workload skew and cache inefficiencies [35–44]. As shown in Table 2, while it is efficient and reactive and does not change resource-level policies, it requires application redesign and does not cover contentions outside the application.

**SPECULATIVE EXECUTION** treats the underlying system as unchangeable and simply sends a backup request (speculative retry) after some short amount of time has elapsed [45–47]. Many user-level storages adopt speculation for its simplicity and end-to-end coverage, but it causes extra load (i.e. speculative retry after waiting for the \( P \)-th-percentile latency will lead to \( 100−P \)% backup requests [8]).

**REPLICA SELECTION** predicts ahead of time which replicas can serve requests faster, often done in a black-box way (ease of adoption) without knowing what is happening inside the resource layers [48–50]. This requires detailed latency monitoring and expensive prediction computation for increasing accuracy. Most of the time, the prediction is only refreshed sparsely (e.g. every few minutes) [51, 52]. As a result, it is not reactive to bursty contention that can (dis)appear in sub-second intervals.

### Table 2: State of the art (§2.5 and §3.4)

<table>
<thead>
<tr>
<th></th>
<th>App-Simplicity</th>
<th>Efficiency</th>
<th>Reactivity</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC advancement</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>App. modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculation</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MitMEM</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3 MitMEM

MitMEM ("Mit" stands for mitigating and "Mem" stands for memory) is a Java Virtual Machine (JVM) extension with request cancellation and delay prediction mechanisms pertaining to garbage collection (GC) pauses. JVM GC is a runtime activity that frees objects no longer referenced. As objects in the virtual address space are being reshuffled, all application threads must be paused (aka. "Stop the World" GC). During this process, requests cannot be served, hence causing long tail latencies. This affects many Java-based user-level storage in a way that requires manual tuning such as in Cas-
3.1 Cancellation Mechanism

We describe several methods that we tried from a naive to a more robust one. Our first try was modifying the Java I/O library; within the network I/O path, Java library makes a read()-like system call to the OS, which perhaps can be wrapped around with request cancellation, but because this call is done within the context of application threads, it will be paused by safepoints [56]. Safepoint is a well-defined internal state inside the JVM. When all the threads are stopped at safepoints, the JVM can safely reschedule the heap and stack such that the threads’ view of the memory remains consistent when they leave the safepoint.

As cancellation via application thread is impossible, another method we attempted was creating a runtime-level thread similar to GC threads that can continue running even when other threads are paused. A possibility is to find the runtime-level networking stack and perform cancellation there. However, we found that runtime threads have to park as well when GC is running (because they also touch the same virtual address area being shuffled). If we unpark runtime threads, the OS will send a SIGSEGV signal and the runtime would crash. This intricate complexity of a real runtime such as JVM has been reported several times, such that some work prefers a clean-slate but partial runtime rather than extending a full-fledged JVM [21]. Fortunately, our failed experiences above led us to a way around the complexity.

We create a new runtime thread that does not have references to the part of the address space being reshuffled (outside the Java heap that contains application objects including URequest and SocketImpl). Our thread must know the sender’s file descriptor for sending cancellation notices, but because this information resides in the Java heap (inside SocketImpl), we modify the JVM to copy relevant information thatMITMEM needs and put them outside the Java heap.

Another job that this thread must perform is to read an arriving request from the OS and check its deadline. Hence, even when GC is running, our thread makes the reqrecv() call and puts the request in a temporary buffer. If the request should be cancelled (the remaining GC pause is longer than the remaining deadline), it is deleted (so that the application does not have to process it) and then a cancellation notice is sent back. Otherwise, the request is forwarded to the application; specifically, after GC unfreezes all the threads, our thread similar to GC threads that can continue running even when other threads are paused.

The main advantages of using MITMEM instead of tuning the garbage collector or manually handling the GC are that MITMEM is straightforward to use as it requires no specific configuration; and MITMEM is easy to integrate with any target system (e.g., MITMEM-powered Cassandra only needs 120 LOC). MITMEM provides custom JVM interfaces to the run-time system (RTS) that enable tail-tolerance support in data-parallel applications. Since MITMEM fast-rejecting mechanism is implemented inside JVM itself as a native extension, the overhead is negligible. MITMEM is not a new approach to garbage collection, but a new approach to dealing with its performance impact in distributed systems.

