Taming Tail Latency in Node.js with Event Dependency Graphs

Abstract

Cloud-based Web services are increasingly adopting the event-driven, scripting language-based programming model to achieve productivity and scalability. We focus on the tail latency of these new server applications using *Node.js* as a case study. We quantitatively demonstrate that *Node.js* applications suffer from serious tail latency issues. Root-causing and improving long tails, however, are difficult due to the inherent asynchronous event-driven execution of *Node.js* applications. We propose a profiling framework called Event Dependency Graph (EDG). The EDG allows us to gain insights about how the event-driven execution model impacts tail latency and how the managed runtime overhead, such as Just-In-Time compilation and garbage collection (GC), exacerbates the tail latency issue. We apply EDG-based analyses to *Node.js* and derive two critical conclusions. First, most latency tails are caused by CPU processing overhead as opposed to external I/O. Second, GC is responsible for almost 50% of the CPU execution time, rendering itself a top optimization target. Guided by The EDG, we demonstrate a GC-oriented turbo boosting technique which reduces tail latency by up to 19.1% with almost zero energy overhead, a significant improvement over two recent proposals that are event-agnostic. Finally, The EDG captures the new tail latency bottleneck after the GC impact is mitigated, demonstrating the potential to be a continuous latency analysis and optimization framework for event-driven servers.

1. INTRODUCTION

Cloud-based Web services are increasingly moving towards server-side scripting for productivity, flexibility, and scalability. Companies such as PayPal, eBay, GoDaddy, and LinkedIn have publicly announced their adoption of scripting frameworks, such as *Node.js*, as part of the back ends for their Web applications [8, 3, 5]. *Node.js* is a cross-platform JavaScript-based framework for developing responsive and scalable Web applications. For instance, Netflix announced that *Node.js* is the backbone of their Platform-as-a-Service infrastructure [4] and that it provides services to over 65 million subscribers, demonstrating the scalability of *Node.js* as a production-scale application platform for cloud services.

The success of *Node.js* comes from its combination of the event-driven execution model and the JavaScript programming language. The event-driven execution model provides better scalability and higher throughput by better multiplexing asynchronous I/O operations and CPU computations [36]. The use of the managed JavaScript scripting language provides Object-Oriented Programming (OOP) patterns, extensive software packages, and automatic memory management (i.e., garbage collection). These features greatly improve developer productivity and code portability, which are essential to sustain today's rapid Web development.

One of the key challenges facing effective development and deployment of large-scale Web services is long *tail latency*. For high-traffic websites, tail length is usually directly correlated with traffic volume. *Node.js*-based Web services are no exception. Using five industry-strength *Node.js* applications from the NodeBench suite [9], we quantitatively show that *Node.js* applications suffer from serious tail latency issues. Specifically, the latency for requests at the 99.9th percentile is about 9.1$\times$ longer than those at the 50th percentile.

Although tail requests typically constitute only a very small portion of the overall requests, to remedy them typically requires a significant amount of experimental effort. The first and foremost step is to understand the root-cause of the tails. *Node.js* presents a unique challenge to identify the root cause of tail latencies due to its event-driven nature and the complex managed language runtime systems. To the best of our knowledge, although plenty of prior work investigated the tail latency issue in traditional Web services [17, 22, 21, 39], none specifically focused on tail latencies that emerge from event-driven execution built on top of a managed runtime.

Event-driven execution inherently complicates the process of identifying a request's latency source. Large request latencies can stem from any of the three major components that operate independently and asynchronously: I/O, event queue, and event callback execution. There are additional non-application-level overheads that the JavaScript managed runtime framework introduces. Managed runtimes suffer from several well known sources of overheads, such as inline caching, garbage collection, code cache management, etc. [33]. These overheads are interleaved with the execution of the application itself, making it yet more challenging to pinpoint the sources of the tail latencies. Therefore, demystifying and piecing together a request's latency requires us to reconstruct the time taken to execute portions of the request across all independently operating components within one main thread. Depending on the request type, the bottleneck could be any one or more of the components, as we demonstrate.

We present a profiling framework for identifying root causes of tail latency in *Node.js*. The framework is capable of providing two pieces of information that are of particular importance to event-driven, managed-runtime-based *Node.js* servers. First, it explains which system-level component, i.e., I/O, event handler (callback) execution, or queuing, is responsible for the long latency in tail requests. Second, it explains to what extent the managed-runtime overhead impacts the tail latency.

The key insight behind our approach is to view of the processing of an request in event-driven servers as a graph where each node corresponds to a dynamic instance of an event and each edge represents the dependency between events. We refer to the graph as the *Event Dependency Graph (EDG)*. The critical path of an EDG naturally corresponds to the latency of a request. By tracking vital timing information of each event, such as I/O time, event queue delay, and execution latency, the EDG reconstructs the critical path of a request processing, thereby letting us attribute tail latency to fine-grained components for further optimizations.

Leveraging the EDG, we show that tail requests in most *Node.js* applications are bounded by CPU performance, which in turn is largely dominated by garbage collection. On average, garbage collection is responsible for almost 50% of event callback processing time in tail requests. We set garbage collection as our optimization target and accelerate the GC execution through frequency and voltage boosting. We call this tail reduction technique GC-Boost. GC-Boost reduces tail latency by up to 19.1% with negligible energy overheads.

Moreover, the EDG is a general technique so it can be reapplied after every optimization. Reapplying it after GC-Boost shows that the tail sources move to the generated native code, thus uncovering new avenues for continuous optimization.

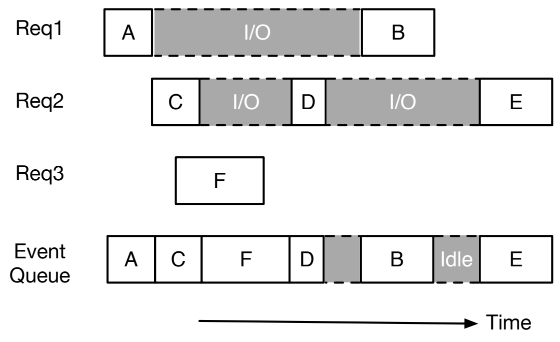
In summary, we make three major contributions:

- We propose the Event Dependency Graph (EDG) as a representation of request processing in event-driven, managed-language-based *Node.js* applications. The EDG allows precise deconstruction of request latency into relevant components amenable to optimizations.
- We demonstrate that the major bottleneck of tail latency in *Node.js* is the CPU processing overhead, especially the garbage collection of the JavaScript managed runtime.
- We design and prototype a GC-oriented tail reduction technique that boosts processor voltage and frequency during GC execution. It requires no modification to application source code, and achieves up to 19.1% tail reduction with almost zero energy overhead, a significant improvement over two recently proposed techniques.

