Microarchitectural Implications of Event-driven Server-side Web Applications

Yuhao Zhu Daniel Richins Matthew Halpern Vijay Janapa Reddi

The University of Texas at Austin, Department of Electrical and Computer Engineering
{yzhu, drichins, matthalp}@utexas.edu, vj@ece.utexas.edu

Abstract

Enterprise Web applications are moving towards server-side scripting using managed languages. Within this shifting context, event-driven programming *is emerging as a crucial programming model to achieve scalability. In this paper, we study the microarchitectural implications of server-side scripting, JavaScript in particular, from a unique event-driven programming model perspective. Using the* Node.js *framework, we come to several critical microarchitectural conclusions. First, unlike traditional server-workloads such as CloudSuite and BigDataBench that are based on the conventional thread-based execution model, event-driven applications are heavily single-threaded, and as such they require significant single-thread performance. Second, the single-thread performance is severely limited by the front-end inefficiencies of today's server processor microarchitecture, ultimately leading to overall execution inefficiencies. The front-end inefficiencies stem from the unique combination of limited intra-event code reuse and large inter-event reuse distance. Third, through a deep understanding of event-specific characteristics, architects can mitigate the front-end inefficiencies of the managed-language-based event-driven execution via a combination of instruction cache insertion policy and prefetcher.*

Categories and Subject Descriptors

C.1 [**Processor Architecture**]: General

Keywords

Microarchitecture, Event-driven, JavaScript, Prefetcher

1. Introduction

Processor architecture advancements have been largely driven by careful observations made of software. By examining and leveraging the inherent workload characteristics, such as instruction-, thread-, and data-level parallelism, processor architects have been able to deliver more efficient computing by mapping software efficiently to the hardware substrate. We must continue to track the developments in the software ecosystem in order to sustain architecture innovation.

At the cusp of the software evolution are managed scripting languages, which provide portability, enhanced security guarantees, extensive library support, and automatic memory management. In particular, JavaScript is the peak of all the programming languages, surpassing C, C++, and Java to be the most widely used language by developers [1]. From interactive applications in mobile systems to large-scale analytics software in datacenters, JavaScript is ushering in a new era of execution challenges for the underlying processor architecture.

In this paper, we focus on server-side JavaScript, specifically its implications on the design of future server processor architectures. While there are numerous studies that have focused on various aspects of dynamic languages on hardware, such as garbage collection [2], type checking [3], exploiting parallelisms [4, 5], and leveraging hardware heterogeneity [6], we study the implications of the *programming model* that is emerging in server-side JavaScript applications, i.e., asynchronous event-driven programming.

In server-side *asynchronous event-driven* programming [7, 8], user requests are treated as application events and inserted into an event queue. Each event is associated with an event callback. The event-driven system employs a single-threaded event loop that traverses the event queue and executes any available callbacks sequentially. Event callbacks may initiate additional I/O events that are executed asynchronously to the event loop in order to not block other requests. The event-driven model has a critical scalability advantage over the conventional thread-based model because it eliminates the major inefficiencies associated with heavy threading, e.g., context switching and thread-local storage [9, 10]. Thus, the event-driven model has been widely adopted in building scalable Web applications, mainly through the *Node.js* [11] framework.

We find that event-driven server applications are fundamentally bounded by *single-core* performance because of their reliance on the single-threaded event loop. However, unlike conventional single-threaded benchmarks (e.g., SPEC CPU 2006) for which current processor architectures are highly optimized, event-driven server applications suffer from severe microarchitecture inefficiencies, particularly front-end bottlenecks, i.e., high instruction cache and TLB misses and branch misprediction rate. Moreover, unlike conventional heavily threaded enterprise workloads that also suffer from front-end

MICRO 2015 Waikiki, Hawaii USA

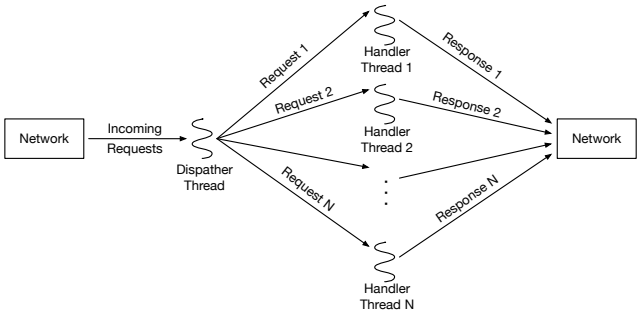


Figure 1: In the thread-based execution model, each incoming client request is assigned to a unique thread, which is responsible for returning a request to the client.

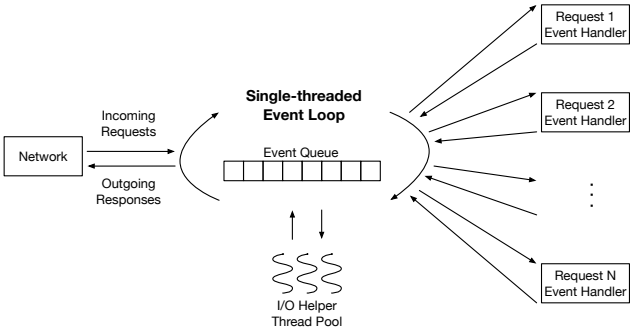


Figure 2: In the event-based execution model, each incoming client request is handled by the single-threaded event loop. I/O operations are handled asynchronously.

issues, the front-end bottleneck of an event-driven server application stems from the single-threaded event execution model, rather than microarchitectural resources being clobbered by multiple threads. With the front-end constituting up to half of the execution cycles, it is clear that current server processor designs are suboptimal for executing event-driven workloads.

To improve the front-end efficiency of event-driven server applications, we study them from an *event perspective*. We find that the severe front-end issue arises fundamentally because events have large instruction footprints with little *intra-event* code reuse. Recent studies on client-side event-driven applications also derive similar conclusions [12, 13]. We take this research a step further to make the key observation that event-driven programming inherently exposes strong *inter-event* code reuse. Taking the L1 I-cache as a starting point, we show that coordinating the cache insertion policy and the instruction prefetcher can exploit the unique inter-event code reuse and reduce the I-cache MPKI by 88%.

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to systematically characterize server-side event-driven execution inefficiencies, particularly the front-end bottlenecks.
- We tie the root-cause of front-end inefficiencies to characteristics inherent to the event-driven programming model, which gives critical microarchitectural optimization insights.
- We show that it is possible to drastically optimize away the instruction cache inefficiencies by coordinating the cache insertion policy and prefetching strategy.

The remainder of the paper is structured as follows. Sec. 2 provides a background into asynchronous event-driven programming and why it has emerged as a crucial tipping point in server-side programming. Sec. 3 presents the workloads we study and shows the event-driven applications' single-threaded nature. Sec. 4 discusses our microarchitecture analysis and presents the extreme front-end bottlenecks. Sec. 5 shows that it is possible to leverage the inherent event-driven execution characteristics to largely mitigate instruction cache inefficiencies through a combined effort between cache insertion policy and a suitable prefetcher. Sec. 6 discusses the related work, and Sec. 7 concludes the paper.

2. Background

Web applications employ server-side scripting to respond to network requests and provide dynamic content to end-users. The traditional approach to providing responsiveness to end-users at scale has been *thread-based programming*, i.e., to increase the number of threads as the number of incoming requests increases. Recently, because of several fundamental limitations of heavy multi-threading that limit system scalability, many industry leaders, such as eBay, PayPal, and LinkedIn, have started to adopt *event-driven programming* as an alternative to achieve scalability more efficiently.

In this section, we first discusses thread-based execution and its limitations (Sec. 2.1). On that basis, we explain why event-driven programming is emerging as an alternative for developing large-scale Web applications (Sec. 2.2).

2.1. Thread-based Programming

Traditional server-side scripting frameworks, such as PHP and Ruby, pair user requests with threads, commonly known as the "thread-per-request" or "single-request-per-script" execution model. These thread-based execution models, shown in generality in Fig. 1, consist of a dispatch thread that assigns each incoming request to a worker thread for processing. The result is that at any given time, the server abounds with the same number of threads as the number of requests.