While the literature in this area mostly focuses on optimizing GC [21–23, 27, 33, 34], MITMEM’s principle is different: “let GC stops the world, but do not stop the universe.” That is, MITMEM’s method can work with any stop-the-world GC algorithms because its main functionality is cancelling paused requests such that the higher-level distributed storage can continue retrying the requests somewhere else (i.e. do not stop “the universe”). Below we describe our solution to two main challenges: how the runtime can send cancellation notification when all applications and many runtime threads are paused and how to predict GC pause delays.

3.2 GC Pause Prediction

JVM provides three GC algorithms, Parallel GC [18], G1GC [17], and ZGC [20]. Albeit the implementation differences such as the heap structure, levels of concurrency, and use of compaction to reduce fragmentation, all of them run GC in two phases: the mark phase, responsible for traversing...
through all the application objects and marking them as alive or dead, and sweep phase, responsible for reclaiming space used by dead objects. Furthermore, GC algorithms define a set of “GC roots” used as starting points of the mark phase. Each root branches to a graph of objects that will be traversed by the GC workers. Live objects will be promoted to another heap region (e.g. from new-generation to survivor area) and the unreferenced ones marked as dead. The former involves copying of live objects from one region into another while the latter executes complex operations such as compacting/defragmenting certain areas of the heap.

Regardless of the implementation detail, we model GC execution time as a linear relationship to the number of live objects in the object graphs. Others have modeled GC in a linear way as well [57] but they model when/how often GC will take place, while we model how long every GC will pause. Because copying is the main bottleneck, our linear model is:

$$T_{gc} = \frac{N_{liveobj} \times T_{copy}}{N_{gw}} + T_{ovh}$$

Here, $T_{gc}$ is the predicted delay, $N_{liveobj}$ is the number of live objects, $T_{copy}$ is the average copy time per object, $N_{gw}$ is the number of GC workers, and $T_{ovh}$ is an additional constant overhead. As MitMEM has visibility on $N_{gw}$ (a constant configuration value) and $N_{liveobj}$ (after the fast initial traversal), we only need to profile the values of $T_{copy}$ and $T_{ovh}$, which are dependent on the memory speed and other environmental factors. We tried several linear modeling algorithms such as RANSAC and OLS and found that RANSAC leads to the highest precision in our benchmarks. It successfully models $T_{copy}$, which depends on object sizes and memory copy speed, and $T_{ovh}$, which depends on some constant overhead, e.g. finding live/dead objects in mostly-static GC roots such as ClassLoader, System Dictionary, JNI handles and Management Beans objects graphs that might fluctuate in the beginning but will remain stable as the application runs for some time.

MitMEM’s GC prediction is also not devoid of imprecision. As often pointed out [58], GC execution time is hard to precisely estimate due to specific implementation details. For example, to predict Parallel GC (that is optimized for throughput), the imprecision lies in the parallelism complexity. Particularly for the mark phase, the whole process is divided into parallel workers, each picks a job (object) from a queue, analyzes the object, traverses down the graph, and adds the newly traversed objects to the worker queues. As the queues are unbalanced, task stealing can occur, which complicates prediction because stealing involves complex dependencies and synchronization operations.

Despite this imprecision, luckily MitMEM does not have to consider a wide range of application behaviors. MitMEM only needs to predict GC delays within the target application.

3.3 Design

As mentioned in the introduction, it is fundamentally challenging to maintain all the benefits of the aforementioned techniques if we attempt to solve the tail latency problem only entirely in the application. With the MitMEM, both application and runtime layer work hand in hand. Figure 4 illustrates MitMEM design, showing how a request flows from client to runtime server. To achieve all the five goals in Table 2, MitMEM consists of five essential elements.