The rest of the paper is laid out as follows. Section 2 provides a background for the event-driven model and its advantages over the thread-per-request model. Section 3 presents our experimental setup and shows tail latency characterization on industry-strength *Node.js* workloads. Section 4 presents the design and implementation of our EDG-based latency analysis framework. Section 5 applies our technique to *Node.js* workloads and discusses the main root cause for tail latency. Section 6 proposes a EDG-guided tail reduction technique called GC-Boost. Section 7 evaluates the performance of GC-Boost against prior arts and other alternatives. Section 8 discusses the related work, and Section 9 concludes the paper.

```
 1  http.createServer(
 2    // register HTTP request handler (event callback)
 3    function httpRequestHandler (request, response) {
 4      // parse file name from request url
 5      var filename = url.pathname;
 6
 7      // read file asynchronously
 8      fs.readFile(filename, "binary",
 9        function readFileCallback (err, file) {
10          // send file to client
11          response.write(file, "binary");
12      } );
13  } ).listen(8000);
```

(a) Example static file server application implemented in event-driven programming, which uses callbacks to handle I/O asynchronously.



(b) Event-driven programming overlaps execution between compute and I/O. While waiting for I/O to complete for one request, the event loop is free to process events from unrelated requests.

Figure 1: Event-driven execution overview.

2. BACKGROUND

Traditional server applications employ a "thread-per-request" execution model where each incoming user request is dispatched to a different OS thread in order to increase overall system concurrency. Recently, though, industry has started moving toward an event-driven execution model as an alternative. Event-driven execution is shown to deliver better system scalability and, therefore, has been widely adopted by major Web services vendors such as Netflix, Uber, and LinkedIn.

Instead of dedicating a thread to each request, the event-driven programming model translates incoming requests to events, each associated with a callback. Event callbacks are pushed into a FIFO event queue that is processed by a single-threaded event loop. Each event callback execution might generate further events, which are in turn pushed into the queue where they wait their turn to be processed.

Figure 1a shows an event-driven implementation of a file server that serves a static file in response to a user request. The application registers an HTTP event callback (line 3) that is triggered whenever a user request comes. In the body of the HTTP event callback, the application registers another

Table 1: Summary of event-driven server-side *Node.js* benchmarks studied in this paper.

Benchmark	I/O Target	#Requests	Description
Etherpad Lite [2]	N/A	20K	Real-time collaborative word processor, similar to Google Docs services. Simulated users create documents, edit document text, and delete documents.
Todo [11]	Redis	40K	Online task management tool, similar to the Google Tasks service. Users create new tasks, update tasks, and permanently remove outdated tasks.
Lighter [7]	disk	40K	Fast, simple blogging engine. Users request a variety of resources, such as web pages (HTML/CSS/JavaScript files), images, and font files.
Let's Chat [6]	MongoDB	10K	Self-hosted online chat application, similar to Slack service. Users create chat rooms, send/receive messages posted in the same room, and leave rooms.
Client Manager [1]	MongoDB	40K	Online address book for storing client contacts and other information. Users add new clients, update client information, and remove clients.

callback for a file I/O operation (line 9). The callback is enqueued and executed whenever the I/O operation finishes.

The benefit of event-driven programming is its non-blocking, asynchronous execution that overlaps I/O and compute. Although the event-driven model uses only one thread to execute event callbacks, it exploits concurrency by interleaving event callback executions with asynchronous I/O operations.

Figure 1b illustrates the case when the server is handling three requests concurrently. Each request execution can be viewed as a sequence of different event executions that are interleaved with asynchronous, non-blocking I/O operations. Once *event A* from request 1 (Req1) finishes execution and starts an I/O operation, the event loop does not wait idly for the I/O to complete; rather, it shifts its focus to executing *event C* from request 2 (Req2). When the I/O operation of request 1 completes and returns to the CPU, it triggers another event, *event B*, which is pushed onto the event queue and will be executed when it reaches the head of the queue.

The event-driven execution model allows the system to maximize the overlap between CPU computation and I/O operations. Therefore, it maximizes the utility of threads since the threads are not blocked, idling on I/O operations. As a consequence, the overhead of thread management diminishes in an event-driven system compared to a thread-per-request system. Thus, event-driven execution achieves greater scalability than the thread-per-request model [37, 36].

3. LATENCY CHARACTERIZATION

We first introduce the *Node.js* workloads that we study and explain how we generate loads to mimic an enterprise deployment (Section 3.1). Next, we study the latency profiles of these workloads (Section 3.2) and demonstrate that tail latency is a serious issue for event-driven workloads.

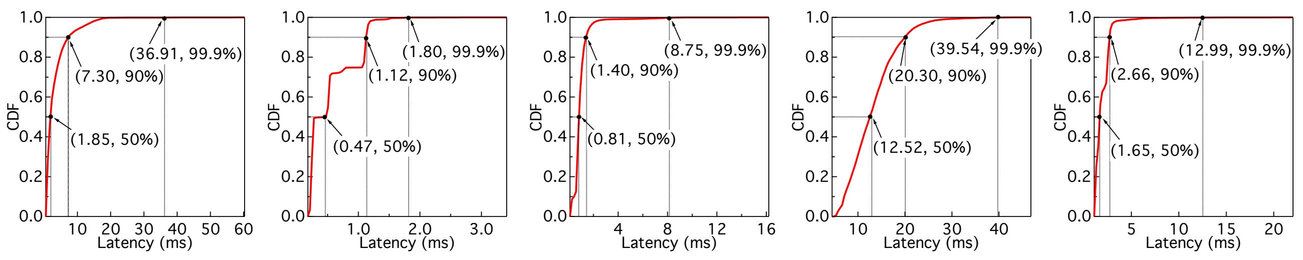
3.1. Workloads and Load Generation

We use industry-strength *Node.js* server applications. We pull our applications from the NodeBench application suite [9] that includes *Node.js* server applications from prior work [40], as well as new applications that encapsulate additional server functionalities. We select five different applications from the NodeBench suite. We exclude other NodeBench applications because they have limited client-server communication with and/or have extremely long average request latency (larger than 1 second), therefore, are much less latency-critical and hence unqualified for typical server applications.

In order to fully characterize event-driven applications, it is necessary to study applications with both diverse application domain and event types. Table 1 lists the applications we study and their descriptions: *Etherpad Lite* is a online document editor that supports multi-user collaboration; *Todo* is a task management application; *Lighter* is a blogging platform for online content management; *Let's Chat* is a self-hosted web messaging application; and *Client Manager* is a client contact management tool for business managers. Besides covering various application domains, these applications also exercise different types of I/O including conventional file I/O, NoSQL database service (*MongoDB*), and in-memory key-value store (*Redis*). Different I/O services trigger different event execution behaviors, and in doing so they cause variations in response latencies. Thus, studying different I/O services is important for characterizing event-driven applications.

We deploy the *Node.js* server applications on a four-core Intel i7-4790K machine with 32 GB DDR3 DRAM and a 240 GB SSD to deliver high I/O performance. We simulate our client and host our database services on separate server-class machines connected through 1 Gbps intranet in order to isolate event-driven compute from I/O services. We set the Linux processor frequency governor to "userspace" and the default operating frequency to 2.6 GHz unless stated otherwise. The use of statically fixed operating frequency can prevent tails caused by improper processor frequency governor [25].

To generate the loads, we use *wrk2*, an HTTP load testing tool that supports parameter tuning and dynamic request generation [12]. For each application, we simulate a series of HTTP requests to mimic a realistic usage scenario as if users were interacting with the application. Each usage scenario consists of a sequence of 4 to 10 different types of HTTP requests that are specific to each application's functionalities. The "#Re-



(a) Etherpad Lite. **(b)** Todo. **(c)** Lighter. **(d)** Let's Chat. **(e)** Client Manager.