While the thread-based execution model is intuitive to program with, it suffers from fundamental drawbacks. As the number of client requests increases, so does the number of active threads in the system. As a result, the operating system overhead, such as thread switching and aggregated memory footprint, grows accordingly. Therefore, operating systems typically have a maximum number of threads that they support, which fundamentally limits the server scalability [10].

To address the scalability limitations of thread-based execution on a single node, modern datacenters *scale-out* the hardware resources. Instead of scaling the number of threads on a single compute node, threads are distributed and balanced across a large number of compute nodes. However, increasing the number of physical compute notes has direct financial im-

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

Workload	Domain	Description
Etherpad Lite [14]	Document Collaboration	An online word processing engine, similar to services such as Google Docs and Microsoft Office Online, for real-time document editing and management. The users we simulate create documents, add and edit document contents, and delete documents.
Let's Chat [15]	Messaging	Multi-person messaging platform, akin to Google Chat and WhatsApp, where users participate in group chats. Each enters a group chat and then sends and receives messages.
Lighter [16]	Content Management	A blogging platform comparable to Blogspot. Each of the users requests resources, such as HTML, CSS, JavaScript, and images, corresponding to blog post webpages.
Mud [17]	Gaming	A real-time multi-user dungeon game with multiple users playing the game simultaneously.
Todo [18]	Task Management	A productivity tool and service, similar to the Google Task list management service within Gmail. Users create, modify, and delete multiple tasks within their task lists.
Word Finder [19]	API Services	A word search engine that finds words matching a user-specified pattern. The pattern is searched against a 200,000-word corpus. Users execute several pattern queries.

plications for the service provider. Scaling out requires more power, cooling, and administrative demands which directly affect total cost of ownership [20].

2.2. Event-driven Programming

A more scalable alternative to the conventional thread-per-request execution model is *event-driven execution*, as employed in *Node.js* [11]. Fig. 2 shows the event-driven execution model. Incoming I/O requests are translated to events, each associated with an event handler, also referred to as a callback function. Events are pushed into an event queue, which is processed by the single-threaded event loop. Each event loop iteration checks if any new I/O events are waiting in the queue and executes the corresponding event handlers *sequentially*. An event handler may also initiate additional I/O operations, which are executed asynchronously with respect to the event loop in order to free the main event loop.

Event-driven server designs achieve orders of magnitude performance improvements over their thread-based counterparts in both industry [21, 22] and academia [9, 10], because they are not limited by the number of threads a system supports. Rather, their scalability depends on the performance of the single-threaded event loop. As such, event-driven programming restores the emphasis on *scale-up* single-core processor designs for large-scale Web applications.

3. Event-driven Server Applications

Event-driven server-side scripting has not been extensively investigated in the past. In this section, we identify several important Web application domains that have embraced event-driven programming and describe the workloads we use to represent them (Sec. 3.1). These applications use *Node.js*, which is the most popular server-side event-driven platform based on JavaScript [11]. Numerous companies [23] such as eBay, PayPal, and LinkedIn have adopted it to improve the efficiency and scalability of their application services [21, 22, 24]. We then describe how we generate loads to study realistic usage scenarios of these applications (Sec. 3.2).

Given the selected applications and their loads, we conduct system-level performance analysis on the *Node.js* software architecture to understand its various applications' execution behaviors (Sec. 3.3). The key observation is that while *Node.js* is multi-threaded, the single-threaded event loop that sequentially executes all the event callbacks dominates the CPU time. We use this observation as our rationale to focus on analyzing the event loop execution in the rest of the paper.

3.1. Workloads

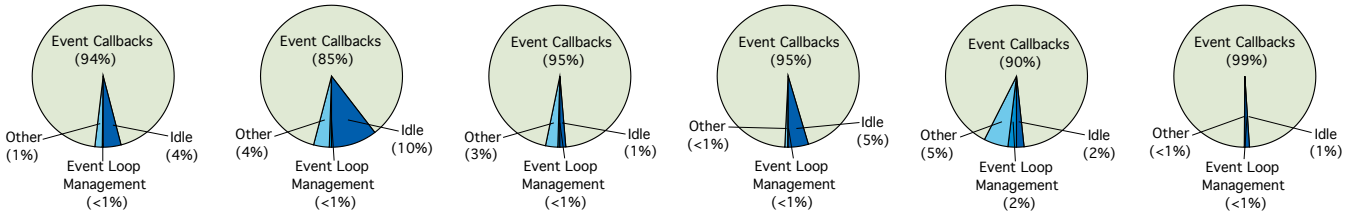
We study important Web application domains that have begun to adopt event-driven programming on the server-side, as shown in Table 1. To enable future research, as well as for reproducibility of our results, we intentionally choose applications that are open-sourced. We release the workload suite at `https://github.com/nodebenchmark`.

Document Collaboration Services such as Google Docs and Microsoft Office Online allow multiple users to collaborate on documents in real-time. As users edit documents, they communicate with the application server to report document updates. We study *Etherpad Lite* [14], a real-time collaborative text editing application. Each user creates a new document, makes several edits to that document, and then finally deletes it once all edits have been made.

Messaging Messengers are amongst the most popular applications used today [25]. Each user sends and receives messages by communicating with the application server that manages a message database. We study the *Let's Chat* [15] messaging application. Each user initiates a new chat, sends multiple messages to other users, and exits the service.

Content Management Many applications and services from news outlets (e.g. CNN) to blogging platforms (e.g. WordPress) rely on content management platforms to serve a variety of file resources to users. We study *Lighter* [16], which is a blogging platform that serves webpage resources corresponding to a blog post. We focus on users who request different blog entry resources using a regular Web browser.

Gaming Online computer gaming, already very popular, is increasingly moving to support multiplayer interaction. These



(a) *Etherpad*. (b) *Let's Chat*. (c) *Lighter*. (d) *Mud*. (e) *Todo*. (f) *Word Finder*.

Figure 3: Execution time distribution of the single-threaded event loop within the *Node.js* workloads we study.

games, such as Farmville and Words With Friends, have to manage the shared game state across multiple users. We study the multiplayer game *Mud* [17], wherein each player is assigned a position on a grid. The players can then update their positions to navigate the environment.

Task Management Cloud-based task management tools such as Asana and Trello have become increasingly popular for organizations and individuals to stay organized. Tasks can be created, modified, and deleted based on what the user desires. We study a simplified form of task management, using the *Todo* [18] task management application. Users can create, view, and modify various tasks in a task list.

API Services Third-party API services provide functionalities that otherwise would not be available in an application. On such example is the Google autocomplete API that aids applications automatically filling out forms [26]. We restrict our study to *Word Finder* [19]. It is a pattern matching API service used for autocompletion and spell checking. Each user queries the API with various word patterns, which are matched against an English dictionary of 200,000 words.

3.2. Load Generation

Our study is focused on processor design at the microarchitecture level, thus we focus on a single instance of *Node.js*. Typically, a production server will run multiple instances of *Node.js* to handle a massive number of requests [27]. However, by design, a single instance of *Node.js* runs primarily in a single thread, coupled with a handful of helper threads.

All of our applications run at the most recent stable software release of *Node.js* at the time of writing (version 0.12.3). To exercise the *Node.js* applications, we develop a query generator to emulate multiple users making requests to the *Node.js* application under study. We model individual users making requests to the server under realistic usage scenarios, which we obtain by observing requests made by real users. We also interleave and serialize concurrent user requests to the server. This does not change the internal execution characteristics of *Node.js* application—because events will eventually be serialized by the single-threaded event loop—but enables crucial reproducibility across experiments. Unless otherwise noted, our results are based on simulating 100 users to minimize experiment runtime, as our detailed microarchitectural analyses are based on simulations. We verified that load-testing with a larger number of users or using different request interleavings did not change the results and conclusions presented throughout the paper.

3.3. Performance Analysis

In order to understand how these applications exercise the *Node.js* runtime and guide the simulations used throughout the remainder of the paper, we conduct a system-level performance analysis using the Intel VTune system profiler tool on a quad-core Intel i5 processor. Because we are running on real hardware, in this section we are able to conduct performance analysis using 100,000 users for each application.

While *Node.js* is actually multi-threaded, we find that the execution time of each *Node.js* application is primarily compute-bound within the single-threaded event loop that is responsible for executing JavaScript-based event callbacks. Our measured results emphasize the need to deeply understand the event-driven execution nature of these workloads.

