Tail Latency in Node.js: Energy Efficient Turbo Boosting for Long Latency Requests in Event-Driven Web Services

Wenzhi Cui*
The University of Texas at Austin
Google
USA

Daniel Richins
The University of Texas at Austin
USA

Yuhao Zhu†
The University of Texas at Austin
University of Rochester
USA

Vijay Janapa Reddi†
The University of Texas at Austin
Harvard University
USA

Abstract

Cloud-based Web services are shifting to the event-driven, scripting language-based programming model to achieve productivity, flexibility, and scalability. Implementations of this model, however, generally suffer from long tail latencies, which we measure using *Node.js* as a case study. Unlike in traditional thread-based systems, reducing long tails is difficult in event-driven systems due to their inherent asynchronous programming model. We propose a framework to identify and optimize tail latency sources in scripted event-driven Web services. We introduce profiling that allows us to gain deep insights into not only how asynchronous event-driven execution impacts application tail latency but also how the managed runtime system overhead exacerbates the tail latency issue further. Using the profiling framework, we propose an event-driven execution runtime design that orchestrates the hardware's boosting capabilities to reduce tail latency. We achieve higher tail latency reductions with lower energy overhead than prior techniques that are unaware of the underlying event-driven program execution model. The lessons we derive from *Node.js* apply to other event-driven services based on scripting language frameworks.

*Work was done entirely at The University of Texas at Austin.
†Work was done in part at The University of Texas at Austin.

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313823

CCS Concepts • **Software and its engineering** → **Runtime environments**; **Application specific development environments**; • **Information systems** → *Web applications*; • **Hardware** → *Power estimation and optimization.*

Keywords Node.js, tail latency, event-driven, JavaScript

ACM Reference Format:
Wenzhi Cui, Daniel Richins, Yuhao Zhu, and Vijay Janapa Reddi. 2019. Tail Latency in Node.js: Energy Efficient Turbo Boosting for Long Latency Requests in Event-Driven Web Services. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3313808.3313823

1 Introduction

Cloud-based Web services are undergoing a transformation toward server-side scripting in response to its promises of increased productivity, flexibility, and scalability. Companies such as PayPal, eBay, GoDaddy, and LinkedIn have publicly announced their use of asynchronous event-driven scripting frameworks, such as *Node.js*, as part of the backends for their Web applications [3, 4, 7]. The use of the asynchronous event-driven execution model coupled with the managed scripting language leads to better application scalability and higher throughput, developer productivity, and code portability, which are essential for Web-scale development [41].

A key challenge facing Web-scale deployment is long *tail latency*, and asynchronous event-driven Web services are no exception. Long tail latency is often the single most important reason for poor user experience in large scale deployments [15]. We use the popular *Node.js* framework for our study of tail latency in managed event-driven systems. *Node.js* is a JavaScript-based, event-driven framework for developing responsive, scalable Web applications. We find that tail latency for requests at the 99.9^{th} percentile is about 10× longer than those at the 50^{th} percentile, indicating that serious tail latency issues exist in event-driven servers.

No prior work addresses the tail latency problem from the event-driven programming paradigm. Prior work addresses server-side tail latency by improving request-level parallelism in request-per-thread Web services [16, 17, 22, 43] or by relying on the implicit thread-based programming model [19, 25, 26]. Other work tends to treat the application and its underlying runtime as a black box, focusing on system-level implications or network implications [10, 15, 27, 30, 44].

Asynchronous event-driven Web services pose a unique challenge for identifying tail latency. Their event-driven nature makes it difficult to reconstruct a request's latency and identify the performance bottleneck because a request's lifetime is split into three components—*I/O*, *event callback execution*, and *event queuing*. The three components of *all* requests are interleaved in a single thread, posing a challenge in piecing together these independently operating components into a single request's latency. In contrast, in a conventional thread-per-request server, each request is assigned a unique thread, which processes each request sequentially and independently of all others. As such, queuing, event execution, and I/O time can be measured more directly [29].

Moreover, in recent years, asynchronous event-driven Web services have been increasingly developed using managed runtime frameworks. Managed runtimes suffer from several well known, sometimes non-deterministic, overheads, such as inline caching, garbage collection, and code cache management [38]. While there are studies focused on the performance implications of managed-language runtimes in general, *how tail latency is affected by the binding of event-driven execution and managed runtimes remains unknown.*

In this paper, we present the first solution to mitigate tail latency in asynchronous event-driven scripting frameworks, using the production-grade *Node.js* event-driven scripting framework. We focus not only on reducing tail latency but on doing so at minimal energy cost, which is closely aligned with the total cost of ownership in data centers [11]. We demonstrate that only by understanding the underlying event-driven model, and leveraging its execution behavior, is it possible to achieve significant tail reductions in managed event-driven systems with little energy cost.

The insight behind our approach to mitigate tail latency in asynchronous event-driven scripting is to view the processing of a 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 corresponds to the latency of a request. By tracking vital information of each event, such as I/O time, event queue delay, and execution latency, the EDG lets us construct the critical path for request processing, thus letting us attribute tail latency to fine-grained components.

We use the EDG profiling framework to show that *Node.js* tail requests are bounded by CPU performance, which is largely dominated by garbage collection and dynamically generated native code. To address these tail latency sources, we first propose GC-Boost, which accelerates GC execution via frequency and voltage boosting. GC-Boost reduces tail latency by as much as 19.1% with negligible energy overheads.

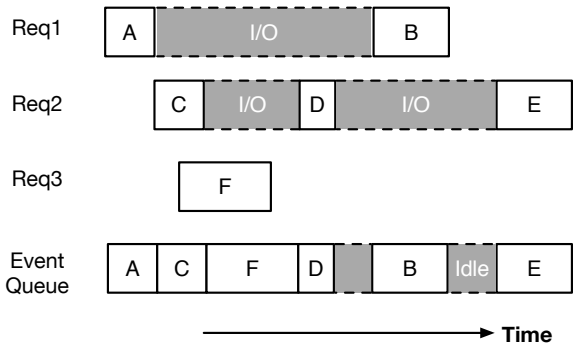


Figure 1. Event-driven execution overlaps compute and I/O execution. While waiting for I/O to complete for a request, the event loop is free to process events from unrelated requests. In traditional thread-per-request servers, each request is assigned a unique thread wherein a request's queue, execution, and I/O time can be directly measured. However, queue time is implicit in event-driven servers.