Figure 2: Cumulative distribution function of server-side latencies (millisecond). In all five workloads, there is a prominent tail.

quests" column in Table 1 shows the total number of requests that are issued for each application. We simulate a total of at least 10,000 requests for each application and discard requests in warm-up and cool-down phases.

3.2. Serious Tail Latency

We focus on server-side latency, i.e., the latency between when a client request is received and when its response is issued. Server-side latency does not include client-server network latency and client-side processing latency but does include the latency between a server application and its backend database.

We instrument the *Node.js* HTTP processing library to measure server-side latency directly without altering the application code. Specifically, we log a timestamp whenever an HTTP request arrives and whenever an HTTP reply is about to be sent out. We associate requests with replies using the event dependency analysis later presented in Section 4. We keep timestamps in memory and flush them to disk only once per minute. Overall, we require at most 8 MB of memory, which has negligible impact on server application performance.

To quantify the request latencies, Figure 2 shows the cumulative distribution functions (CDFs) of server-side latency for each application. Any point $< x, y >$ reads as such: y% of the total requests have a latency below x ms; in other words, the y^{th} percentile tail is x ms. We report three representative latencies: 50^{th} percentile, 90^{th} percentile, and 99.9^{th} percentile.

With respect to the latency variation across the applications, the *Node.js* applications' response latency varies drastically from one application to another. For applications where the processing per request is typically small, such as in the case of *Todo* where each request is updating a small entry, the response latencies are short (less than 2ms for each request). However, for heavyweight applications such as *Let's Chat*, the latency is significantly higher (on the order of 10s of milliseconds).

With respect to the overall distribution of server-side latency, we make two key observations. First, tail latency is a serious issue for all *Node.js* applications. In each graph, the CDF curve rises very quickly initially; however, as the curves approach the top, the slopes diminish and the curves become long, nearly flat lines, indicating there is a significant difference between the longest latencies and the majority of latencies. On average, the latency for requests in the 99.9^{th} percentile is about $9.1\times$ longer than those in the 50^{th} percentile. Second, we observe that in most cases the 99.9^{th} percentile point is almost at the midpoint of the trailing portion of CDF curve, indicating 99.9^{th} percentile is a good tail evaluation metric [34, 38]. We refer to requests between the 99.8^{th} percentile and 100^{th} percentile as "tails" unless stated otherwise. Note that this tail definition is only to simplify further discussions. Our methodology is generally applicable however tails are defined.

4. LATENCY DECONSTRUCTION USING THE EVENT DEPENDENCY GRAPH

We present a method for gaining a deeper understanding of tail latency root causes at the application level (while not losing insights at the system level). There are two important aspects of application-level information in *Node.js* servers that are of particular interest to us. First, we want to explain which events are causing long request latencies as *Node.js* servers are event-driven. Second, we want to understand whether it is the native code or the virtual machine (VM) functionalities that lead to tails as *Node.js* servers are managed language-based.

To this end, we first introduce an analytical model for request latency in event-driven servers, which provides a guideline for attributing sources of tail latency to various root causes (Section 4.1). We then describe a new runtime analysis technique called Event Dependency Analysis (EDA), which is based on dynamically generated Event Dependency Graphs (EDGs), for performing root-cause analysis (Section 4.2). We present the implementation details of conducting EDA (Section 4.3). Finally, we present a first-order validation of our root-cause analysis framework (Section 4.4).

4.1. Analytical Latency Model

We present a direct, precise, and fine-grained way to determine tail latency root causes in event-driven systems. Based on an analytical model, this model precisely attributes the latency of any request as a function of its various application and system components. We statically instrument the event-driven runtime, which in our case is *Node.js*, and collect the latency of each component during runtime. This allows us to identify the root cause of request latency, from the system-level down to the application-level breakdown.

Figure 3 intuitively shows the hierarchy of the latency com-

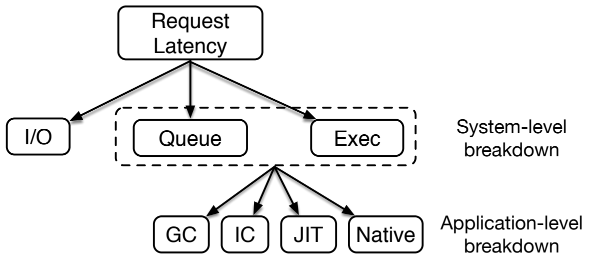


Figure 3: The hierarchical latency analytical model.

ponents captured by the analytical model. A request's latency is broken down into its I/O, Queue, and Execution time components (as first explained in Section 2). The queuing and execution time components are further broken down by the model to provide application-level insights, specifically in a manner that can provide insights into the managed language components. We now explain the analytical model details.

A request in event-driven servers is a sequence of event executions. Therefore, request latency is the sum of the latencies of individual events that are on a request's critical path. Equation 1 expresses the relationship. R denotes a particular request, $ECP(R)$ denotes the set of events that are on the critical path of R, and $T(e_i)$ denotes the processing latency of e_i.

$$Latency(R) = \sum_{i=1}^{N-1} T(e_i), e_i \in ECP(R) \tag{1}$$

The event processing latency $T(e_i)$ can be further decomposed into three major system-level components—I/O, scheduling, and execution—expressed as follows:

$$T(e_i) = IO(e_i) + Sched(e_i) + Exec(e_i) \tag{2}$$

The I/O latency refers to the latency of the I/O operation leading e_i. After an I/O operation is fulfilled and comes back to the CPU, it pushes e_i into the event queue. The scheduling latency then refers to the time e_i has to wait in the event queue before being scheduled for execution. After an event gets scheduled (i.e., reaches the head of the queue), the execution latency refers to the event callback execution time.

In order to gain application-level insights of root causes, we further dissect the event execution latency into four major finer-grained components: native code (Native), just-in-time compilation (JIT), garbage collection (GC), and inline cache miss handling (IC) (Equation 3). We focus on the four components because they have been shown to be responsible for the majority of execution cycles in JavaScript-based applications [29, 13]. The scheduling latency is implicitly expressed in the four categories as the scheduling time is equivalent to the execution time of all the preceding events.

$$Exec(e) = Native(e) + JIT(e) + GC(e) + IC(e) \tag{3}$$

Equation 1, Equation 2, and Equation 3 together form a model for request latency that is sufficiently fine-grained to capture the system- and application-level component breakdowns and explain requests with long tails.

Table 2: Description of common events in *Node.js*

Event Type	Name	Description
File System	readFile	file read
	readDirectory	directory read
	stat	file status (e.g., permission)
Network	accept	new TCP connection accepted
	read	TCP/UDP packet received
IPC	signal	OS signal notified
Timer	TimeOut	timer expired
Miscellenaous	source	the first event in a request
	sink	the event that sends a response
	idle	user-defined low-priority events

4.2. Event Critical Path

To use the analytical model for root-cause analysis we must identify the event critical path $ECP(R)$, a set containing all events on the critical path between a request R and its corresponding response. The key to track the event critical path is to identify the dependencies between events, based on which we construct an event dependency graph (EDG) where each node in the graph corresponds to an event and each edge in the graph represents a dependency between two events. With the EDG, we can readily identify the event critical path by following the path from the request to the response.