A) AN END-TO-END REQUEST ABSTRACTION (Figure 4a) enables request abstraction on the runtime layer as what can be seen by application. Currently, operating systems, runtime and libraries are often oblivious to the end-to-end request context. They operate on abstractions such as streams, packets and functions that are hard to map to the notion of “user request” and its latency sensitivity. With the prevalence of interactive services, user request should be a first-class citizen. request acts as a unifying abstraction for scattered resource layers.

B) DEADLINE AWARENESS (Figure 4b) is added to the resource layers through which request flows. The importance of deadline information has been highlighted many times [59–63] and deadline-aware resources have been proposed such as in the TCP or block layer [30, 64, 65] but usually the awareness is only contained within the layer. With request, user’s deadline information can be simply added and automatically forwarded to runtime layers. The request structure also contains the send time. We also envision a scenario where client and server machines live in the same data center (e.g. for data locality); data-center clock synchronization has been solved to nanosecond level [66].

C) REQUEST CANCELLATION (Figure 4c) is now supported in the runtime layer. That is, when replicas are available, user-level storage can tag requests as cancellable such that when the deadline cannot be met by runtime layer, the
layer is given the liberty to cancel the request. The decision is made based on the “\texttt{remDL}” (the time that has elapsed since the request was sent) and “\texttt{predDelay}”. The concept of cancellable tasks or requests is common in real-time community \cite{67–69}, but has not been fully deployed in many standard systems. Likewise, request cancellation is an essential mechanism that will notify applications that a particular resource in the datapath is currently contended. Google also promoted the efficiency of cancellable requests \cite{8}, but the article only described cancellation that is done in a user-level file system, while in our work, we advocate the runtime layer to be aware of deadlines and capable of cancelling requests.

D) \textbf{RESOURCE DELAY PREDICTION} (Figure 4d) is now built in the runtime layer for making a decision to serve or cancel every arriving request. The decision is made based on the request deadline and the current contention level (JVM’s GC). The prediction ideally must be precise. Fortunately, a high precision is possible because the predictor is built inside the resource layer, hence has the full view of the contention inside the resource (\texttt{MitMEM} is capable of measuring GC duration).

E) \textbf{GUIDED RETRY} (Figure 4e) is now possible to be implemented by the application. Unlike timeout-based speculative retry that must wait for a certain time before sending backup requests, with \texttt{MitMEM}, the application can trigger retries as guided by the cancellation notification it receives from one of the resources.

\section{3.4 Goals}

\texttt{MitMEM} design achieves all the goals in the following ways (as summarized in Table 2). (1) The application remains simple because its job is straightforward: instantiate requests and perform speculative retries upon receiving cancellation notices. (2) Our approach is efficient because delayed requests are cancelled before being served, hence no extra load. (3) \texttt{MitMEM} has fast reactivity because the predictors we build always check the current delay, hence capable of adapting to millisecond burstiness. (4) The \texttt{MitMEM}-powered system also achieve flexibility in terms of adapting with workload changes because our rejection notification is based on per request deadline and per GC pause prediction.

Before we proceed, we emphasize that \texttt{MitMEM}’s objective is to provide a more effective way of mitigating contention in runtime resource layer. \texttt{MitMEM}’s notification design has finer granularity compared to other solution \cite{21, 27, 70, 71} that also combat tail-latency problem induced by GC. We acknowledge that there are other sources of tail latencies (e.g. data skew, CPU and libraries). For this reason, applications should still enable timeout-based speculative retry to anticipate “unknown” root causes. In the evaluation we show that \texttt{MitMEM}-guided retries produce an effective outcome.

\section{3.5 APIs}

We extended JVM OpenJDK with a new \texttt{URquest} class and its send/receive functions that wrap our custom system calls. This class allows Java applications to inform resource layers, including the runtime itself, about request-level information. For example, Java runtime now can drop requests and send cancellation notice via the corresponding socket. \texttt{MitMEM}’s APIs are straightforward as they are wrappers to our modified system calls.