Event Loop Although the *Node.js* architecture possesses multiple threads for handling asynchronous I/O, its computation is bounded by the single-threaded event loop. The thread-level parallelism (TLP [28]) column in Table 2 shows that *Node.js* applications are effectively single-threaded, indicating the importance of single core performance. This aligns with the experience reported from industry [24, 29].

To further understand the importance of single core performance, we study the compute versus memory boundedness of the single-threaded event loop by measuring how the performance changes with the CPU frequency. Fig. 4 shows each application's normalized execution time as the CPU frequency scales from the peak (3.2 GHz) down to 50%. We observe that the overall performance scales almost linearly with the CPU frequency. For example, for *Mud*, halving the CPU frequency translates to almost 2X slowdown, emphasizing the importance of single-core performance.

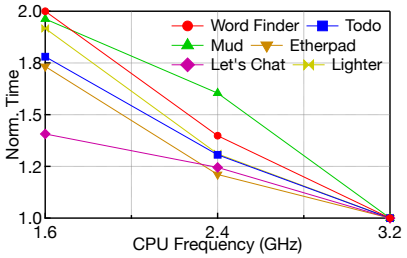


Figure 4: Application performance as CPU frequency scales.

Table 2: Execution characteristics of *Node.js* applications.

Application	System-level			Callback-level		
	User	System	TLP	Generated	VM	Code Gen
Etherpad Lite	98%	2%	1.002	79.8 %	19.21%	0.5%
Let's Chat	95%	5%	1.010	56.0%	38.21%	5.3%
Lighter	96%	4%	1.011	55.6 %	40.59%	4.2%
Mud	99%	1%	1.000	53.7 %	44.09%	1.8%
Todo	85%	15%	1.002	63.1 %	33.95%	3.1%
Word Finder	99%	<1%	1.000	63.8 %	30.66%	5.2%

Due to the dominant nature of the event loop thread, we provide an execution breakdown of the event loop for all *Node.js* applications in Fig. 3. We divide the event loop execution time into four major categories: *event callback execution*, *event loop management* (through *libuv* [30]), *idle*, and *other*. We see that event callback execution dominates the event loop execution time. It consumes between 85% (*Let's Chat*) and nearly 100% (*Word Finder*) of the event loop execution time. In contrast, the event loop management overhead is minimal. In all applications but *Mud*, the event loop management is responsible for less than 1% of the execution time. Idleness due to I/O is also relatively small across all of the applications. Except for *Let's Chat* whose idleness is 10%, the other applications exhibit idleness of less than 5%.

Event Callbacks Because of the dominance of event callback execution in the event loop thread, we provide further details of callback execution behaviors. Event callbacks are written in JavaScript. To execute event callbacks, *Node.js* relies on Google's *V8* JavaScript engine [31]. *V8* supports JavaScript callback execution through various functionalities such as just-in-time (JIT) compilation, garbage collection, and built-in libraries. Table 2 dissects the callback function execution time into three major categories. *Generated* indicates the execution of dynamically compiled code of callback functions. *VM* corresponds to the virtual machine management such as garbage collection and code cache handling. *Code-Gen* corresponds to the JIT compilation.

We make two important observations. First, the majority of the callback time is spent executing the application code (*Generated*). Second, *V8* spends little time generating code (*Code-Gen*), with the largest time spent in *Let's Chat* at 5.3%. This is important to verify because it confirms that our analysis is conducted on each application's steady state, which is the normal state for server applications.

4. Microarchitectural Analysis

Given the single-threaded nature of the event loop, we conduct microarchitectural analysis to identify the bottlenecks for efficient processing of event-driven server applications. We conduct microarchitectural bottleneck analysis using cycle-per-instruction (CPI) statistics to show that instruction delivery dominates execution overhead (Sec. 4.1). Our finding motivates us to perform analysis on the three major microarchitectural structures that impact instruction delivery efficiency: the L1 I-cache (Sec. 4.2), branch predictor (Sec. 4.3), and L1 I-TLB (Sec. 4.4) to understand execution behavior.

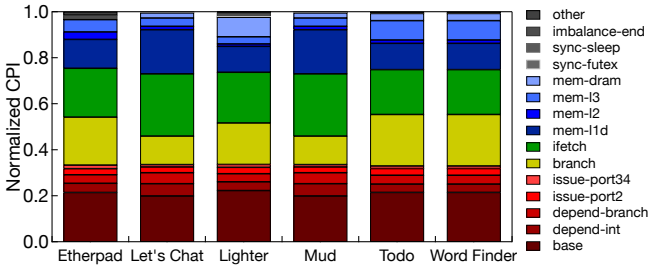


Figure 5: CPI stacks for the different *Node.js* applications.

Throughout our analysis, we compare the *Node.js* applications against SPEC CPU 2006 because the latter has long been the *de facto* benchmark for studying single-threaded performance. A head-to-head comparison between *Node.js* and SPEC CPU 2006 workloads reveals critical insights into the unique microarchitecture bottlenecks of *Node.js*. Other server-side applications, such as CloudSuite [32], MapReduce [33], BigDataBench [34], and OLTP [35] do not specifically emphasize single-thread performance.

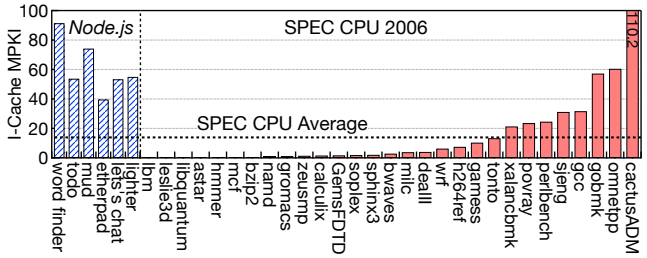
We focus on CPU microarchitecture due to the importance of single-core performance. Hence, our experimental setup is geared toward studying core activities and does not capture the I/O effects (i.e., storage and network). *Node.js* applications may also be I/O intensive. However, a complete I/O characterization is beyond the scope of our paper.

4.1. Microarchitectural Bottleneck Analysis

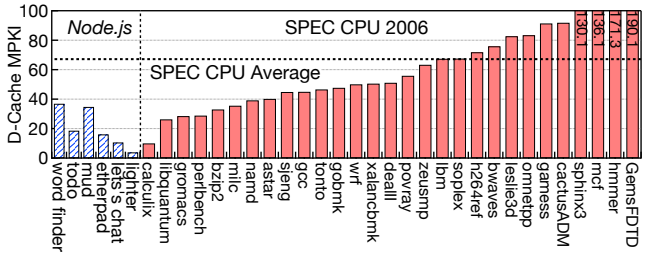
We analyze the microarchitecture bottlenecks for event-driven *Node.js* applications by examining their cycle-per-instruction (CPI) stacks. A CPI stack breaks down the execution time of an application into different microarchitectural activities (e.g., accessing cache), showing the relative contribution of each activity. Optimizing the largest component(s) in the CPI stack leads to the largest performance improvement. Therefore, CPI stacks are used to identify sources of microarchitecture inefficiencies [36, 37].

We use SniperSim [38] to simulate all the *Node.js* applications and generate their corresponding CPI stacks. The CPI stack for the main event loop thread within each application is shown in Fig. 5. Components on each bar represents the percentage of cycles that an application spends on a particular type of microarchitectural activity. For example, the *base* component represents an application's execution time if there were no pipeline stalls. The *ifetch* and *branch* components indicate the total processing overhead due to instruction cache misses and branch mispredictions, respectively. The *mem-* components indicate the time spent accessing different memory hierarchy levels. The *sync-* and *imbalance-end* components correspond to multithreaded execution overheads.

We make two observations from Fig. 5. First, about 80% of the processing time is spent on various types of on-chip microarchitectural activities. Amongst all sources of overall processing overhead, fetching instructions (*ifetch*) and branch



(a) *Node.js* applications show worse I-cache locality than SPEC CPU 2006.



(b) Current designs already capture the data locality of *Node.js*.

Figure 6: I- and D-cache MPKIs comparison for *Node.js* applications and SPEC CPU 2006.

prediction (*branch*) emerge as the two most significant sources. These two components alone contribute about 50% of the total execution overhead, which implies that delivering instructions to the backend of the pipeline is of critical importance to improve the performance of event-driven *Node.js* applications.