Definition We define event dependency as a happens-before relationship, denoted $\prec$. We use $E_i \prec E_j$ to indicate that E_j depends on E_i. There are three types of happens-before relationships in *Node.js* applications:

- If event E_i is the first event triggered by the request R, then we have $E_i \prec E_j$, where E_j denotes any other subsequent event. We call the first event triggered by an incoming request a *source* event.
- If the callback of event E_i registers an event E_j, i.e., E_i issues an I/O operation which eventually leads to E_j, then we have $E_i \prec E_j$.
- If event E_i is the event that sends a response corresponding to the request R, then we have $E_j \prec E_i$, where E_j denotes any preceding event. We call the event that issues the response a *sink* event.

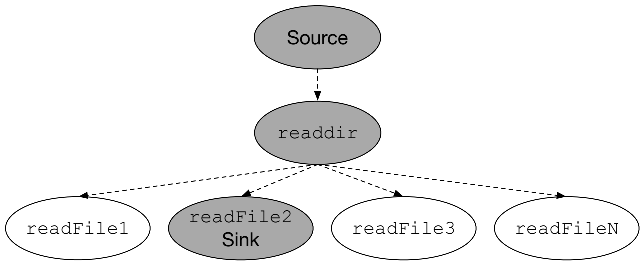
Example We use an example to illustrate the event dependency concept. Figure 4a describes a simple server which, upon request, returns all the files under a particular directory. Figure 4b shows its corresponding event dependency graph. The source event (code not shown) registers the *readdir* event, which when triggered returns all the file names under a directory. The callback of *readdir* then iterates through all the file names and further registers a *readFile* event for each file. Assuming there are N files, we have an event dependency between the *readdir* event and each *readFile* event.

As soon as the *readFile* event returns, its callback checks whether every file has been processed. The last *readFile* event

```
var count = N;

fs.readdir(src_dir, function afterReaddir (err, files) {
  files.forEach(function Loopbody (file, index) {
    var fname = src_dir + "/" + file;
    fs.readFile(fname, function afterReadFile (err, data) {
      if (--count === 0)
        sendResponse();
    });
  });
});
```

(a) Directory read application.



(b) Event dependency graph (EDG). Gray events are on the critical path.

Figure 4: Example application and its EDG.

that returns sends all the file content back to the client in one response. It is important to note that the *N readFile* events do not necessarily come back in the same order in which they are registered because the I/O subsystem can arbitrarily reorder I/O operations as it sees fit. Assume in our case that the second read event is the last one to return, at which point *readFile2* is the event that sends the response and therefore is the sink event. The event critical path in this case proceeds from the source event to the sink event as shown in Figure 4b.

4.3. Implementation

We implemented a runtime mechanism that constructs the event dependency graph to track the latency within each processing component of the analytical model. Our implementation is integrated directly into the *Node.js* platform, and so there is no need to modify the application source code.

The *Node.js* runtime automatically identifies the event dependency graph while an application is executing. During that time it also logs timestamp data needed to construct the latency analytical model. We store the logging information on a local file system and post-process the data after the server application terminates to construct the event dependency graph.

EDG Construction Table 2 lists the set of events that we instrument to correctly construct the EDG. We consider I/O events related to file and network operations (including both TCP and UDP sockets and DNS resolution events) and other sources of asynchronous execution. For example, *readFile* and *readdir* in Figure 4a are events caused by file system accesses. Other events also include Inter-Process Communication events, timers, etc. Generally speaking, one must instrument the modules that support asynchronous execution.

The EDG can be constructed only at runtime due to I/O operation reordering. According to the definition of event dependency, recording event dependency is equivalent to tracking the event callback registration. We make the observation that whenever a new event is registered, a new JavaScript function object will be created in order to be passed as the callback function. Leveraging this observation, we intercept all the JavaScript function object creation and invocation sites inside the JavaScript Virtual Machine. Whenever a function object is created, we record the current event's ID in a shadow field of the function object. Whenever the function is invoked, our instrumentation compares the current event ID with the shadow event ID logged within the function object. When the two event IDs do not match, we discover a new event dependency.

Log Event Timestamps For each event, we record four important timestamps: register time, ready time, begin time, and end time. Register time refers to the time an event is registered, which is also when its associated I/O operations are issued. Ready time refers to the time when an event is ready, i.e., its preceding I/O operation finishes and the event is pushed into the event queue. Begin time and end time refer to the time an event callback starts and finishes execution, respectively. According to the four timestamps, the three major components in Equation 2 can be derived as follows:

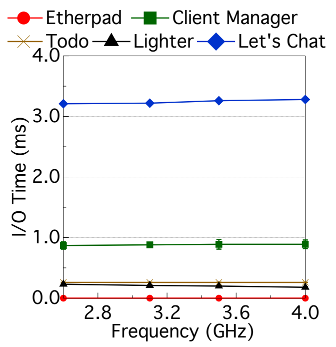
$$\begin{aligned} IO(e) &= Ready - Register \\ Sched(e) &= Begin - Ready \\ Exec(e) &= End - Begin \end{aligned}$$

Logging timestamps is mostly trivial except for the ready time, which is because I/O events are first observed by the OS kernel and only then (i.e., later) become visible to the userspace when the *Node.js* runtime polls it (via *epoll* in a POSIX system). *Node.js* only polls ready events when all the current events in the event queue finish. As such, the times are different between when an event is truly ready (in the kernel space) and when it becomes visible to the *Node.js* runtime (in the userspace). In practice, this difference can be large.

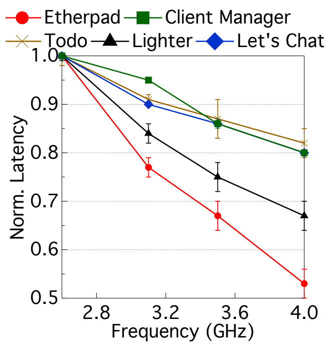
To precisely track an event's ready time, we create a helper thread that periodically polls at a much finer granularity than *Node.js*'s default polling mechanism. We verify that the helper thread introduces only negligible impact on both average and tail latency. This is because *Node.js* applications are mostly single-threaded (backed by a few worker threads), and there are abundant CPU resources reserved for the helper thread.

4.4. Validation

It is difficult to validate the root-cause analysis approach because that would entail measurements that are absolutely non-intrusive. Instead, we report our best-effort validation by experimentally proving two propositions that should hold true if our root-cause analysis framework is performing as expected: 1) I/O time is independent of the processor frequency; 2) total



(a) I/O latency (ms).



(b) Normalized total latency.

Figure 5: Validation experiments. All reported data is averaged over all requests, including both non-tail and tail requests.

server-side latency decreases as we increase processor frequency, and the decreasing slope depends on the I/O intensity.