- Class \texttt{URquest}
  This class contains the same fields as struct \texttt{urquest}. This class was added to the Java SDK that we modified.

- \texttt{URquest.Send(OutputStream stream)}
  This function writes a request to the \texttt{OutputStream} given as argument. The provided \texttt{OutputStream} should be the one obtained by calling \texttt{Socket#getOutputStream()} on the socket in charge of sending replies to the client.

- \texttt{URquest.Receive(InputStream stream)}
  In addition to receiving requests, this function plays another important role—it needs to check if there are requests that have been received (and not cancelled) during GC activities. Remember that cancellations are sent by our special non-blocked runtime thread. In order to do so, the runtime thread must “steal” incoming requests from the receiving socket while the application is frozen. Otherwise, the runtime does not know whether the arriving requests should be cancelled or not (the deadline and cancellable information is inside the request structure). Thus, when our non-blocked runtime thread receives a request that should not be cancelled, it puts it to a special temporary buffer. When the application threads awake and call \texttt{URquest.Receive(stream)}, this function first checks if there are requests in this temporary buffer and if so it returns the buffered request to the application. The application then will perform more \texttt{URquest.Receive(stream)} calls, which will read future requests either from this temporary buffer or from the network in case the buffer is empty.

\section{3.6 Implementation}

\texttt{MitMEM} is implemented in 1000 LOC in OpenJDK8. Our changes are modular and did not alter the main code of the platforms (e.g. the QoS code). For the application, we modify Cassandra v3.11.6 \cite{72} only in 120 LOC, demonstrating the non-intrusiveness of our approach. We use Cassandra (written in Java) due to its popularity \cite{73} and for showing the impact of multi-layer contention.
4 Evaluation

First, we present our performance evaluation on a single and six nodes cluster. And then, we evaluate the precision of our gc-pause prediction.

4.1 Performance

We conducted our evaluation to prove that MITMEM shorten the tail latency without introducing significant overhead. We carried out two experiments, a single node setup and a multi node cluster. The first experiment is done by involving cassandra-stress tool. The cassandra-stress tool is a Java-based stress testing utility for basic benchmarking and load testing a Cassandra cluster. This tool support a heavy-workload generator that can trigger MajorGC easily. Therefore, the cassandra-stress tool is a perfect choice to be used as heavy-workload client. In addition to that, we also have a light-workload client. The latency measured in Figure 5 are coming from the light-workload client. We use the heavy-workload client to trigger MajorGC only. Thus, we can observe the impact of fast-rejecting mechanism against light-workload client.

The second experiment is done in a six nodes cluster, half used as client and half as server nodes. We use Emulab d430 machine [74–76] that has 16 cores (32 logical), 64 GB DRAM, and 1 Gbps network. The retry overhead (machine-to-machine ping-pong) is only 120µs. We use Cassandra and warm up the cluster to have the user’s data reside in memory. Every key-value has three copies (the default configuration). For the client workload, we use a microbenchmark where every client node sends 10,000 requests per second. The experiments are performed several times to ensure reproducibility. For GC noises, we periodically send a large batch of non-critical requests to trigger GC within the experiment (e.g. mimicking a large database update or scan that is not latency sensitive, hence treated as regular, non-cancellable requests). This noise is sufficient for showing GC pauses within Cassandra.

We repeat our motivational experiment (§2.4) and compare the “Raw” setup (no tail mitigation) with MITMEM. In this configuration, we use 3 client nodes and 3 server nodes. First, the “Best” line in Figure 6 shows the latency CCDF of the client requests when there is no contention in the runtime layer (JVM). The line is vertically straight around x=0.7ms, the best-case scenario we should target. Second, the “Raw” lines (with noise) in Figure 6 shows that the noise inflicts long tail latencies to the user requests 5% of the time compared to the “Best” line. Finally, the MITMEM lines show that our methods can quickly react to the contention at runtime layer and eliminates 99% of the tail latencies caused by JVM’s GC (the large gap between the MITMEM and “Raw” lines).