Second, the processing overhead for synchronization (*sync-sleep* and *sync-futex*) is negligible. We corroborate this result by performing VTune analysis on 50,000 users. We find that the time spent on synchronization is only about 0.5%. This is not an unexpected result because *Node.js* uses a single thread to execute event callbacks, and all I/O operations are asynchronous without having to explicitly synchronize. This implies that improving the performance of synchronization primitives in the processor will likely not yield much benefit for event-driven server applications.

Thus, we devote the rest of the section to performing detailed analysis on the three microarchitectural resources that significantly impact the front-end's execution efficiency: the L1 instruction cache and L1 instruction TLB (for instruction fetching) and the branch predictor (for branch prediction).

4.2. Instruction Cache Analysis

To understand the front-end execution inefficiency of *Node.js* applications, we start by examining the workloads' instruction cache (I-cache) behavior. We sweep a wide range of cache configurations in order to study the workloads' instruction footprints. We show that all the *Node.js* applications suffer from significantly higher misses per kilo-instruction (MPKI) than the vast majority of SPEC CPU 2006 applications on a standard cache configuration. To achieve SPEC CPU-like MPKI, the processor core would require an I-cache so large that it cannot be practically implemented in hardware.

Current Design Performance Implications We study a modern CPU cache with 32 KB capacity, 64-byte line size, and 8-way set associativity. At this default configuration, we examine the I-cache's efficiency using the MPKI metric. We compare the *Node.js* programs against the SPEC CPU 2006 workloads' MPKIs, and present the results in Fig. 6a.

We make two important observations. First, the average I-cache MPKI of *Node.js* applications is 4.2 times higher than that of the SPEC applications. Even *Etherpad*, which has the lowest I-cache MPKI of all the *Node.js* applications, shows over twice the MPKI of the SPEC average (indicated by the horizontal dotted line in the figure). At the other extreme, *Word Finder* and *Mud* have I-cache MPKIs higher than all but one of the SPEC applications.

Second, the typical behavior of event-driven applications is on par with the worst-case behavior of single-threaded SPEC CPU 2006 applications that are known to stress the microarchitecture. The event-driven *Node.js* applications have MPKIs comparable to some of the worst MPKI of SPEC applications, such as *gobmk*, *omnetpp*, and *cactusADM*.

To understand the reason for the poor I-cache performance, we study the instruction reuse distance to quantify the applications' working set size. Reuse distance is defined as the number of distinct instructions between two dynamic instances of the same instruction [39]. Fig. 7 shows the instruction reuse distances for all of the *Node.js* application. Each (x, y) point corresponds to the percentage of instructions (y) that are at or below a particular reuse distance (x). For comparative purposes, we also include two extreme applications from the SPEC CPU 2006 suite: *lbm* has the lowest I-cache MPKIs and *omnetpp* suffers from the one of the highest I-cache MPKIs.

The event-driven *Node.js* applications have very large reuse distances. The instruction footprint of *omnetpp*, the worst SPEC CPU 2006 application, can be effectively captured within a reuse distance of 2^{11}. In our measurement, the average instruction size is about 4 bytes; this means an I-cache of just 8 KB would be sufficient to capture *omnetpp*'s instruction locality (assuming a fully-associative cache). In contrast, *Let's Chat* has a significantly larger reuse distance of up to 2^{18} instructions, requiring a cache of 1 MB to capture.

For comparison purposes, we also examine the data cache behavior of *Node.js* applications, and compare and contrast it against the SPEC CPU 2006 applications. Fig. 6b shows the D-cache MPKI of *Node.js* and SPEC CPU 2006 applications. Event-driven *Node.js* applications do not appear to stress the data cache heavily. All the *Node.js* applications have MPKIs that are significantly lower than the SPEC CPU average. Even the extreme cases, *Word Finder* and *Mud*, which have the highest MPKIs of 37 and 34, are comparable to the lowest MPKI of SPEC CPU applications.

Ideal Resource Requirements To determine the ideal instruction cache resource requirements for our event-driven *Node.js* applications, we sweep the I-cache size and determine application sensitivity under a variety of resource configurations. We find that the instruction working set sizes approach 1 MB, which far exceeds the typical L1 cache capacity.

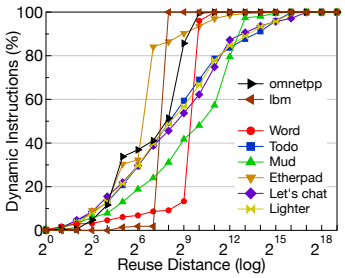


Figure 7: Instruction reuse distances for *Node.js* and SPEC CPU applications.

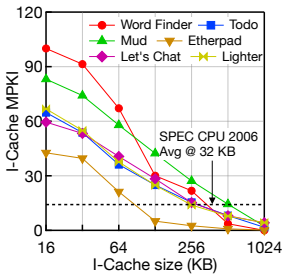


Figure 8: I-Cache MPKI sensitivity of *Node.js* applications to cache sizes.

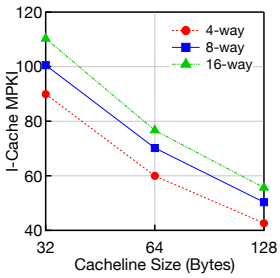


Figure 9: *Mud*'s MPKI with respect to cache line size and associativity.

In Fig. 8, the cache size is swept from 16 KB to 1024 KB on the *x*-axis (in log-scale) and the resulting MPKIs of the *Node.js* applications are shown on the *y*-axis. The SPEC CPU 2006 average I-cache MPKI for a cache of 32 KB is indicated by the horizontal dotted line.

The most significant observation from Fig. 8 is that the I-cache MPKI keeps improving as the cache size increases for all of the *Node.js* applications. Some applications such as *Word Finder* and *Etherpad* show a knee in their MPKIs at the 128 KB cache size. However, it is not until 256 KB, or even 512 KB, that all the event-driven applications have MPKIs that are comparable to the SPEC CPU 2006 average. Such a large L1 I-cache is infeasible for practical implementation.

Instruction cache performance on the event-driven applications cannot be easily improved by adjusting conventional cache parameters, such as line size and associativity. Using *Mud*, which has an MPKI close to the average of all *Node.js* applications, as an example, Fig. 9 shows the impact of line size and associativity while keeping the same cache capacity. The line size is held constant while sweeping the associativity from 4 to 16 ways, and then holding the associativity at 8 ways while sweeping the line size from 32 to 128 bytes. The I-cache MPKI is improved when the two parameters are properly selected. For example, increasing the line size from 64 bytes to 128 bytes improves the MPKI by nearly 25%, indicating that *Node.js* applications exhibit a noticeable level of spatial locality in their code execution behavior. However, the 43 MPKI on a 128-byte line is still significantly higher to the average 14 MPKI of SPEC CPU 2006 applications.

Comparing the impact of the two parameters, cache line size and associativity, changing the associativity has less impact than changing the cache line size. Increasing the cache associativity actually worsens the MPKI on average by about 10 for *Mud*. Between increasing associativity and line size while keeping the cache size the same, increasing the line size to capture spatial locality is a better design trade-off than increasing the associativity to reduce cache conflict misses. But even this cannot reduce the cache misses to the average level of SPEC CPU 2006 workloads. The difference is still as much as two orders of magnitude or more.

4.3. Branch Prediction Analysis

Event-driven *Node.js* applications suffer from bad branch prediction performance. Such behavior stems from the large number of branch instructions in the *Node.js* applications. In SPEC CPU 2006, only 12% of all dynamic instructions are branches. In *Node.js* applications, 20% of all instructions are branches. As such, different branch instructions tend to alias into the same branch prediction counters and thus are likely to pollute each other's predictions. We further show that reducing branch aliasing by attempting to simply scale the branch predictor structures would require excessive resources that are infeasible to implement.