We validate the first proposition by reporting the I/O times of all benchmarked applications as the processor frequency scales from 2.6 GHz to 4.0 GHz. The I/O times are averaged across all requests, and include both the tail and non-tail requests for each application. Figure 5a presents the results with the error bars representing one standard deviation across 10 runs. The I/O time for all of the five applications is steady across the different frequency settings. Furthermore, the small and steady standard deviations across the runs proves that the results are statistically significant.

We also validate the second proposition by reporting the total server-side latencies as the processor frequency scales from 2.6 GHz to 4.0 GHz. The results are normalized to the latency at 2.6 GHz for each application and are shown in Figure 5b. We make two observations. First, overall the server-side latency decreases as the frequency increases. Second, the applications have different decreasing slopes because the applications have varying levels of I/O intensity, matching our intuition. For example, *Etherpad Lite* has the highest decreasing ratio. Recall from Section 3 that *Etherpad Lite* does not trigger any external I/O. We will present a detailed I/O time breakdown in Section 5.

5. TAIL LATENCY ROOT-CAUSE ANALYSIS

We conduct root-cause analysis of *Node.js* applications using our hierarchical analysis framework to identify tail latency bottlenecks. We find that the root cause at the system level is largely dictated by the CPU's processing time (i.e., queueing and execution) as opposed to the I/O subsystem time (Section 5.1). We go one step further and look into the sources of the CPU processing overhead at the application and runtime system level; we identify garbage collection as a major bottleneck and target for optimization (Section 5.2).

5.1. System-level Tail Latency Root-cause Analysis

System-level request latency consists of three components: I/O, queueing, and execution. Figure 6 shows the system-level latency breakdown of the benchmarked applications as stacked bar plots. Each bar corresponds to a particular request type (i.e., HTTP request URL) within an application. Though there are only a few request URLs per application, there are many dynamic instances (e.g., different clients) of a request type throughout an experiment. Each bar shows the average latency for all instances of a request type over 10 experiment runs. For comparison purposes, we show the results of both non-tail requests in the left half and tail requests in the right half of each figure. The CPU frequency is fixed at 2.6 GHz.

We find that the tail latencies of the *Node.js* applications are predominately caused by event callback execution and the queueing overhead as opposed to I/O time. Across all *Node.js* applications, the average queuing time contributes to about 45.8% of the tail latency and the average callback execution time contributes to 37.2% of the tail latency. The significance of queuing and callback execution time indicates that tail latencies in *Node.js* applications are bottlenecked primarily by CPU computations instead of long latency I/O operations.

On the contrary, I/O time only contributes to about 21.2% of tail latency for applications that involve I/O operations (i.e., excluding *Etherpad-Lite* which has no I/O operations). In extreme cases such as *Todo*, the I/O time can amount to almost half (45.2%) of the tail latency. To remedy the I/O-induced tail latency issues, one could optimize the network stack or database operations. Although such optimizations are beyond our scope, it is the EDG that let us identify the bottleneck.

Table 3: Application-level latency breakdown. The average Just-In-Time (JIT) compilation time is effectively "zero" because JIT happens infrequently, when the applications are in steady states.

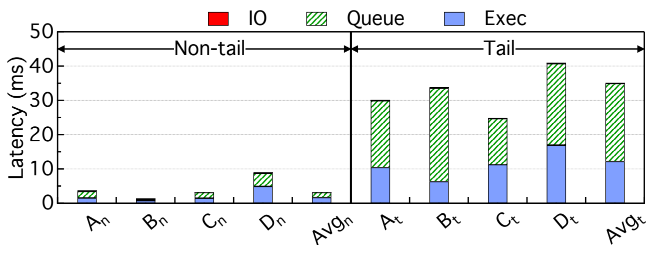
	Non-tail				Tail			
Application	**IC**	**GC**	**JIT**	**Native**	**IC**	**GC**	**JIT**	**Native**
Etherpad Lite	0.3%	**1.9%**	0%	97.8%	7.0%	**41.9%**	0%	51.1%
Todo	1.8%	**0.5%**	0%	97.7%	0.6%	**31.0%**	0%	68.4%
Lighter	4.9%	**4.7%**	0%	90.4%	4.4%	**45.3%**	0%	50.3%
Let's Chat	2.7%	**3.9%**	0%	93.4%	3.6%	**48.4%**	0%	48.0%
Client Manager	3.2%	**3.1%**	0%	93.7%	3.9%	**70.3%**	0%	25.8%

5.2. Application-level Tail Latency Root-cause Analysis

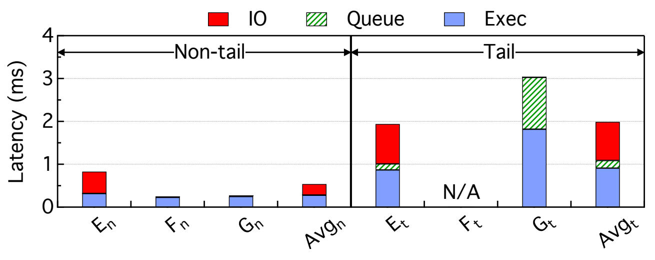
The high queueing and execution time in tails indicate that improving computation performance, such as scaling up processor frequency, could be a potential optimization strategy. However, such performance optimizations often come with high energy overhead. To better optimize for tail latency, we must pinpoint the root-cause in queueing and execution.

We further breakdown the queueing and execution time into four application-level components, including the JIT, GC, IC miss handling, and Native code (as discussed in Section 4.1). Table 3 shows the breakdown results for each application. The table shows the data across these components during the non-tail requests, as well as the tail requests.

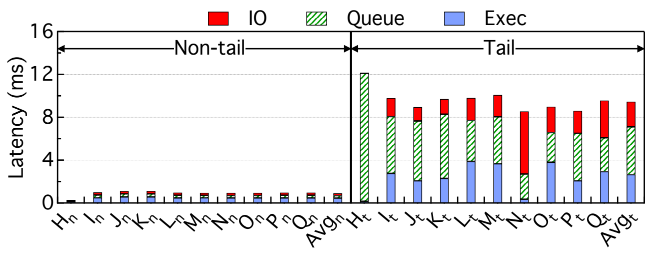
We make two key observations. First, the results show



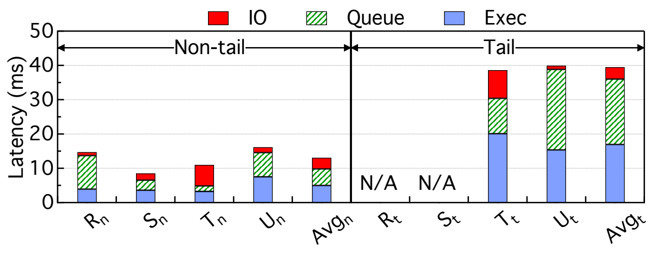
(a) Etherpad Lite



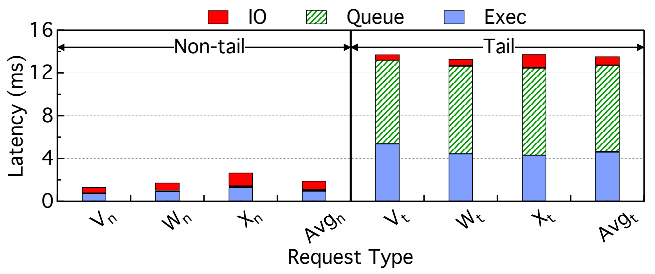
(b) Todo



(c) Lighter



(d) Let's chat



(e) Client Manager

Figure 6: System-level latency breakdown. Each bar represents a particular request type (i.e., HTTP URL). Bars in the left half show the average latency for non-tail requests, and bars in the right half show the average latency when requests are in the tail.

that the JIT and IC miss handling, the two commonly optimized components in any managed runtime system, are not the most lucrative optimization targets for reducing tail latency in *Node.js* applications. Tail requests spend minimal time (up to 7%) in the JIT compiler, indicating that the compilation of JavaScript code is not a major cause of tail latency. Most of the code is compiled early on, and we are focused on the steady state application behavior. Therefore, we can effectively assume that the code executed in tail requests is mostly compiled "ahead of time." The same conclusion also applies to IC miss handling. Although the performance penalty of missing the inline cache in JavaScript programs is high, tail requests do not suffer from inline cache misses.