We note that both in the “Best” and MITMEM lines, we still observe a small <1% latency tail, caused by “unknown” cases not covered by MITMEM (§3.4). For example, in the Emulab testbed, we always observe 0.3–0.5% long latency tail in a simple ping-pong workload, probably caused by network contention.

Furthermore, results also show that MITMEM’s overhead is negligible, as medians are very close between vanilla Cassandra and MITMEM-powered Cassandra. These results allow us to answer the research question by showing that MITMEM substantially reduces not only the tail of the latency distribution but also the overall request completion time (roundtrip latency) with negligible overhead. In summary, regardless of the service size, enabling MITMEM substantially reduces not only the tail of the latency distribution but also the ratio of the tail to the median, leading to a more predictable latency profile. MITMEM’s overhead is negligible, and there is no throughput loss for the considered cases.
shows the distribution of the 4.2 Limitation and Discussion

by +

that client can failover when receiving a

tion level modification is needed to implement the feature so

get system must have failover feature. Otherwise, applica-

tion mechanism techniques on the other layer will provide

contention status. The integrated rejection mechanism could

latency management as each layer knows its current resource

design opens up a new possibility of achieving better tail-

layer (i.e. CPU, network, libraries, etc.). The stackable

ing M

of contention on each resource layer. Therefore, combin-

Figure 7: MItMEM precision (§4.2). The figures show CDFs of GC pause prediction errors (D_{err}).

4.2 Precision

To evaluate MItMEM’s precision when it comes to pre-
dicting gc pauses, we chose 3 widely used Java benchmark
suites, SPECjvm2008, DaCapo, and Renaissance. To mea-
sure imprecision, similar as above, MItMEM predicts how
long every GC activity will run (D_{err}) and we also instru-
ment the JVM to measure the actual GC duration (D_{real}).
For every benchmark, we run the experiments until it cap-
tures 1000 D_{err} data points.

Figure 7 shows the distribution of the D_{err} values. Every
figure shows the two modeling algorithms we used (§3.2),
RANSAC and OLS. As shown in Figure 7a-c, MItMEM
is not precise in predicting GC pauses in benchmarks with
complex memory usage, due to the reasons described before
(§3.2). Fortunately, as we target storage systems, we notice
that its memory usage pattern is not too complex (i.e. simple
key-value (de)allocation). For now, it is sufficient for MIt-
MEM to estimate under this simple pattern. Figure 7d shows
that with 1 MB of key-value (de)allocation per second with
random sizes from 0.1 to 1 KB, MItMEM is only imprecise
by +/-4 ms in about 8% of the time with RANSAC.

5 Limitation and Discussion

We now summarize the limitations of MItMEM. First, MIt-
MEM relies on failover mechanism to exploit, hence the tar-
get system must have failover feature. Otherwise, applica-
tion level modification is needed to implement the feature so
that client can failover when receiving a server-busy notifi-
cation. Second, MItMEM only mitigates the pauses during
stop-the-world periods. Finally, MItMEM doesn’t consider
other source of contention (i.e. CPU and network) which
could be addressed by the future work as explained below.

MItMEM general design can be applied to other resource
layer (i.e. CPU, network, libraries, etc.). The stackable
design opens up a new possibility of achieving better tail-
latency management as each layer knows its current resource
contention status. The integrated rejection mechanism could
send a server-busy notification based on a certain degree
of contention on each resource layer. Therefore, combin-
ing MItMEM’s rejection on runtime layer with other rejection
mechanism techniques on the other layer will provide
sufficient coverage, especially when major resource layers
(e.g. CPU/thread, memory, and disk management) adopt our
method in supporting prediction and cancellation.

6 Related Work

Runtime management overhead in managed languages is a
major source of tail latency that various works attempt to
mitigate. Most of the works in this space focus on optimiz-
ing the JVM GC algorithms and coordinating the GC timing
within the cluster. The novelty of MItMEM’s technique is
that it offers finer granularity of tail-tolerance mechanism.
MItMEM does not flag a certain node as unavailable, but the
”unavailability” can dynamically change depending on
each request’s deadline. Thus, each request has its own per-
spective of the node status (busy or not-busy) relative to its
deadline. The summary of all related works are described
below.