We also show an orthogonal technique called Queue Boost that operates by boosting event callback execution in the event queue. In combination, GC-Boost and Queue Boost outperform state-of-the-art solutions [19, 25] by achieving higher tail reduction with lower energy overhead, providing Web service operators a much wider trade-off space between tail reduction and energy overhead. GC-Boost, Queue Boost, and their combination also pushes the Pareto optimal frontier of tail latency towards a new level unmatched by prior work.

In summary, we make three major contributions:

- We present the insight that understanding the underlying programming model is important to effectively pinpoint and optimize the sources of tail latency.
- We propose the Event Dependency Graph (EDG) as a general representation of request processing in event-driven, managed language-based applications, which we use to design a tail latency optimization framework.
- Using the EDG, we show that the major bottleneck of tail latency in *Node.js* is CPU processing, especially processing that involves the JavaScript garbage collector and the runtime handling of the event queue.
- We propose, design, and evaluate two event-based optimizations, GC-Boost and Queue Boost, that reduce tail latency efficiently with little energy overhead, both offering significant improvements over prior approaches that are agnostic to the event-driven model.

The paper is organized as follows. Section 2 provides brief background on the event-driven programming model and its advantages over the thread-per-request model. Section 3 gives an overview of the baseline *Node.js* setup and the tail optimization system we propose and introduces the high-level

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 [8]	Redis	40K	Online task management tool, similar to the Google Tasks service. Users create new tasks, update tasks, and permanently remove outdated tasks.
Lighter [6]	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 [5]	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.

concepts for the sections that follow. Section 4 presents the design and implementation of our EDG-based latency analysis framework. Section 5 applies our technique to *Node.js* workloads to pinpoint the sources of tail latency. Section 6 introduces our optimizations. Section 7 presents an evaluation of the performance and energy overheads for our system and compares our techniques against alternatives. Section 8 discusses prior work and Section 9 concludes the paper.

2 Background and Motivation

Server applications have traditionally employed a "thread-per-request" execution model where each incoming user request is dispatched to a different OS thread to increase system concurrency. Instead of dedicating a thread to each request, the event-driven programming model translates incoming requests into events, each associated with a callback. Event callbacks are pushed into a FIFO event queue 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 to be processed.

Figure 1 illustrates the case where 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 improves 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 servers generally achieve greater scalability than thread-per-request ones [41, 42].

The challenge in reducing tail latency in the event-driven execution model is identifying the sources of tail latencies. Deriving tail latency bottlenecks is challenging for two reasons. First, a request is broken into separate *independently* operating events, which will be interleaved with events in other requests. Second, the end-to-end request latency can stem from any of the three *asynchronous* components: I/O, queuing, and event execution. Thus, determining a request's latency in event-driven servers requires us to reconstruct the execution time of portions of the request across all three independently operating components within one main thread.

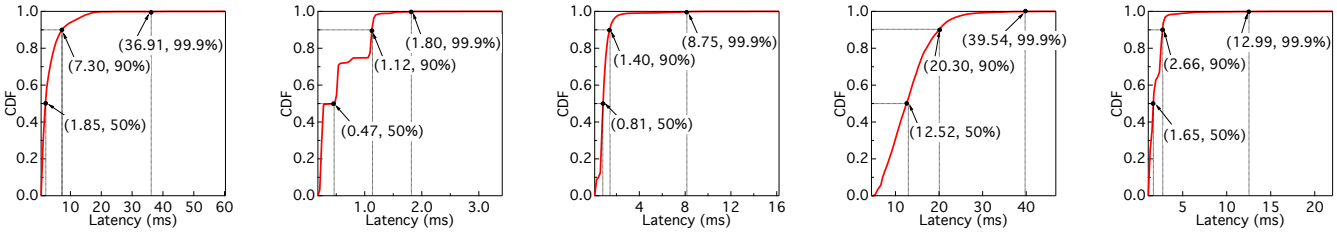
3 System Overview

We introduce an optimization framework to identify the source of tail latency in event-driven Web services and guide optimizations to mitigate tail bottlenecks and thus reduce tail latency. We use *Node.js* as the event-driven framework, since it is widely adopted, and build our optimizations into it. We present the details of our experimental setup (Section 3.1) and then give an overview of how the optimization framework operates and build it into *Node.js* (Section 3.2). Subsequent sections provide additional details for each of the stages.

3.1 *Node.js* Baseline

We use industry-strength event-driven server applications from the NodeBench application suite [46, 47], as well as new applications that encapsulate additional server functionalities. Table 1 describes the applications we study. The applications cover various domains and exercise different types of I/O including file, 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.

We use *wrk2*, an HTTP load testing tool featuring parameter tuning and dynamic request generation [39]. To realistically mimic users interacting with each application, we use 4 to 10 different types of HTTP requests (depending on each application's functionalities) and generate over 10,000 requests per application. We discard warm-up behavior.



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

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

We deploy the applications on a high-performance quad-core/eight-thread Intel Core i7 processor with 32 GB DRAM and a 240 GB SSD to deliver high I/O performance. Our work is more affected by the clock frequency of the core(s) than the number of cores, since *Node.js* is predominantly a single-threaded application. We set the processor's frequency to 2.6 GHz to prevent tails caused by "floating" frequencies [30]. We simulate our clients and host our database services on separate server-class machines connected through 1 Gbps intranet, isolating event-driven compute from I/O services.

Figure 2 shows the server-side latency cumulative distribution functions (CDFs) for each application. Tail latency is a serious issue in each. The CDF curves rise very quickly initially, but as the curves approach the top, the slopes diminish and the curves become long, nearly flat lines, indicating a significant difference between the longest latencies and the vast majority. Any point in each graph $< x, y >$ reads as follows: $y\%$ of the total requests have a latency below x ms, i.e., the y^{th} percentile tail is x ms. We label three points: 50^{th}, 90^{th}, and 99.9^{th} percentile. On average, 99.9^{th} percentile request latency is 9.1× longer than 50^{th} percentile latency.