Second, native code execution and garbage collection constitutes about 48.7% and 47.4% of the processor time on average, respectively. Native code contributes heavily to the execution time because it carries out the fundamental functionality of an application. Garbage collection, on the other hand, is merely an artifact of *Node.js* using JavaScript as the underlying language. The high GC time is thus an overhead. Our results suggest that GC is a rewarding optimization target.

6. TAIL LATENCY OPTIMIZATION

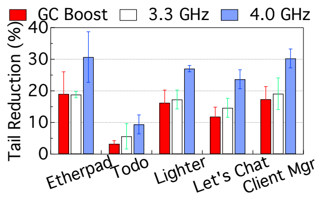
The EDG is a powerful tool that provides us critical insights into the runtime characteristics of event-driven servers. Leveraging the EDG, we demonstrate how frequency boosting can be applied to reduce tail latency at almost no energy overhead. Frequency boosting through Dynamic Voltage and Frequency Scaling (DVFS) or Turbo Boost is a well-established technique to increase performance and it has been extensively used to reduce tail latency in traditional servers [21, 17]. Blindly increasing processor frequency, however, comes with a high energy penalty. Therefore, the challenge is to wisely choose which application execution phase is critical to accelerate.

Fortunately, the high GC overhead exposed by the EDG naturally leads us to target GC for reducing tail latency. Although a variety of techniques were previously proposed to improve GC performance (e.g., concurrent GC, hardware GC accelerator [19]), most of them require significant compiler/hardware modifications. Our goal is to leverage frequency boosting to improve GC performance *at minimal engineering cost*.

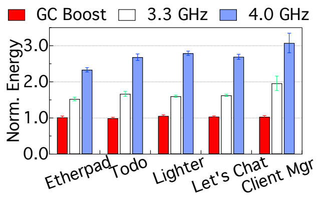
Frequency boosting is a natural candidate for reducing GC execution for three reasons. First, the *V8* JavaScript Engine uses a "stop-the-world" GC, which means that the GC's execution is serialized with respect to the application code. Thus, improving GC performance directly impacts the tail latency as GC contributes almost 50% of CPU time in the tail.

Second, using hardware performance counters (PAPI version 5.4.3 [10]), we find that the GC's average instruction-per-cycle (IPC) is modestly high at 1.3, suggesting that GC is compute-bound and thus can benefit from frequency boosting.

Third, the GC contributes to only a small fraction of the non-tail request latency. Table 3 shows the latency breakdown for non-tail requests. GC has little to no impact on non-tail



(a) Tail Reduction



(b) Total Energy

Figure 7: Tail reduction and energy consumption after boosting frequency. The results are averaged over 10 experiments and normalized to latency and energy under 2.6 GHz.

requests, suggesting that increasing the CPU clock frequency during GC would significantly improve the tail latency, while having minimal impact on the overall energy consumption.

We implemented our GC frequency boosting optimization (GC-Boost) as a userspace scheduler integrated into *V8*, which requests that the kernel set the CPU frequency to a certain boost value. Typical server processors operate at a nominal frequency ranging between 2 GHz and 3 GHz. We choose 2.6 GHz as our baseline. This choice is well aligned with recent tail latency papers [21, 17]. The peak frequency of our server is 4.0 GHz, which is the maximum boost frequency. Our scheduler dynamically increases the CPU clock frequency from 2.6 GHz to 4.0 GHz whenever *Node.js* enters GC.

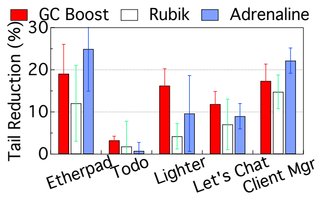
7. EVALUATION

This section evaluates the proposed GC-Boost technique. We show that GC-Boost can reduce tail latency by up to 19.1% with almost zero energy overhead (Section 7.1) and outperforms Rubik and Adrenaline, two recently proposed tail reduction techniques (Section 7.2). We further evaluate GC-Boost against an orthogonal technique that fine-tunes the GC-related parameters (GC-Tuning). While the two techniques entail different trade-offs, GC-Boost is more generally applicable (Section 7.3). Finally, we summarize the results and outline some further optimization opportunities (Section 7.4).

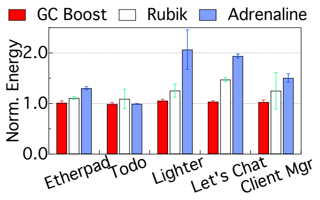
7.1. GC-Boost Results

Figure 7a shows the tail latency reduction under frequency boosting with the error bars indicating one standard deviation. For comparison purposes, we also show the tail latency while globally setting the frequency to 3.3 GHz and 4.0 GHz. Fixing the frequency at 4.0 GHz yields the highest system performance and the optimal tail reduction one can gain from frequency boosting. All of the results are averaged across all the benchmarked applications and are normalized to the results without boosting (i.e., statically fixed at 2.6 GHz).

We find that boosting CPU frequency during GC consistently improves the tail latency across different applications. The improvements are comparable to globally increasing the CPU frequency to 3.3 GHz. Overall, GC-Boost reduces the tail latency by 13.6% on average and up to 19.1% in the case of *Etherpad Lite*, which is around 50% of the maximal possible tail reduction attained by fixing the frequency to 4.0 GHz.



(a) Tail Reduction



(b) Total Energy

Figure 8: Tail reduction and energy consumption for GC-Boost, Rubik and Adrenaline. The results are averaged over 10 experiments and normalized to latency and energy under 2.6 GHz.

Todo is the least sensitive to GC frequency boosting with an improvement of about 3.3%. This is because GC only constitutes about 16% of *Todo*'s tail latency.

Boosting GC performance has little impact on the overall energy consumption. Figure 7b shows that increasing the CPU frequency to 4.0 GHz during GC introduces only a 2.8% energy overhead compared to the baseline 2.6 GHz. This is because GC contributes to only about 3% of the overall request latency. In contrast, globally boosting the CPU frequency to 3.3 GHz and 4.0 GHz introduces significant energy overheads of 67.6% and 171.2%, respectively. We conclude that GC-Boost significantly reduces tail in an energy-efficient manner.