Yak [21] says that Big Data systems built in Java suffer se-
vere GC issues due to the massive volume of objects created
to process input data. It proposes specialized heap reorganiza-
tion for big data workloads, dividing the heap into control
(i.e., permanent) space and data (i.e., transient) space, im-
proving GC execution time by up to 50x. NumaGC [23] says
that in NUMA environments GC performance is affected
by the many remote memory accesses while scanning the
reference graph. It designs a distributed GC processing
scheme that allows reference processing to be completed by
threads accessing that memory in a local fashion, improving
performance of classic GC algorithms up to 5.4x. Hotspot
PGC [34] says that ParallelGC performance is affected by
design choices that limit parallelism, such as unfairness in
mutex acquisitions, dynamic GC task assignments/stealing
and imperfect OS load balancing. It proposes a series of
modifications to solve these issues, such as re-designing
the work stealing algorithm and using GC thread affinity to
improve load balancing, improving ParallelGC execution time
up to 1.87x. NAPS [22] shows how badly throughput-
oriented GC algorithms (e.g., ParallelGC) scale beyond eight
cores, identifying the lack of NUMA awareness and heavy
lock contention as the major bottlenecks. It modifies Paral-
lelGC by adding lock-free queues and favoring local mem-
ory accesses, resulting in a GC implementation that does not
degrade its performance when parallelized in up to 48 cores.
GCI [77] observes that GC is one of the major factors affect-
ing tail latency in modern cloud web services. It proposes
a load balancing component that routes requests to non mem-
ory pre-assured nodes and manages garbage collection exe-
cutions, improving p99.9 request latency by almost 2x. Tau-
rus [27] observes that, since in a distributed setting runtime
environments are unaware of each other, individual garbage
collections have a negative effect in latency-sensitive work-
loads. It proposes an holistic runtime system that coordi-
nates garbage collection (similar to other works [70, 71]) ex-
executions among nodes, improving Cassandra p99.9 request latency by 2x. ROLP [33] says that there are unpredictable and unacceptably long tail latencies in Lucene, GraphChi, and Cassandra since these systems allocate all objects in the same space and rely heavily on object copying to promote longer-lifetime objects. Theoretically, reductions of long tail latencies are up to 85% for GraphChi, 51% for Lucene, and 69% for Cassandra. With that, it designs an application-code profiler at runtime to help pretenuring gc algorithms such as N2GC (made by the same author) to allocate objects with similar-lifetime close to each other to reduce the GC pause. It modifies OpenJDK8 by implementing the profiler and the new pretenuring GC.

7 Conclusion

In this paper we propose and evaluate the MITMEM – Mitigating Millisecond Tail Latency with Fast-Rejecting GC-Aware Mechanism. Instead of attempting to minimize the impact of garbage collector interventions, MITMEM transparently transforms these interventions into a temporary node unavailability that is evaluated per request basis. The novelty of MITMEM’s technique is that it offers finer granularity of tail-tolerance mechanism. MITMEM does not flag a certain node as unavailable, but the “unavailability” can dynamically change depends on each request’s deadline. Thus, each request has its own perspective of the node status (busy or not-busy) relative to its deadline.

MITMEM allows the garbage collector to run normally, but it will reject all the incoming requests with a special notification, Server-Busy. MITMEM can shed incoming requests during GC based on fast-rejecting mechanism design, which prevents STW from trapping those requests. MITMEM’s API is simple enough that most Java-based parallel applications can implement it without introducing a burden to the developer (e.g., adding 120 LOC to integrate with Cassandra). Our experiments indicate that the MITMEM-powered Cassandra successfully reduces the tail latency up to 99%. In the future, we would like to perform a broader evaluation of our approach. That includes combining MITMEM’s rejection on runtime layer with other rejection mechanism techniques on other layers such as OS, network, and library.

References