3.2 *Node.js* Tail Latency Optimization Framework

Figure 3 shows an overview of the optimization system. It consists of three stages: 1) constructing the event dependency graph (EDG), which reveals the critical path of a tail request; 2) leveraging the EDG to identify the system components that contribute the most to the tail latency; and 3) improving the performance of the bottleneck in order to reduce tail latency. The first two stages are performed in non-user facing systems as EDG construction incurs a small overhead, which we discuss later. The third stage improvements do not rely on real-time EDG construction and are applied to all servers.

Tail Request Reconstruction This stage (Section 4) identifies the critical path of any incoming request by constructing an event-dependency graph (EDG). Each node in an EDG represents a particular event in a request, and an edge in the EDG represents the causal relationship between two events. Each node is also annotated with timing information.

Tail Latency Bottleneck Analysis The constructed EDG from the previous stage is fed into this stage (Section 5), which identifies the event critical path of a request and generates a latency profile. The latency profile quantitatively attributes the request latency to the major components in event-driven execution, including I/O, queuing, and event handler execution. Event handler time could further be decomposed into various components such as native code execution, just-in-time compilation, garbage collection, etc., all of which could potentially be tail latency bottlenecks.

Tail Latency Optimization The generated profile is sent to the backend (Section 6 and 7) to mitigate and optimize the tail sources. Among many optimization alternatives, we choose to focus on leveraging the turbo-boosting capability of server processors. Turbo Boost is a widely-used technique to increase performance and it has been extensively used to reduce tail latency in traditional servers [19, 25]. Blindly increasing processor frequency, however, comes at a high energy penalty. Our optimizer leverages the tail latency profile generated from the analysis stage to intelligently apply turbo boosting only to lucrative application phases, resulting in significant tail reduction with almost negligible overhead.

4 Tail Latency Reconstruction

We present a direct, precise, and fine-grained way to pinpoint sources of tail latency in event-driven systems. We first introduce a breakdown model for request latency in event-driven servers to provide a guideline for attributing sources of tail latency (Section 4.1). We then describe a new runtime analysis technique called Event Dependency Analysis, which is based on dynamically generated EDGs for performing latency analysis that we implemented directly in *Node.js* so that applications require no code changes (Section 4.2). We show that the profiling overhead is negligible (Section 4.3). Finally, we validate our EDG analysis approach (Section 4.4).

4.1 Request Latency Breakdown

Figure 4 intuitively shows the hierarchy of the latency components 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 times are further broken down by the model to provide application-level insights, specifically in a manner that

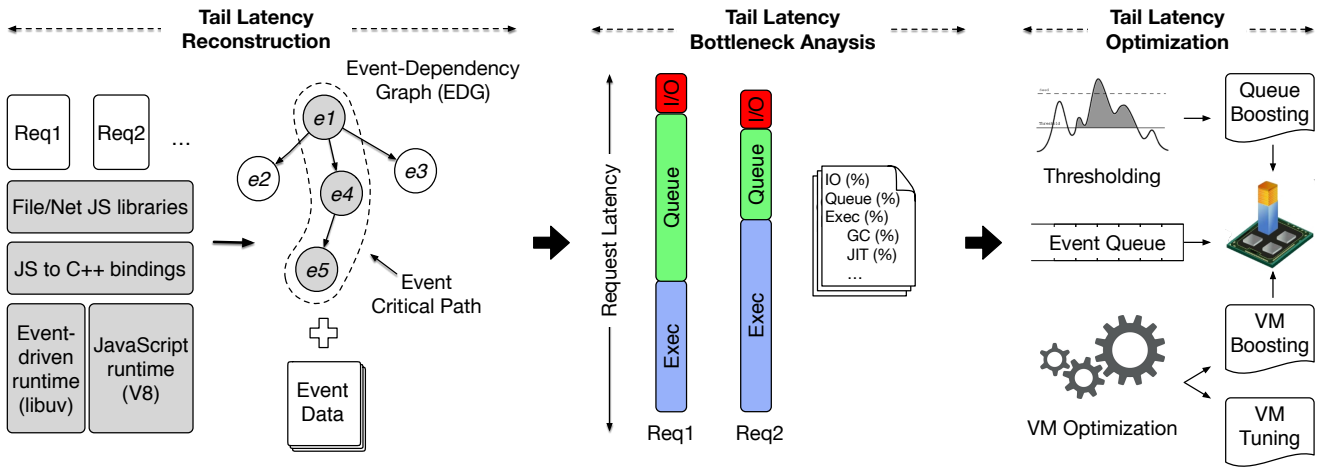


Figure 3. Overview of our tail analysis and optimization framework for event-driven Web services. The shaded parts indicate the components that are modified in our *Node.js*-based implementation to construct the EDG to identify request bottlenecks. The optimization stage uses the EDG breakdown to dynamically determine what optimization(s) to perform on individual requests.

can provide insights into the managed language components. We now explain the latency model details.

A request in event-driven servers is a sequence of events. 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.

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

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:

$$T(e_i) = IO(e_i) + Queue(e_i) + Exec(e_i) \quad (2)$$

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 queuing 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 on sources of tail latencies, 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 these four components because they have been shown to be responsible for the majority of execution cycles in JavaScript-based applications [9, 34]. 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.

$$Exec(e) = Native(e) + JIT(e) + GC(e) + IC(e) \quad (3)$$

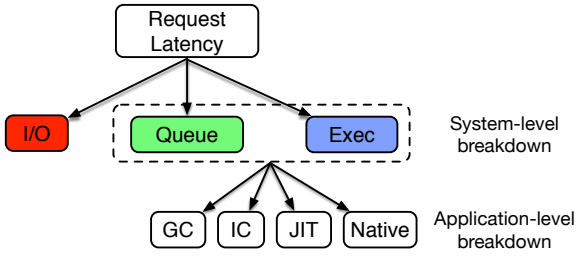


Figure 4. The hierarchical latency breakdown that allows us to precisely identify the bottleneck in event-driven applications.

Equations 1, 2, and 3 together form a model for request latency that is sufficiently fine-grained to capture the system- and application-level components and explain tail requests.

4.2 Event Critical Path

One of the challenges of using the EDG is that one parent event may spawn multiple events simultaneously but only events on the critical path can influence the end-to-end latency. To use the analytical model for analyzing tail latencies, 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 $ECP(R)$ encodes the latency of a given request. The key to tracking the event critical path is to identify the dependencies between events, from which we construct an event dependency graph (EDG) where each node in the graph corresponds to an event and each edge 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.