7.2. GC-Boost vs. Prior Art

In this section we evaluate GC-Boost against Adrenaline and Rubik, which are two recently proposed DVFS schemes for reining in the tail latency in traditional non-event-driven servers. We show that the EDG-guided GC-Boost technique achieves higher tail reduction with much lower energy overhead, highlighting the importance of capturing event-specific characteristics in optimizing tail latency in *Node.js* servers.

Rubik Comparison The goal of Rubik [21] is to meet the tail latency target with minimal power consumption. Rubik treats request arrival as a form of stochastic random process and determines the optimal frequency for each request according to the probability distribution of request processing time. In our implementation of Rubik, we used the reduced tail latency after applying GC-Boost as the tail latency target.

Figure 8a and Figure 8b compare the tail reduction and energy overhead of GC-Boost and Rubik, respectively. We find that GC-Boost consistently achieves better tail reduction (up to 3×) with lower energy overhead compared to Rubik across all five benchmarked applications. In addition, Rubik also missed the tail target for most applications (not plotted).

The main reason behind Rubik's limited capability to reduce tail latency in *Node.js* is that Rubik's statistical model does not directly apply to event-driven servers. In particular, Rubik assumes that server applications process requests sequentially and independently. However, these assumptions do not hold for event-driven applications where requests are divided into events that are processed in an interleaved manner.

Adrenaline Comparison Adrenaline [17] is motivated by the observation that certain types of requests have a higher

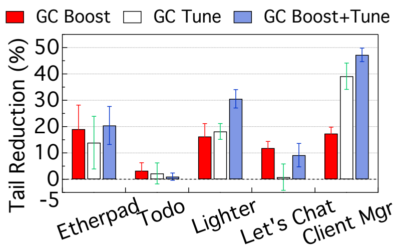


Figure 9: Tail latency reduction comparing GC-Boost from Figure 7 against GC-Tuning. The combined technique reduces tails significantly, in some cases even outperforming 4.0 GHz.

probability of becoming tail requests and they can be identified by indicators such as their request URL. Adrenaline boosts the frequency when encountering any such request. We build an oracle version of Adrenaline by statically identifying request URLs that have the highest probability of becoming a tail and boost those requests to maximum frequency at runtime.

Figure 8a and Figure 8b show the tail reduction and energy overhead comparison between GC-Boost and Adrenaline, respectively. We find that Adrenaline achieves better tail reduction in two out of five benchmarked applications (*Etherpad Lite* and *Client Manager*). However, it consistently introduces much higher energy overheads than GC-Boost. On average, Adrenaline costs 0.51× more energy than GC-Boost even as they achieve approximately the same tail reduction.

The significant energy overhead of Adrenaline is caused by the fact that request type (i.e., URL) is not a perfect indicator of tail latency in *Node.js*. We find that any request URL in *Node.js* applications has at most a 1.28% chance of becoming a tail. Adrenaline wastes energy boosting non-tail requests. In addition, Adrenaline boosts frequency throughout the entire request processing. In contrast, GC-Boost judiciously accelerates only the GC phase guided by our EDG analysis.

7.3. GC-Boost vs. GC-Tuning

It is worth noting that tuning application/system parameters, such as TCP/IP configuration and CPU affinity [14], is a common practice in deploying server applications. In this section, we carefully tune the GC-related parameters such as heap sizes and thresholds for each application to study the upper-bound of performance improvement without frequency boosting. We name this approach GC-Tuning. Our goal is not to propose GC-Tuning as a new optimization technique; rather we use GC-Tuning to thoroughly evaluate GC-Boost.

The *V8* JavaScript engine uses a generational garbage collector [20], which organizes the heap into two spaces (generations): a new-space and an old-space. We focus on the sizes of the new-space and old-space heaps as the two key tuning parameters after experimentally observing that GC performance in *Node.js* is most sensitive to the sizes of the the two heaps.

Figure 9 compares the tail latency reduction of GC-Tuning and GC-Boost. Properly sizing the heap achieves 14.8% tail reduction on average and up to 39.1%. Compared to GC-Boost's improvement at 13.6%, GC-Tuning has a similar average tail latency reduction. The primary difference between GC-Boost and GC-Tuning is in their generality. GC-Boost is application-agnostic, requiring minor modifications only to the *Node.js* runtime (*V8* specifically) without affecting the application code. As such, it is a general tail-reduction strategy that once deployed can benefit all *Node.js* applications running in a server system. In contrast, GC-Tuning is application-specific, requiring considerable tuning effort.

In summary, GC-Tuning makes the *Node.js* runtime more closely tied to a particular application, potentially hurting the platform's generality in the long-term. However, since GC-Boost is orthogonal to tuning GC parameters, both GC-Boost and GC-Tuning can be applied at the same time. If GC-Boost is combined with GC-Tuning, Figure 9 shows that we can achieve, on average, a 21.7% reduction in tails with up to a 47% best-case tail reduction.

Table 4: Summary of all five tail reduction schemes. All values are normalized to 2.6 GHz without any optimizations.

Technique	Tail Reduction(%)	Norm. Energy
GC-Boost	13.56	1.03
GC-Tuning	14.84	1.01
GC-Boost+Tuning	21.68	1.05
Rubik	7.99	1.22
Adrenaline	13.31	1.56

7.4. Summary

We now summarize the evaluation of GC-Boost against all other alternatives. Table 4 lists the average tail reduction and normalized energy for GC-Boost, GC-Tuning, GC-Boost+Tuning, Rubik, and Adrenaline. GC-Boost and GC-Tuning significantly outperform Rubik and Adrenaline in both tail reduction and energy overhead. GC-Boost+Tuning achieves maximal tail reduction (22% on average) with acceptable extra energy (5%). Given the simplicity of their implementation, GC-Boost, GC-Tuning, and GC-Boost+Tuning are readily useful for real-world products. We plan to release our reference implementation for GC-Boost, GC-Tuning, and GC-Boost+Tuning after publication.

To further explain the effect of our techniques, Table 5 shows the latency breakdown at the 99.9th percentile before and after applying GC-Boost+Tuning to *Client Manager*. GC's contribution to the tail latency drops from 8.94 ms (66%) to 1.32 ms (19%). Meanwhile, the root cause of tail latency has moved from GC to native code execution after applying

Table 5: Tail latency breakdowns (in ms) at 99.9 percentile before and after applying GC-Boost+Tuning to *Client Manager*.

	IO	IC	GC	JIT	Native	Total
Baseline	0.83	0.50	**8.94**	0	3.28	13.55
GC-Boost+Tuning	0.80	0.64	1.32	0	**4.01**	6.77

GC-Boost+Tuning. The shift in tail latency indicates there is no "silver bullet." GC-Boost and GC-Boost+Tuning are not the only solutions for the tail latency problem in *Node.js*. We leave it as future work to develop additional tail reduction techniques that target other components of *Node.js* servers.

8. RELATED WORK

We describe prior work in the context of themes: mechanisms to pinpoint the sources of tail latency, effects of garbage collection on tail latency, approaches to classify dependency relationships, and other generic event-driven research.