Current Design Performance Implications We compare *Node.js* applications with SPEC CPU 2006 applications under three common branch predictor designs—global, local, and tournament predictor. Intel and AMD do not provide the necessary details to mimic the actual implementation. However, they do provide sufficient information about program optimization [40] that indirectly indicate reasonable predictor parameters. Based on those informational resources, we mimic branch predictor parameters that are typically adopted in today's processors. For all three predictors, we use history registers of 12 bits, which leads to 4 K unique bimodal predictors. The local predictor is configured to use 256 local branch histories. The tournament predictor further utilizes another 4 K bimodal prediction array of its own.

Branch misprediction results are shown in Fig. 10. We draw two conclusions. First, even under the best-performing predictor (the tournament predictor), *Node.js* applications have an average misprediction rate (8.8%) over 2 times higher than that of SPEC CPU (3.7%). Four *Node.js* applications (*Todo*, *Mud*, *Let's Chat*, and *Lighter*) are as hard to predict as the hardest of the SPEC CPU programs (e.g., *gobmk* and *astar*).

Second, the performance difference between the local and global predictors depends heavily on the applications. There-

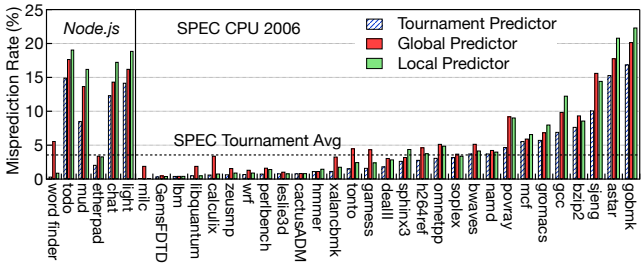
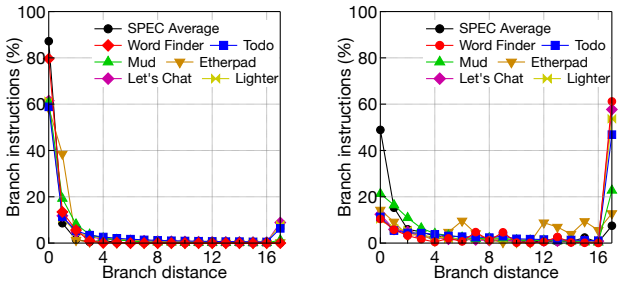
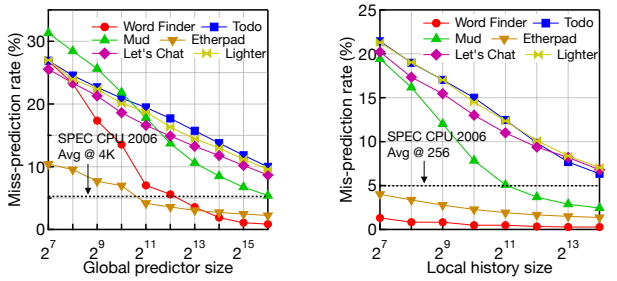


Figure 10: Comparison of *Node.js* and SPEC CPU 2006 applications under three classic branch predictor designs.



(a) Global predictor. (b) Local predictor.

Figure 11: The branch "reuse" distance for the global and local branch predictors across *Node.js* and SPEC CPU 2006.



(a) Global predictor. (b) Local predictor.

Figure 12: The sensitivity of branch misprediction rate with respect to global predictor size and local history table size.

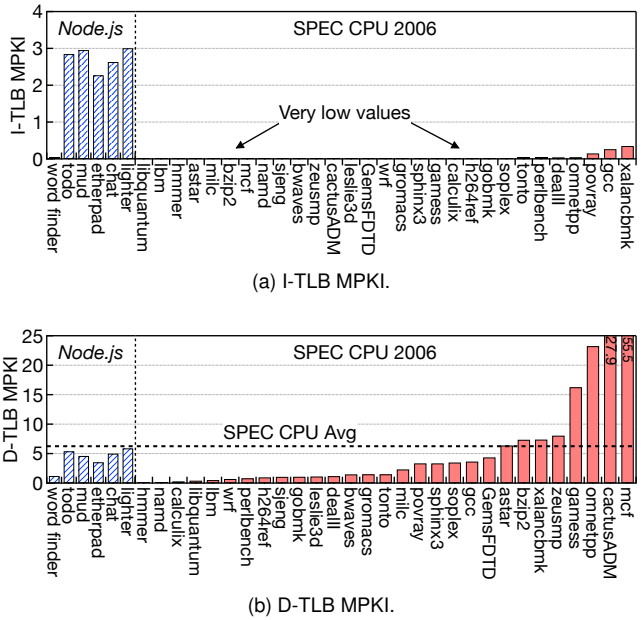
fore, a tournament predictor is necessary to achieve better prediction results. The local predictor is equivalent to or more accurate than the global predictor for *Word Finder* and *Etherpad* but performs worse in the other four applications.

To understand the high branch misprediction rate for the event-driven applications, we study the possibility for destructive branch aliasing to occur. Destructive branch aliasing arises when two or more branch instructions rely on the same prediction counter. We quantify branch aliasing by capturing the number of unique branches between two dynamic instances of the same branch instruction being predicted. We call this number the "branch aliasing distance," which, conceptually, is similar to instruction reuse distance that indicates the number of unique instructions that occur between two dynamic instances of a specific static instruction.

We bin the branch aliasing distance of all the bimodal predictors into 18 bins, each represents a single distance from 0 to 16 and 17+. Zero-aliasing distance is ideal because it indicates that the branch predictor predicts for the same branch instruction back-to-back without any intervening interference from the other branches. A non-zero value for the reuse distance indicates the degree of branch aliasing.

Fig. 11 shows the branch aliasing distances for *Node.js* applications. It also includes the average for SPEC CPU 2006 applications. Each (x, y) point in the figure corresponds to the percentage of dynamic branch instruction instances (y) that are at a particular branch aliasing distance (x).

Node.js applications suffer from heavier branch aliasing



(a) I-TLB MPKI.

(b) D-TLB MPKI.

Figure 13: I-TLB and D-TLB MPKIs for the *Node.js* applications and SPEC CPU 2006 applications.

than SPEC CPU 2006. For the global predictor, Fig. 11a shows that about 90% of the branch predictions in the SPEC applications have zero-aliasing distance. By comparison, in the *Node.js* applications that figure drops to about 60%. Furthermore, about 10% of the predictions in *Node.js* applications have an aliasing distance 17+. SPEC has none that far.

The contrast between *Node.js* and SPEC applications is more prominent in the local predictor (Fig. 11b). Over 50% of the *Node.js* predictions have aliasing distance 17+ while only about 10% do in the SPEC applications. The local aliasing is higher than the global aliasing because local histories are more varied than the global history. Note that we omit the tournament predictor's aliasing as it is indexed identically to the global predictor and so would produce the same results.

Ideal Resource Requirements To determine whether scaling the hardware resources will address the branch prediction and aliasing issues, we sweep the global and local predictor sizes. Even with much larger predictors, the full set of *Node.js* applications never becomes universally well-predicted.

Fig. 12a shows the misprediction rates of the *Node.js* applications as the number of prediction table entries in the global predictor changes from 128 (2^7) to 64 K (2^{16}). Even with 64 K entries, *Word Finder*, *Todo*, *Let's Chat*, and *Lighter* still exceed the average SPEC CPU 2006 misprediction rate at the much smaller 4 K entry predictors. In addition, for most of the applications, as predictor size increases, we observe a remarkably linear trend without a knee of the curve. This indicates that the branch misprediction is far entering the diminishing return area, and further reducing the misprediction requires significantly more hardware resources.

Local predictor trends are similar to the global predictor trends. Fig. 12b shows the misprediction rates as the number

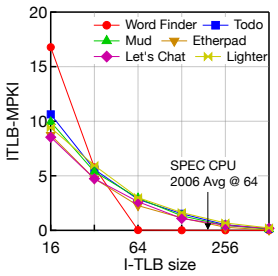


Figure 14: I-TLB sensitivity to varying TLB sizes for *Node.js* applications.

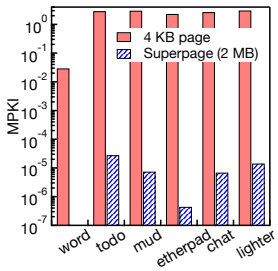


Figure 15: I-TLB MPKI significantly reduces to almost zero with superpages.