Event Dependency Graph (EDG) Construction Table 2 lists the 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. 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 event dependency definition, 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. We record the event dependency in a hash map and dump it to disk once a minute.

Event Data Besides the event dependencies, we also record important timestamps for each event that help track the event critical path: 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 and end times refer to the time an event callback starts and finishes execution, respectively. According to the timestamps, the components in Equation 2 can be derived as follows:

$$IO(e) = Ready - Register$$
$$Sched(e) = Begin - Ready$$
$$Exec(e) = End - Begin$$

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
Miscellaneous	source	the first event in a request
	sink	the event that sends a response
	idle	user-defined low-priority events

Logging timestamps is mostly trivial except for the ready time, which is problematic because I/O events are first observed by the OS kernel and only later become visible to the user space 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, there is a difference between when an event is truly ready (in kernel space) and when it becomes visible to the *Node.js* runtime (in user space). In practice, this difference can be large.

To track an event's ready time, a helper thread periodically polls at a much finer granularity than *Node.js*'s default polling mechanism. The helper thread introduces only negligible overhead on both the average and tail latency. This is because the polling thread uses the common Linux `epoll()` API to check for events. When there are no events, the polling thread will be inactive instead of idly spinning. As a result, the polling overhead will only be proportional to the number of events. Also, *Node.js* applications are mostly single-threaded (backed by a few worker threads), and there are abundant CPU resources reserved for the helper thread.

4.3 Profiling Overhead

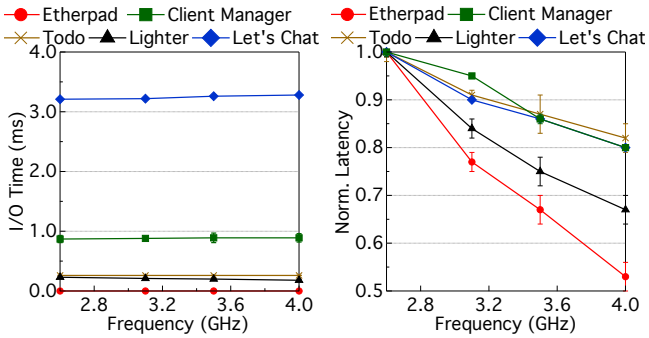
EDG construction has a performance penalty. Note, however, that constructing the EDG is performed *only at the initial warm-up stage of a long-running Web service*, similar to JIT compilation. Once the profiling results are gathered, profiling is not triggered for the rest of application execution. Thus, profiling overhead has very little impact on performance.

There are two overheads: event instrumentation and the polling thread. Event instrumentation consists of recording event timestamps and tracking event dependencies, both of which are done transparently inside the JavaScript VM without application involvement. Table 3 shows the overheard for the individual applications. Overall, the average overhead is 6.69% for non-tail requests and 6.77% for tail requests.

In practical deployment, we expect that the EDG construction and analysis are performed on a dedicated system where the impact on service performance is not evident (i.e., the system is not servicing real requests, but it is exposed to the incoming queries). We provide a switch to easily turn off

Table 3. EDG profiling overhead. The overhead is calculated by comparing the average server-side latency between when EDG is enabled and that when the EDG is disabled. All requests are classified into tail and non-tail when EDG is disabled.

Application	Non-tail	Tail
Etherpad Lite	3.34%	3.53%
Todo	8.31%	6.9%
Lighter	4.3%	4.6%
Let's Chat	8.45%	8.1%
Client Manager	9.05%	10.72%



(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.

instrumentation and polling threads should a user choose to use EDG profiling periodically without restarting the *Node.js* server. Because it is unlikely to profile for all the possible user requests ahead of time, we expect that the system administrator will periodically perform profiling online to adjust the EDG profile to realistic user requests in time.

The requests we use to profile and construct the EDG in the system are independent of the requests we use in evaluation. In our analysis, we found that the profile information was sufficient to construct the EDG fully. We re-profile in the event the profiled requests triggers an incomplete EDG.

4.4 Validation

It is important to validate that the root-cause analysis approach we propose is indeed correct. However, it is difficult to directly validate the 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 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. The application does not trigger any external I/O as we show via a detailed I/O time breakdown later.

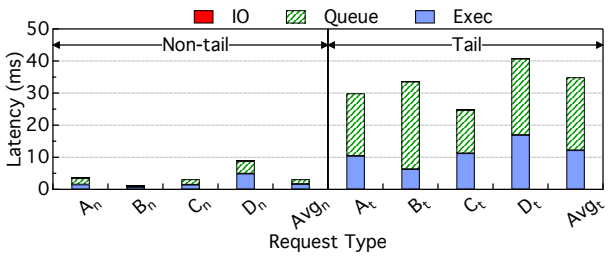
5 Tail Latency Analysis

We use the EDG framework to dissect the request latencies to determine the critical bottlenecks in event-driven server-side applications. We dissect the bottlenecks at the system-level (Section 5.1) and the application-level (Section 5.2), progressively going deeper into the bottleneck sources as per Figure 4. Finally, we show that our analysis results hold true for scale-out *Node.js* workloads as well (Section 5.3).

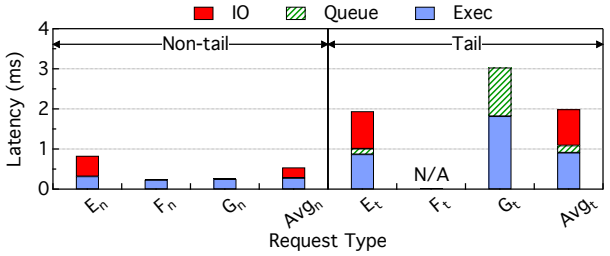
5.1 System-Level Tail Latency Analysis

We remind the reader that system-level request latency consists of three components: I/O, queuing, and event execution. Figure 6 shows the 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. 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. Though there are few request URLs per application, there are many dynamic instances (i.e., clients) of a request type throughout an experiment. Each bar shows the average latency for all instances of a request type over 10 runs. For comparison, 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 throughout this experiment.