Pinpointing Sources of Tail Latency Identifying root causes of tail latency is an active research area. A common characteristic of prior art is the tendency to treat the application and its underlying runtime layer (*Node.js* in our case) as a black box and focus on system-level implications. For instance, Kasture et al. model servers as queuing systems and identify tails by statistically correlating response latency with queue length [21]. Li et al. further show that modeling a server as a simple monolithic queuing system is insufficient to understand the sources of tail latency [24]. They demonstrate the need to consider other system-level factors such as inter-process interference and process-to-core mapping. Zhang et al. design a load tester, Treadmill, to quantify tail latency in Facebook's servers [39]. Using Treadmill, they leverage regression techniques to determine the system-level root causes.

We take a different approach. We understand application-level impact on tail latencies (while not losing insights at the system level). Accounting for the application and its runtime behavior is particularly important for *Node.js* servers because of their inherent complexity, mixing event-driven execution with managed runtime-induced overheads. We show that gaining application-level understanding can empower system designers and application developers to diagnose tail latency issues at a finer granularity, such as garbage collection.

Garbage Collection in Tail Latency Garbage collection has long been an important research area [20]. Recently, GC has garnered interest in big data and cloud computing contexts as emerging large-scale distributed applications are increasingly developed in managed languages. Dean and Barroso pointed out that garbage collector activities in managed languages can lead to high tail latency [15]. Maas et al. demonstrated that distributed cloud services such as Apache Spark may suffer from uncoordinated GC pauses that hurt tail latency [27]. They further designed a holistic runtime system *Taurus* that coordinates GC pauses to reduce tail latency [26]. Terei et al. propose treating GC pauses as predictable failures and leverage existing failure recovery mechanisms to eliminate GC-related latency in distributed applications [35].

To the best of our knowledge, we are the first to quantify and remedy GC's impact on tail latency in *event-driven* servers. The GC-induced overhead is particularly detrimental to tail latency in event-driven servers where event execution is serialized and thus GC delay directly contributes to the end-to-end latency. Our analysis framework leverages event-specific knowledge, which none of the previous work is equipped with.

Capturing Event Relationships At the core of our tail latency analysis framework is the event dependency graph (EDG). The event dependency is a type of inter-event relation, but it differs from previously proposed inter-event relations in that an EDG is constructed *dynamically* while all previous proposals are based on *static* event relations. For example, Madsen et al. proposed an inter-event relation called *event call graph*, which is statically constructed and it is used to detect bugs in event-driven applications (e.g., incorrectly registered callbacks), performed through static analysis [28].

Our formalization of event dependency as a happens-before relation has similarity to recent work on race detection in event-driven applications [31, 32, 16], which also define a happens-before relation. However, the two definitions have different semantics. Happens-before in event race detection focuses on capturing the read and write behaviors of events in order to detect races, whereas our happens-before definition captures the event registration and triggering sequence that let us measure the critical path latency of an HTTP request.

Event-driven Optimizations Event-driven programming has long existed in highly concurrent servers [30, 18]. It has been optimized both at the system and architecture level.

At the system level, Welsh et al. improved server performance and resource efficiency by combining threads and event-driven programming [36]. Ogasawara characterized runtime behaviors of *Node.js* applications and also found that on average little time is spent on GC and inline cache miss handling [29]. At the architecture level, Zhu et al., identified the microarchitectural bottlenecks of *Node.js* applications and exploited heavy inter-event code reuse to improve front end efficiencies [40]. Larus et al. developed *cohort scheduling* to improve data and code locality by consecutively executing similar events on the same processor [23].

Prior work improves the overall performance of event-driven servers and is complementary to our contributions. For example, one could first leverage cohort scheduling to reduce the overall latency of both tail and non-tail requests, and then apply our techniques to remedy excessive tails.

9. CONCLUSION

The Event Dependency Graph enables direct, precise, and fine-grained latency characterization with respect to the event-driven programming model and the managed language runtime system. Leveraging EDG, we show that tail latencies on *Node.js* applications mainly arise from the CPU execution overhead, in which garbage collection constitutes almost 50% of the time. We thus devise a GC-oriented turbo boosting technique which significantly improves the tail latency of *Node.js* applications with negligible energy overhead and no modification to applications. We expect our results and technique to be readily applicable to production-scale *Node.js* cloud services.

References

[1] Client Manager. https://github.com/alessioalex/ClientManager.

[2] Etherpad Lite. https://github.com/ether/etherpad-lite.

[3] Exclusive: How linkedin used node.js and html5 to build a better, faster app. http://venturebeat.com/2011/08/16/linkedin-node/.

[4] How node.js powers the many user interfaces of netflix. http://thenewstack.io/netflix-uses-node-js-power-user-interface/.

[5] How we built ebay's first node.js application. http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/.

[6] Let's Chat. https://github.com/sdelements/lets-chat.

[7] Lighter. https://github.com/mehfuzh/lighter.

[8] New node.js foundation survey reports new "full stack" in demand among enterprise developers. https://nodejs.org/en/blog/announcements/nodejs-foundation-survey/.

[9] Node benchmarks. https://github.com/nodebenchmark/benchmarks.

[10] Papi 5.4.3 release. http://icl.cs.utk.edu/papi/software/view.html?id=245.

[11] Todo. https://github.com/amirrajan/nodejs-todo.

[12] wrk2: a HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2.

[13] Wonsun Ahn, Jiho Choi, Thomas Shull, María J Garzarán, and Josep Torrellas. Improving javascript performance by deconstructing the type system. In *Prof. of PLDI*, 2014.

[14] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.Js in Action*. Manning Publications Co., 2013.

[15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 2013.

[16] Chun-Hung Hsiao, Cristiano L. Pereira, Jie Yu, Gilles A. Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *Proc. of PLDI*, 2014.

[17] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proc. of HPCA*, 2015.

[18] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. High performance web servers on windows nt: Design and performance. In *Proc. of USENIX Windows NT Workshop*, 1997.

[19] José A. Joao, Onur Mutlu, and Yale N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proc. of ISCA*, 2009.

[20] Richard Jones and Rafael D Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[21] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proc. of MICRO*, 2015.

[22] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proc. of ASPLOS*, 2014.

[23] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *Proc. of USENIX ATC*, 2002.

[24] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proc. of Cloud Computing*, 2014.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proc. of ISCA*, 2014.

[26] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proc. of ASPLOS*, 2016.

[27] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proc. of the HotOS*, 2015.

[28] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proc. of OOPSLA*, 2015.

[29] Takeshi Ogasawara. Workload characterization of server-side javascript. In *Proc. of IISWC*, 2014.

[30] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proc. of USENIX ATC*, 1999.

[31] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proc. of PLDI*, 2012.

[32] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proc. of OOPSLA*, 2013.

[33] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[34] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. of NSDI*, 2015.

[35] David Terei and Amit A. Levy. Blade: A data center garbage collector. *CoRR*, 2015.

[36] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. of SOSP*, 2001.

[37] Matt Welsh, Steven D Gribble, Eric A Brewer, and David Culler. A design framework for highly concurrent systems. In *TR UCB/CSD-00-1108*, 2000.

[38] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proc. of NSDI*, 2013.

[39] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proc. of ISCA*, 2016.

[40] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proc. of MICRO*, 2015.