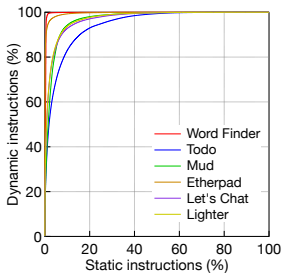


Figure 16: Hotness of instructions as a CDF for *Node.js* applications.

of local histories increases from 128 (2^7) to 16 K (2^{14}). *Mud* is a notable example in that it approaches the prediction accuracy of *Word Finder* and *Etherpad*, which are the easier to predict (see Fig. 10). The remaining three applications, however, require heavy branch prediction hardware investment to even approach the average of SPEC CPU 2006 applications.

4.4. Instruction TLB Analysis

Traditionally, I-TLBs have not been the primary focus in TLB-related studies due to their extremely low miss rates [41–43]. However, I-TLB performance is crucial because every instruction fetch requires a TLB lookup, and TLB misses result in expensive page table walks. Our analyses show that the event-driven *Node.js* applications suffer from high I-TLB misses. Scaling the number of TLB entries reduces the miss ratio, but only at prohibitive hardware cost.

Current Design Performance Implications We simulate a TLB using a Pin tool that resembles the TLB found in modern Intel server processors with 64 entries and 4-way set associativity. Because TLB results are typically sensitive to system-level activities and Pin only captures user-level code, we validated our tool's accuracy using hardware performance counters. Its accuracy is within 95% of the measured hardware TLB results on an Intel processor.

The I-TLB MPKIs of *Node.js* applications dwarf those of the SPEC CPU 2006 suite. Fig. 13a compares the I-TLB MPKI of *Node.js* applications with the SPEC applications. SPEC applications hardly experience any I-TLB misses whereas almost all *Node.js* applications have close to 3 MPKI. In stark contrast, *Node.js* applications fall far short of the worst applications in SPEC in terms of D-TLB MPKIs. As Fig. 13b shows, *Node.js* are roughly comparable to the average D-TLB miss rate of SPEC applications.

To understand whether the poor I-TLB performance is caused by a large code footprint, we analyze the contribution of static code footprint to dynamic instruction execution behavior. Specifically, we study if the event-driven *Node.js* applications contain a few hot instructions or a lot of cold instructions that contribute to a majority of the dynamic instructions that impact the TLB's performance.

We discover that *Node.js* applications have a small number of hot instructions that contribute to a large percentage of the total dynamic instruction count. Fig. 16 shows the hotness of static instructions as a cumulative distribution function. On average, 5% of the static instructions are responsible for 90% of the dynamic instructions. This behavior is similar to many SPEC CPU 2006 applications whose code footprints are attributed to a few hot static instructions [44] and yet do not suffer from poor I-TLB performance.

The data in Fig. 16 suggests that the poor I-TLB performance of *Node.js* applications is not due to a lack of hot code pages; rather it must be due to the poor locality of execution. Sec. 3.3 showed that the *Node.js* applications rely heavily on native call bindings that are supported by the *V8* VM, thus we hypothesize that the user-level context switches between the *Node.js* event callbacks and native code (inside the VM) are the main reason for the poor I-TLB performance.

Ideal Resource Requirements The event-driven *Node.js* applications require unconventionally large I-TLB sizes to achieve SPEC-like I-TLB performance. Fig. 14 shows the I-TLB behavior as the TLB size is progressively increased from 8 to 512 entries. In order to get SPEC-like behavior (indicated by the arrow and so close to the 0 line as to be nearly indistinguishable from it), the I-TLB would have to be increased to 256 or more entries.

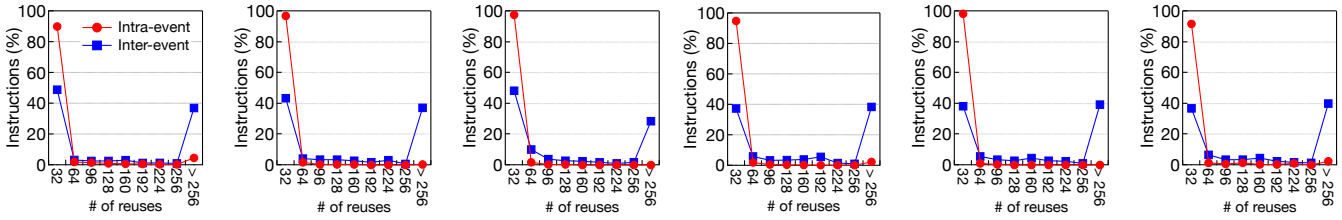
Building such a large I-TLB is inefficient. Current TLB lookups already impose non-negligible energy costs, and therefore scaling the TLB sizes will likely increase energy per access [45, 46]. In fact, TLB sizes have largely remained stable over the past several generations [47].

The alternative to increasing the TLB size is to use superpages. In event-driven applications, switching to a large page size reduces the MPKI significantly. Fig. 15 compares the I-TLB MPKI under 4 KB and 2 MB (i.e., superpage) page sizes. Although superpages are traditionally used for reducing D-TLB misses [43, 48], our results indicate that large pages would be helpful for improving I-TLB performance.

5. Event-based Optimizations

To improve the execution efficiency of event-driven applications, we must mitigate several front-end inefficiencies. However, given all of the major bottlenecks in the front-end, this section specifically focuses on alleviating the instruction cache inefficiencies. The insights are likely to be generalizable to the other structures (i.e., TLB and branch predictor).

We study I-cache misses from an event callback perspective. We find that individual events have large instruction footprints



(a) *Etherpad*. (b) *Let's Chat*. (c) *Lighter*. (d) *Word Finder*. (e) *Todo*. (f) *Mud*.

Figure 17: Frequency of instruction reuse across intra-event and inter-event execution behavior for *Node.js* applications.

with little reuse, which leads to cache thrashing. Fortunately, event-driven programming inherently exposes strong inter-event instruction reuses (Sec. 5.1). Such heavy *inter*-event instruction reuse exposes a unique opportunity for improving the instruction cache performance. We demonstrate that it is necessary to coordinate cache insertion policy and instruction prefetcher (Sec. 5.2). The combined efforts reduce the instruction cache MPKI by 88% (Sec. 5.3).

5.1. Optimization Opportunity

We examine event execution along two important dimensions to discover opportunities for mitigating I-cache inefficiencies: intra-event and inter-event. In the intra-event case, execution characteristics correspond to one event, whereas in inter-event execution the characteristics correspond to the shared execution activity across two or more events.

We analyze intra-event and inter-event instruction reuse to understand the poor I-cache behavior of event-driven applications. Fig. 17 shows the percentage of instructions (y-axis) that are reused a certain amount of times (x-axis) both within and across events for all six *Node.js* applications. The reuses are reported as buckets on the x-axis. The n^{th} bucket represents reuses between X_{n-1} and X_n with the exception of the first bucket, which represents less than 32 reuses and the last bucket which represents 256 or more reuses.

When we consider the event callbacks in isolation (i.e., intra-event) almost 100% of the instructions across all the *Node.js* are reused less than 32 times. The low intra-event reuse is inherent to event-driven programming. Developers consciously program the event callbacks to avoid hot, compute-intensive loops to ensure application responsiveness. Recall that events in the event queue are executed sequentially by the single-threaded event loop, thus all of the events must execute quickly, similar to interrupts (Sec. 3.3).

When the low intra-event code reuse is coupled with the large instruction footprint of each event, it leads to the large instruction reuse distance that results in poor I-cache performance (as previously shown in Fig. 7). In Fig. 18, we show the code footprint (i.e., total byte size of static instructions) for all the events in each application. Each (x, y) point in the figure indicates the percentage of events (x) whose footprints are at or below a particular size (y). We overlay a 32 KB line marker to indicate the L1 I-cache capacity. Almost all of the events have a footprint greater than the standard 32 KB I-cache capacity. In some events, the footprints exceed 1 MB.

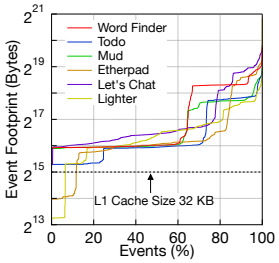


Figure 18: Event footprints.