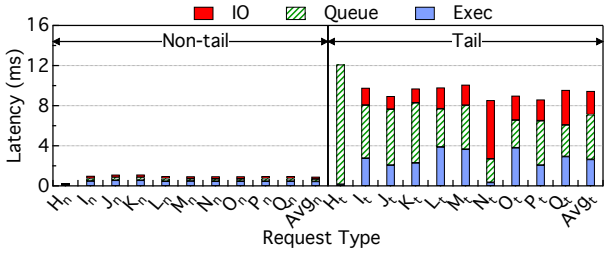
We find that tail latencies of event-driven applications are predominantly caused by event callback execution and the queuing overhead rather than I/O time. Across all of the applications, the average queuing time contributes about 45.8% of the tail latency and the average callback execution time contributes 37.2% of the tail latency. The significance of queuing and callback execution time indicates that tail latencies in event-driven applications are bottlenecked primarily by CPU computations instead of long latency I/O operations.



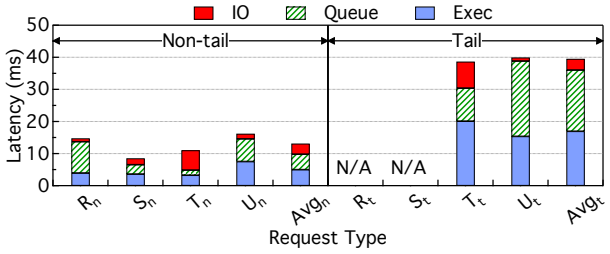
(a) Etherpad Lite



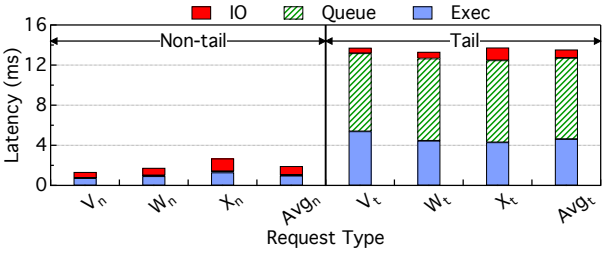
(b) Todo



(c) Lighter



(d) Let's chat



(e) Client Manager

Figure 6. 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 deemed as long latency requests.

Table 4. 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%

Conversely, I/O time only contributes 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. To remedy the I/O-induced tail latency issues, one could optimize the network stack or database operations. Though we leave such optimizations to future work, it is the EDG that let us identify the bottleneck.

5.2 Application-Level Tail Latency Analysis

The EDG allows us to go a level deeper, as shown in Figure 4, and break down the queuing and execution time into four application-level components, including the JIT, GC, IC miss handling, and native code, all of which are potentially major sources of runtime overhead contributing to tails (as discussed in Section 4.1). Table 4 shows the breakdown for each application. The table shows the data across all of the components during the non-tail and the tail requests.

We make two key observations. First, the results show 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 constitute about 48.7% and 47.4% of the processor time on average, respectively. Native code contributes heavily to the execution time because it contains the functionality of an application. Garbage collection, on the other hand, is an artifact of the event-driven framework using JavaScript.

5.3 Scale-Out Analysis With *Node.js* Cluster

We also applied our profiling technique to *Node.js* applications in cluster mode to characterize the tail latency of

Table 5. Tail latency breakdown on *Node.js* clusters, showing the percentage of I/O, IC, GC, JIT, and native code time in tail requests when *Node.js* servers are executed in cluster mode.

Application	I/O	IC	GC	JIT	Native
Etherpad Lite	0%	0.4%	66.9%	0%	28.7%
Todo	10.9%	1.4%	32.2%	0%	55.5%
Lighter	2.4%	3.3%	56.9%	0%	37.3%
Let's Chat	20.3%	3.3%	47.3%	0%	29.1%
Client Manager	11.4%	1.9%	60.1%	0%	26.6%

scale-out workloads. In cluster mode, one *Node.js* master process will spawn multiple slave processes to handle incoming requests concurrently. The number of spawned processes is usually the same as the number of cores of the machine (including simultaneous multithreading), e.g., eight slave processes on our testbed. Each spawned process will have its own event queue and JavaScript VM. By default, incoming HTTP requests are assigned to slave *Node.js* processes in round-robin style. Table 5 shows the tail latency breakdown for all five workloads in cluster mode.

Our key observation on scale-out *Node.js* servers is that GC and native code execution are still the two major bottlenecks for tail latency. The GC bottleneck is more severe than in the single instance case, constituting 52.4% of overall tail latency compared to 40% in single instance mode. The major reason is that all the spawned *Node.js* instances will share the memory; therefore each VM has less available memory, and thus tail requests experience more frequent GC events.

6 Tail Latency Optimization

As the final step in our framework (Figure 3), we present techniques that target the major contributors to tail latency; these techniques can be applied to a production environment without the profiling overhead of enabling EDG generation. We describe the optimizations aimed at reducing overheads associated with the managed runtime, specifically involving garbage collection (Section 6.1). We then discuss a complementary technique that improves the event callback execution performance using event queue studies (Section 6.2).

6.1 VM Optimization

The first of the optimization schemes we propose targets the managed runtime specifically with the goal of alleviating tails caused by the garbage collector (GC), as GC has been identified as a major source of tail latency in Section 5.2.

Google's V8 JavaScript Engine uses a "stop-the-world" GC implementation, meaning the GC's execution is serialized with respect to the application. Therefore, GC blocks event queue processing whenever it triggers, and as such we want to remove it as a head-of-line "event." A variety of techniques have been proposed to improve GC performance (e.g., concurrent GC and hardware GC accelerator [23]), however most of them require compiler/hardware modifications.

Our goal is to leverage frequency boosting to improve GC performance without introducing significant changes so that a solution can be readily deployed in existing *Node.js* deployments. Frequency boosting is a good fit for reducing GC time for two important reasons. First, GC's average instructions-per-cycle (IPC) is moderately high at 1.3 (measured with hardware performance counters using PAPI [21]), suggesting that GC is compute-bound and can benefit from frequency boosting. Second, GC contributes only a small fraction of the non-tail requests. Table 4 shows the latency breakdown for non-tail requests. GC has little to no impact on non-tail requests, suggesting that a higher clock frequency during GC would significantly improve the tail latency with minimal impact on the overall energy consumption.

In addition, we carefully tune the GC-related parameters, such as heap sizes and thresholds, to study the upper-bound of performance improvement with frequency boosting. Tuning application/system parameters, such as TCP/IP configuration and CPU affinity [14], is a common practice in deploying server applications. The V8 JavaScript engine uses a generational garbage collector [24], which organizes the heap into two spaces (generations): a new-space and an old-space. We focus on the new-space and old-space heap sizes as the two key tuning parameters after observing that GC performance in *Node.js* is most sensitive to the sizes of the two heaps.

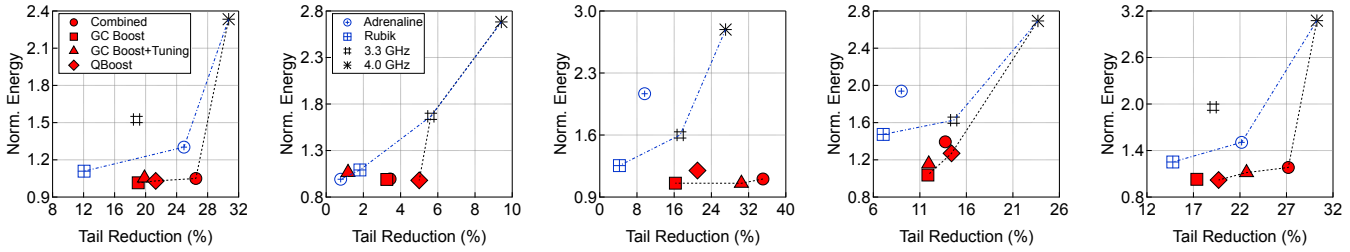
Our results indicated that a smaller new-space and moderately large old-space can reduce tail latency most because it will make GC pauses shorter but more frequent, i.e., more requests will experience GC but each time GC finishes faster. As a result, combining GC-Tuning and GC-Boost as a further optimization can yield greater tail reduction.

6.2 Event Queue Optimization

As seen in Figure 6, time spent in GC is not the only contributor to tail latency; time spent waiting in the event loop queue is another major cause of tail requests. We therefore propose another, complementary optimization scheme to address long queue wait times. Queue waiting times may arise when the event loop is especially busy executing the callback functions of earlier events or when the event queue experiences head-of-line blocking, i.e. the queue is blocked by an event taking an extraordinarily long time to process.

For our optimization, we address both causes of an overly busy event queue. To determine if the queue is busy, we use two metrics: event queue length and head processing time. Event queue length refers to the number of events in the queue at a given point of time, excluding the currently executing event. Head processing time refers to the time that has been spent on the current head-of-queue event.

We periodically check whether queue length or head processing time is larger than some threshold. If the condition is satisfied, *Node.js* automatically boosts the processor to maximum frequency. We check event queue busyness every microsecond. We verified that the overhead is negligible.



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

Figure 7. Tail reduction and energy consumption for GC-Boost, GC-Boost (after Tuning), Queue Boost, and the combined scheme. Results are normalized to statically running the system at 2.6 GHz. We compare against four baselines: Rubik, Adrenaline, and statically running at 3.3 GHz and 4.0 GHz. Each figure shows the Pareto optimal frontiers with and without the proposed techniques. Our techniques push the Pareto optimal frontier to a new level, providing better system operating choices that are more energy-efficient with better tail latency.

Rather than use a static threshold, we dynamically decide the threshold. We calculate the threshold for the event queue length by computing the average event queue length on the fly and scaling it by a constant factor. The constant factor is a hyperparameter that can be set by an end user through an environmental variable. Similarly, we use another hyperparameter for head processing time to calculate the threshold. In practice, we discovered that a hyperparameter between 2.0 and 3.0 can achieve good performance. This is consistent with the intuition that the tail portion is at least 2.0 longer than the median value in a normal distribution.

7 Evaluation

Our event-oriented optimizations (Section 7.1) outperform thread-based solutions by achieving higher tail reduction with lower energy overhead (Section 7.2). Also, our comprehensive set of solutions strictly Pareto-dominate prior work and offer alternative trade-offs (Section 7.3). Furthermore, we consider cluster level development of *Node.js* and study the effects of global versus per-core DVFS (Section 7.4). Finally, we summarize our results and point out new optimization directions for event-driven Web services (Section 7.5).

7.1 Energy-Efficient Tail Boosting

We implemented the frequency boosting technique as a user space scheduler integrated into V8, which requests that the kernel set the CPU frequency to a boost value on demand.

Boost Server processors operate at a nominal frequency ranging from 2 GHz to 3 GHz. Our server can reach a maximum boost frequency of 4.0 GHz. Our framework dynamically increases clock frequency from 2.6 GHz to 4.0 GHz whenever GC starts orevent queue thresholds are detected.

Per-core DVFS Our hardware platform does not support per-core DVFS. Thus, all cores are boosted in the current implementation. This implementation decision does *not* provide artificial benefits to our evaluation because event execution and GC happen in the same thread as the event loop. Boosting multiple cores has the equivalent performance as boosting only one core. However, we note that as per-core DVFS using integrated voltage regulators (IVRs) [13] becomes more prevalent, our proposal can be directly applicable to platforms equipped with such capabilities.

We considered upgrading our system to the latest Intel Skylake processor for evaluating per-core DVFS. However, the integrated voltage regulator (IVR) support requires operating system level driver support. At the time of writing, these drivers are yet to be released for Linux and Windows.

Cluster Mode It is worth noting that our optimization is also applicable to *Node.js* cluster mode, where each *Node.js* instance runs on one core. As we have shown in Section 5.3, our findings regarding bottlenecks hold true in cluster mode. In fact, GC overhead contributes more to the tail latency in cluster mode than in single instance mode. Thus, our techniques also benefit the cluster mode for tail latency reduction.

Note that our own setup lacks per-core DVFS; boosting the frequency of one core boosts all cores' frequency. Thus, the energy-efficiency results reported for the cluster mode should be taken as the *lower-bound* of the gains that our technique could provide. With per-core DVFS capability, our technique is expected to be more energy-efficient.

7.2 Event-Based versus Thread-Based Solutions

We investigate tail latency reduction and energy overhead under GC-Boost, GC-Boost (after Tuning), Queue Boost, and the combination of all three (Figure 7). We also evaluate GC-Boost and Queue Boost against two recently proposed, state-of-the-art DVFS schemes: Adrenaline [19] and Rubik [25]. Both techniques aim to reduce tail latency in traditional, non-event-driven servers. All results are averaged across all the benchmark applications and are normalized to the results without boosting (i.e., statically fixed at 2.6 GHz). For comparison, we also evaluate tail latency reduction while setting the frequency statically to 3.3 GHz and 4.0 GHz, which gives us the limit for improvement if energy were not a concern.