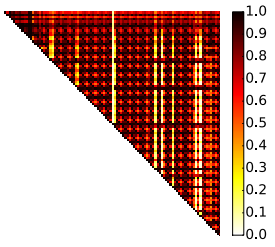


Figure 19: Events similarity.

In contrast, instruction reuse is much higher for inter-event application activity. Fig. 17 shows that over 40% of the instructions are reused over 256 times in the inter-event case. Such frequent reuse implies that events share a considerable number of instructions, otherwise the inter-event behavior would be similar to intra-event behavior.

Inter-event reuse is the direct result of the event-driven programming paradigm: events exercise the same JavaScript VM system functionalities (i.e., the *VM* category in Table 2). In particular, for *Node.js* applications, different event callbacks need the same *V8* JavaScript runtime features, which provides support for compiler optimizations, inline cache handling, garbage collection, various built-in functions, etc.

To quantitatively demonstrate that different events indeed share similar code paths within *V8*'s VM, we show instruction-level similarity between different events. Fig. 19 is a heat map where each (i, j) point in the figure corresponds to an event pair (i, j), where event i appears earlier than event j in the application. Each (i, j) point indicates the percentage of *V8* instructions that event j uses that can also be found in event i. The darkness of the heatmap at any given point is proportional to the percentage of code sharing between those two events, as indicated by the color scale on the right side of the figure. For the purposes of presentation, we limit the data in the figure to 100 randomly chosen consecutive events. We verified that the results hold true when we expand the graph to include all events in the application. The figure confirms that most of the events share close to 100% of the *V8* code.

5.2. Optimization Strategy

The low intra-event reuse coupled with large event footprints suggests that even an optimal cache cannot fully capture the entire working set of all the events. However, the heavy inter-event reuse indicates the potential available locality. Intuitively, the instruction cache needs to first retain the "hot"

fraction of the event working set in the cache so that at least that portion reduces cache misses. In addition, it is necessary to deploy an instruction prefetcher that can always prefetch instructions that are not fully retained in the cache by capturing the instruction-miss sequence pattern.

Caching We propose to use the LRU Insertion Policy (LIP) [49] for the instruction cache (while still maintaining the LRU eviction policy) to retain the hot portion of the event footprint. LIP is known for being able to effectively preserve a subset of a large working set in the cache by inserting all the incoming cache lines into the LRU way instead of the MRU way and only promoting the LRU way to the MRU way if it has a cache hit. As such, the less frequently-used instructions that cause the instruction footprint to exceed the cache capacity will be quickly evicted from the LRU way instead of thrashing the cache. A critical advantage of LIP is that it requires little hardware and design effort and can be readily adopted in existing designs. LIP was originally proposed for last-level caches and used primarily for addressing large data working sets. To the best of our knowledge, we are the first to apply LIP to the instruction stream and show its benefits.

Prefetching Although LIP preserves a subset of event footprints in the I-cache, improvement is still fundamentally limited by cache capacity. As discussed in Sec. 4.2, simply increasing the cache size will lead to practical design issues. To compensate for cache capacity limitations, we must orchestrate the prefetcher to accurately fetch instructions in the miss sequence. Our key observation is that the instruction miss sequence in event-driven applications exhibits strong recurring patterns, primarily because inter-event execution has significant code similarities. For instance, as Fig. 19 shows, different events heavily share code from the *V8* JavaScript engine.

To quantify the recurring patterns in *Node.js* applications, we perform oracle analysis to determine the number of repetitive patterns in the instruction cache miss sequence. We use the SEQUITUR [50] tool, which is widely used to detect patterns in a given stream, to analyze the miss instruction stream of an LIP cache. We classify instruction cache misses into three categories as originally defined in [51]. *Non-repetitive* misses do not belong to any recurring pattern. *New* misses are those instructions misses that appear in a pattern when it first occurs. The subsequent misses in an recurring pattern are classified as *Opportunity* misses.

The oracle repetitive pattern analysis results are shown in Fig. 20. For all the *Node.js* applications, over 50% of the cache misses are opportunity misses. This means up to 50% of the instruction misses can be eliminated if the prefetcher can capture all the recurring patterns and accurately match instruction misses to their corresponding patterns.

We propose to use the Temporal Instruction Fetch Streaming (TIFS) prefetcher [51] to prefetch recurring missing instructions. TIFS predicts and prefetches future instruction misses through recording and replaying the recurring instruction miss pattern. Specifically, it records all the missing instructions into an instruction missing log (IML). Upon an instruction

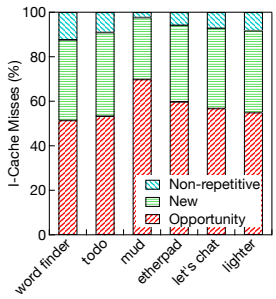


Figure 20: Repetition study.

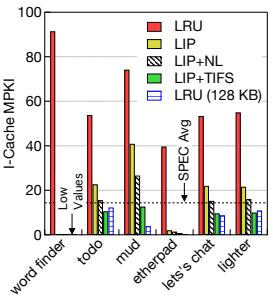


Figure 21: MPKI Results.

miss, TIFS finds the location in IML where the miss address was most recently seen and begins prefetching subsequent instructions from the addresses in the IML.

5.3. Evaluations

We evaluate our proposal using an in-house instruction cache Pin tool. The reason we choose to only simulate the instruction cache is that it isolates other microarchitecture effects and provides crisp insights into the I-cache issue.

We implemented LIP as it was described by Qureshi *et al.* [49]. We do not find noticeable differences between LIP and the bimodal insertion policy (BIP) as observed for the LLC in [49]. Because of the additional hardware cost, we choose LIP instead of BIP. TIFS is implemented as described by Ferdman *et al.* [51]. We find that it is sufficient for the IML to keep track of 8 K instruction misses. More IML entries only lead to marginal improvements. The total storage overhead of TIFS is about 100 KB per core.

The baseline we compare against is the standard LRU cache with a 32 KB I-cache, 64-byte line size, and 8-way set associativity. We compare it with the following schemes. First, we compare it against an instruction cache with LIP insertion policy to understand the effectiveness of retaining the hot fraction of the event working set. Second, we compare it against a LIP-enabled cache with a next-line prefetcher to understand the effectiveness the common prefetching scheme. Third, we compare it against a LIP-enabled instruction cache enhanced with the TIFS prefetcher to understand the benefits of prefetching recurring instruction misses. Finally, we compare against a LIP instruction cache of 128 KB size without TIFS to understand whether the storage overhead introduced by TIFS can be simply used to increase the cache size.

The I-cache MPKI comparison results are shown in Fig. 21. We also overlay the average MPKI of SPEC CPU 2006 applications at 32 KB. We see that LIP drastically improves the MPKI by at least 45% and 70% on average compared to the LRU-only cache policy. This suggests that without any prefetching scheme, simple changes to the cache insertion policy can already reduce the I-cache MPKI for *Node.js* application significantly. In two applications, *Word Finder* and *Etherpad*, LIP is able to eliminate almost all of the cache misses. For other applications, however, their MPKIs are still higher than the SPEC applications' average.

The TIFS-based instruction prefetcher reduces the MPKI by

another 60% on top of the cache improvements. As a comparison, using a next-line prefetcher only reduces the MPKI by 33.6%. With TIFS, all applications' MPKI fall below SPEC CPU 2006's average. This shows the necessity of capturing the instruction misses' recurring pattern for prefetching. Combining the LIP cache with TIFS prefetching effectively reduces the I-cache MPKI by 88%, which would otherwise be impossible to achieve without event-specific optimization. LIP+TIFS is almost as effective as an extremely large L1 I-cache. In all but one applications (*Mud*), LIP+TIFS achieves an equivalent or better MPKI than a 128 KB I-cache.

Cost Analysis The cost of LIP is negligible. TIFS operations (e.g., logging miss sequences in IML, updating the Index Table) are off the critical path, following the general design principle of prefetching structures [52, 53]. Hence, TIFS is not likely to affect the cycle time. We also estimate that the additional power consumption of TIFS-related structures is only about 92 mW based on CACTI v5.3 [54].

6. Related Work

Characterization of Emerging Paradigms At the time multicore was starting to become ubiquitous on commodity hardware, Bienia *et al.* developed the PARSEC multicore benchmark suite [55]. Similarly, Ranger *et al.* characterized the implications of MapReduce applications when MapReduce was becoming prevalent in developing large-scale data-center applications. More recently, numerous research efforts have been devoted to characterizing warehouse-scale and big data workloads [32, 34, 35, 56–58].