Rubik Comparison The goal of Rubik [25] is to meet the tail latency target using minimal power. 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.

As Figure 7 shows, our optimizations consistently achieve better tail reduction (up to 8×) with lower energy overhead than Rubik across all five applications. In addition, Rubik misses the tail target for most applications (not plotted). Rubik has limited tail reduction capability in *Node.js* because its 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 true for event-driven applications where requests are divided into interleaved events.

Adrenaline Comparison Adrenaline [19] is motivated by the observation that certain request types have a higher probability of becoming tail requests and that they can be identified by indicators such as 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 peak frequency.

We find that Adrenaline achieves significant tail reduction in two out of five benchmarked applications (*Etherpad Lite* and *Client Manager*). However, it introduces much higher energy overheads than other alternatives except in the case of *Todo*. On average, Adrenaline costs 1.51× the energy of GC-Boost even as they achieve similar tail reductions.

The 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 our 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 lifetime. In contrast, GC-Boost accelerates only the GC phase guided by our EDG analysis. Also, the original Adrenaline relies on a fine-grained, per-core DVFS mechanism which is not available on our platform.

7.3 Pareto Frontier Analysis

We propose three more tail reduction techniques that form a new Pareto frontier. Figure 7 shows that GC-Boost, GC-Boost (after Tuning), Queue Boost, and Combined fall into or close to the Pareto frontier of the alternatives we consider.

GC-Boost We find that GC-Boost consistently improves the tail latency across different applications. The improvements are usually 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*. *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 execution has little impact on total energy consumption. GC-Boost introduces only a 2.8% energy overhead over the baseline 2.6 GHz. This is because GC contributes only about 3% of the overall request latency. In contrast, globally boosting CPU frequency to 3.3 GHz and 4.0 GHz introduces major energy overheads of 67.6% and 171.2%, respectively. We conclude that GC-Boost can significantly improve tail latency in an energy-efficient manner.

GC-Boost (after Tuning) We find GC-Tuning increases energy overhead since GC happens more frequently, especially for *Todo*, *Let's Chat*, and *Client Manager*. On average, it introduces 6% (up to 9%) extra energy overhead. But enabling GC-Tuning can reduce tail latency by up to 30.5% for *Lighter*, which is almost 2× better than GC-Boost alone. The average gains are less pronounced, with GC-Tuning reducing tail latency by only 3.67% over GC-Boost.

Queue Boost Queue Boost offers a slightly different trade-off than GC-Boost (after Tuning). It is most effective when GC is not a major bottleneck. On average, Queue Boost reduces tail latency by 16.3% with 10.4% more energy. Queue Boost achieves better tail reduction than GC-Boost for all applications. We believe this is because the overhead of GC is also manifested as queue anomalies. For example, if a request is suffering from GC, the event queue will detect head-of-line blocking, or a longer busy queue. Because our anomaly detection heuristic looks at overall queue activity, agnostic of what is causing the busyness, it will waste some energy on non-tail requests, explaining the higher energy overhead.

We also note that Queue Boost outperforms GC-Boost and GC-Boost (after Tuning) on *Todo*. We believe this is because the tail latency in *Todo* does not generally arise from GC. Since Queue Boost is more comprehensive than just GC, it succeeds at improving the tail latency of *Todo*.

Combined The combination of GC-Boost, GC-Tuning, and Queue Boost offers an optimal trade-off between tail reduction and energy overhead. The combined approach provides 21.18% tail reduction with 14% more energy. The best tail reduction for the combined optimization is 35.08% for *Lighter*, better even than continuously running at 4.0 GHz. This may be because *Lighter* is most sensitive to GC-Tuning.

7.4 Cluster Evaluation

We also evaluated GC-Boost and Queue Boost under cluster mode as shown in Table 6. Recall that in cluster mode, one *Node.js* master process spawns eight slave processes to handle incoming requests concurrently. Each process is a full instance of *Node.js* with its own event queue and VM.

On average, GC-Boost offers 12.7% tail reduction with 6% extra energy. Queue Boost also offers 13.5% tail reduction with 18.0% extra energy. Compared to the single process mode, our proposed techniques achieved slightly worse tail reduction with almost 2× more extra energy (GC-Boost cost 3% extra energy and Queue Boost cost 10% extra energy on

Table 6. Tail-reduction and energy overhead after applying GC-Boost and Queue Boost to *Node.js* servers in cluster mode. The baseline is the average tail latency and energy of all applications running with the same cluster settings without any boosting.

Application	Tail Reduction(%)		Norm. Energy	
	GC-Boost	Queue Boost	GC-Boost	Queue Boost
Etherpad Lite	15.8	25.0	1.08	1.19
Todo	1.2	7.9	0.99	1.07
Lighter	24.9	13.9	1.06	1.28
Let's Chat	6.3	10.1	1.05	1.24
Client Manager	15.1	10.7	1.12	1.11

Table 7. Summary of all six 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-Boost (after Tuning)	17.23	1.09
Queue Boost	16.27	1.10
Combined	21.18	1.14
Rubik	7.99	1.22
Adrenaline	13.31	1.56

average). These numbers are worse than our own proposed solution, but they are on par with related work.

We also see a drop in energy efficiency when the number of slave processes goes from 2 to 4. We suspect that the drop is largely due to the lack of per-core DVFS. Even if one *Node.js* process in the cluster decides to boost frequency, all the cores' frequency will boost as a side effect, thus skewing the results. But with per-core results, similar to the assumptions made by prior work [19, 25], our energy efficiency results will be on par with the gains we have shown earlier.

7.5 Summary

We summarize our comparisons against GC-Boost in Table 7. It lists the average tail reduction and normalized energy for GC-Boost, GC-Boost (after Tuning), Queue Boost, Combined, Rubik, and Adrenaline. GC-Boost, GC-Boost (after Tuning), Queue Boost, and Combined significantly outperform Rubik and Adrenaline in both tail reduction and energy overhead. GC-Boost (after Tuning) achieves the maximum tail reduction (21% on average) with acceptable energy overhead (14%).

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.

Sources of Tail Latency Identifying sources of tail latency is an active research area. Prior work mainly focuses on identifying system-level sources of tail latency [25, 29, 44]. We take a different approach. We understand applica-tion-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.

Garbage Collection in Tail Latency Though garbage collection has long been an important research area [24], recently it has garnered interest in big data and cloud computing contexts as emerging distributed applications are increasingly developed in managed languages [15, 23, 31, 32, 40**?**]. As far as we know, 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, non-existent in previous work.

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 which differs from previously proposed inter-event relations in that an EDG is constructed *dynamically* while previous proposals are based on *static* event relations [33]. Our formulation of event dependency as a happens-before relation has similarity to recent work on race detection in event-driven applications [18, 36, 37], which also defines a happens-before relation. However, the two definitions have different semantics. Happens-before in event race detection captures the read and write behaviors of events to detect races; our definition captures the event registration and triggering sequence to measure the critical path latency.

Event-Driven Optimizations Event driven execution has long existed in highly concurrent servers [20, 35]. It has been optimized both at the system [34, 41] and architecture levels [12, 28, 47] and in the mobile computing space [45]. However, no prior work specifically targets the tail latency issue. They instead improve overall performance of event-driven servers and are thus complementary to our work. For example, one could first leverage cohort scheduling [28] to reduce the overall latency of both tail and non-tail requests and then apply our techniques to remedy excessive tails.

9 Conclusion

Prior tail optimizations that assume a request is bound to a thread cannot be directly applied to managed event-driven systems. And if applied, they incur a severe energy penalty. We have presented a novel approach to identify, understand, and optimize the sources of tail latency. We introduce the EDG framework to propose VM and event queue boosting optimizations that cater to the event-driven system's characteristics. Where prior solutions increase energy consumption by 22-56% in an event-driven system, our solutions reduce tails more aggressively with as little as 3% extra energy.

Acknowledgements

The work was supported by sponsorship from NSF under CCF-1619283 and support from Intel and Google.

References

[1] [n. d.]. Client Manager. https://github.com/alessioalex/ClientManager.
[2] [n. d.]. Etherpad Lite. https://github.com/ether/etherpad-lite.
[3] [n. d.]. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app. http://venturebeat.com/2011/08/16/linkedin-node/.
[4] [n. d.]. How We Built eBay's First Node.js Application. http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/.
[5] [n. d.]. Let's Chat. https://github.com/sdelements/lets-chat.
[6] [n. d.]. Lighter. https://github.com/mehfuzh/lighter.
[7] [n. d.]. New Node.js Foundation Survey Reports New "Full Stack" In Demand Among Enterprise Developers. https://nodejs.org/en/blog/announcements/nodejs-foundation-survey/.
[8] [n. d.]. Todo. https://github.com/amirrajan/nodejs-todo.
[9] Wonsun Ahn, Jiho Choi, Thomas Shull, María J Garzarán, and Josep Torrellas. 2014. Improving JavaScript performance by deconstructing the type system. In *Proc. of PLDI*.
[10] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. of NSDI*.
[11] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013).
[12] Sapan Bhatia, Charles Consel, and Julia Lawall. 2006. Memory-manager/Scheduler Co-design: Optimizing Event-driven Servers to Improve Cache Behavior. In *Proc. of ISMM*.
[13] Edward A. Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J. Lambert, Kaladhar Radhakrishnan, and Michael J. Hill. 2014. FIVR – Fully integrated voltage regulators on 4th generation Intel Core SoCs. In *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014*. 432–439.
[14] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. 2013. *Node.Js in Action*. Manning Publications Co.
[15] Jeffrey Dean and Luiz André Barroso. [n. d.]. The Tail at Scale. *CACM'13* ([n. d.]).
[16] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proc. of ASPLOS*.
[17] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proc. of MICRO*.
[18] Chun-Hung Hsiao, Cristiano L. Pereira, Jie Yu, Gilles A. Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. In *Proc. of PLDI*.
[19] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ronald G. Dreslinski. 2015. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *Proc. of HPCA*.
[20] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. 1997. High Performance Web Servers on Windows NT: Design and Performance. In *Proc. of USENIX Windows NT Workshop*.
[21] The Innovative Computing Laboratory (ICL). 2016. PAPI 5.4.3 release. http://icl.cs.utk.edu/papi/software/view.html?id=245
[22] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *Proc. of ASPLOS*.
[23] José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proc. of ISCA*.
[24] Richard Jones and Rafael D Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.
[25] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proc. of MICRO*.
[26] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads. In *Proc. of ASPLOS*.
[27] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking Network DMA. In *Proc. of HotOS*.
[28] James R. Larus and Michael Parkes. 2002. Using Cohort Scheduling to Enhance Server Performance. In *Proc. of USENIX ATC*.
[29] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proc. of Cloud Computing*.
[30] David Lo, Liqun Cheng, Rama Govindaraju, Luiz AndrÃĪ Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proc. of ISCA*.
[31] Martin Maas, Krste Asanović, , and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proc. of ISCA*.
[32] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proc. of ASPLOS*.
[33] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proc. of OOPSLA*.
[34] Takeshi Ogasawara. 2014. Workload characterization of server-side JavaScript. In *Proc. of IISWC*.
[35] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. Flash: An Efficient and Portable Web Server. In *Proc. of USENIX ATC*.
[36] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. In *Proc. of PLDI*.
[37] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proc. of OOPSLA*.
[38] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.
[39] Gil Tene. 2018. wrk2: a HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2
[40] David Terei and Amit A. Levy. 2015. Blade: A Data Center Garbage Collector. *CoRR* (2015).
[41] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. of SOSP*.
[42] Matt Welsh, Steven D Gribble, Eric A Brewer, and David Culler. 2000. A design framework for highly concurrent systems. In *TR UCB/CSD-00-1108*.
[43] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-grain Principled Borrowing from Latency-critical Workloads Using Simultaneous Multithreading. In *Proc. of USENIX*.
[44] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proc. of ISCA*.
[45] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. 2015. Event-Based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications. In *Proc. of HPCA*.
[46] Yuhao Zhu, Daniel Richins, Wenzhi Cui, Matthew Halpern, and Vijay Janapa Reddi. 2016. Node Benchmarks. https://github.com/nodebenchmark/benchmarks
[47] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural implications of event-driven server-side web applications. In *Proc. of MICRO*.