We address a new and emerging computing paradigm, i.e., event-driven programming, as others have done in other domains at the time those domains were becoming important. Although event-driven programming has existed for many years for highly concurrent server architecture [9, 59, 60], large-scale simulations [61, 62], and interactive graphical user interface (GUI) application design [63], server-side event-driven applications that are tightly coupled with scripting languages have only recently become important. In this context, our work is the first to present a comprehensive analysis of the microarchitectural bottlenecks of scripting-language-based server-side event-driven applications.

Asynchronous/Event Execution Analysis Prior work on event-driven applications primarily focus on client-side applications [12, 13] whereas we study server-side applications. While prior art also attributes front-end bottlenecks to little intra-event code reuse and proposes instruction prefetching and pre-execution techniques, we take the event-level analysis a step further to demonstrate heavy inter-event code reuse. As a result, we show that simple changes to the instruction cache insertion policy can drastically improve the front-end efficiency, even without the prefetching. Hempstead et al. [64] designed a specialized event-driven architecture for embedded wireless sensor network applications. Our work focuses on server-side event-driven programming and studies its implications on the general purpose processor. EBS [65] improves the energy-efficiency of client-side event-driven Web applications and is orthogonal to the performance study of our paper.

Scripting Languages Richards *et al.* explore language-level characteristics of client-side JavaScript programs [66]. Our work studies server-side JavaScript and focuses on the nature of events and their microarchitectural implications. Prior work on improving the performance of JavaScript, especially its dynamic typing system [3, 67], complements our event-level optimizations. Ogasawara conducted source code analysis of server-side JavaScript applications, also using *Node.js*, and found that little time is spent on dynamically compiled code, leading to limited optimization opportunity [68]. We take an event perspective and demonstrate significant optimization opportunities by exploiting event-specific characteristics. In addition, the prior work does not focus on or investigate the microarchitectural implications of event-driven execution.

7. Concluding Remarks

As computer architects, it is important to understand how to optimize (micro)architecture in light of emerging application paradigms. This paper systematically studies microarchitectural implications of *Node.js* applications, which represent the unique intersection between two trends in emerging server applications: managed language systems and event-driven programming. We show that *Node.js* applications are bottlenecked by front-end inefficiencies. By leveraging heavy *inter-event* code reuse inherent to event-driven programming, we drastically improve the front-end efficiency by orchestrating the instruction cache insertion policy with an instruction prefetcher. Our results are readily useful for building an optimized server architecture for event-driven workloads and provide a baseline, which further research can build upon.

8. Acknowledgments

We are thankful to our colleagues as well as the anonymous reviewers for the many comments that have contributed to this work. This work is partially supported by AMD Corporation. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] "The redmonk programming language rankings: January 2015." http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *Proc. of SIGMETRICS*, 2004.

[3] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked load: Architectural support for javascript type-checking on mobile processors," in *Proc. of HPCA*, 2011.

[4] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *Proc.of HPCA*, 2011.

[5] M. Mehrara and S. Mahlke, "Dynamically Accelerating Client-side Web Applications through Decoupled Execution," in *Proc. of CGO*, 2011.

[6] T. Cao, T. Gao, S. M. Blackburn, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proc. of ISCA*, 2012.

[7] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proc. of SIGOPS European Workshop*, 2002.

[8] "Why threads are a bad idea (for most purposes)." http://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf.

[9] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," in *Proc. of SOSP*, 2001.

[10] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler, "A design framework for highly concurrent systems," in *Technical Report No. UCB/CSD-00-1108*, 2000.

[11] Joyent, Inc., "Node.js." https://nodejs.org/.

[12] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven webapplications," in *Proc. of PACT*, 2014.

[13] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating asynchronous programs through event sneak peek," in *Proc. of ISCA*, 2015.

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

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

[16] "Lighter." https://github.com/mehfuzh/lighter.

[17] "Mud." https://github.com/gumho/simple-node.js-mud.

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

[19] "Word Finder." https://github.com/amirrajan/word-finder.

[20] D. A. Wood and M. D. Hill, "Cost-effective parallel computing," in *IEEE Computer*, 1995.

[21] "Exclusive: How linkedin used node.js and html5 to build a better, faster app." https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.

[22] "Node.js at paypal." https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.

[23] "Projects, applications, and companies using node." https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node.

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

[25] "Why Apps for Messaging Are Trending." http://www.nytimes.com/2015/01/26/technology/why-apps-for-messaging-are-trending.html.

[26] "Autocomplete for Addresses and Search Terms." https://developers.google.com/maps/documentation/javascript/places-autocomplete.

[27] Joyent, Inc., "Node.js Cluster." https://nodejs.org/api/cluster.html.

[28] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "Thread-level parallelism and interactive performance of desktop applications," in *Proc. of ASPLOS*, 2000.

[29] "Node.js high availability at box." https://www.box.com/blog/node-js-high-availability-at-box/.

[30] "libuv." https://github.com/libuv/libuv/.

[31] "Chrome V8." https://developers.google.com/v8/.

[32] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. of ASPLOS*, 2012.

[33] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proc. of HPCA*, 2007.

[34] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: a big data benchmark suite from internet services," in *Proc. of HPCA*, 2014.

[35] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos, "Slicc: Self-assembly of instruction cache collectives for oltp workloads," in *Proc. of MICRO*, pp. 188–198, 2012.

[36] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *Proc. of ASPLOS*, 2006.

[37] W. Heirman, T. E. Carlson1, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *Proc. of IISWC*, 2011.

[38] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[39] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," in *IEEE Micro*, 2007.

[40] "Intel 64 and ia-32 architectures optimization reference manual," 2014.

[41] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Prof. of MICRO*, 2012.

[42] G. B. Kandiraju and A. Sivasubramaniam, "Characterizing the d-tlb behavior of spec cpu2000 benchmarks," in *Proc. of SIGMETRICS*, 2002.

[43] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proc. of OSDI*, 2002.

[44] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *Proc. of ISCA*, 2007.

[45] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proc. of ISCA*, 2014.

[46] A. Sodani, "Exascale: Opportunities and challenges," in *MICRO 2011 Keynote address*, 2011.

[47] "Intel 64 and ia-32 architectures optimization reference manual." http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[48] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in supporting two page sizes," in *Proc. of ISCA*, 1992.

[49] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of ISCA*, 2007.

[50] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," in *Journal of Artificial Intelligence Research*, 1997.

[51] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proc. of MICRO*, 2008.

[52] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. of MICRO*, 2009.

[53] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, 2009.

[54] "Cacti 5.3." http://www.hpl.hp.com/research/cacti.

[55] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. of PACT*, 2008.

[56] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, D. Brooks, *et al.*, "Profiling a warehouse-scale computer," in *Proc. of ISCA*, 2015.

[57] K. Lim, D. Meisner, A. G. Said, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proc. of ISCA*, 2013.

[58] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Proc. of ISCA*, 2008.

[59] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable web server," in *Proc. of USENIX ATC*, 1999.

[60] J. C. Hu, I. Pyarali, and D. C. Schmidt, "High performance web servers on windows nt: Design and performance," in *Proc. of USENIX Windows NT Workshop*, 1997.

[61] J. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young, and D. E. Shaw, "Hardware support for fine-grained event-driven computation in anton 2," in *Proc. of ASPLOS*, 2013.

[62] B. Wang, Y. Zhu, and Y. Deng, "Distributed time, conservative parallel logic simulation on gpus," in *Proc. of DAC*, 2010.

[63] R. W. Scheifler and J. Gettys, "The x window system," in *Proc. of ACM Transactions on Graphics*, 1986.

[64] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *Proc. of ISCA*, 2005.

[65] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *Proc. of HPCA*, 2015.

[66] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysisof the dynamic behavior of javascript programs," in *Proc.of PLDI*, 2010.

[67] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving javascript performance by deconstructing the type system," in *Prof. of PLDI*, 2014.

[68] T. Ogasawara, "Workload characterization of server-side javascript," in *Proc. of IISWC*, 2014.