Locality Lost: Unlocking the Performance of Event-driven Servers*

Daniel Richins Yuhao Zhu Matthew Halpern Vijay Janapa Reddi

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

1. Introduction

Server-side Web applications are in the midst of a software evolution. Application developers are turning away from the established thread-per-request model, where each request gets a dedicated thread on the server, and toward event-driven programming platforms, which promise improved scalability and CPU utilization [1]. In response, we perform a microarchitectural analysis of these applications in current server processors and identify several serious performance bottlenecks and optimization opportunities for future processor designs.

The event-driven model is illustrated in Fig. 1. Incoming requests require periods of computation to prepare their responses, interspersed with periods of inactivity while long-latency I/O events are waiting to complete. The event-driven model hides these idle periods by placing all the periods of computation—*events*—into an *event queue*, where they are processed, one at a time, by a single thread. The processing of events is done by *event callback functions*, labeled **A** through **E** in the figure. In the event queue, the callback functions from different requests can be interleaved. We make the following important observations about event-driven programming:

- **Event-driven programming has limited code locality.** Event-driven programming relies on flat functions that are used repeatedly. This programming pattern leads to large instruction reuse distances and severely limits and handicaps available intra-event (i.e., within an event) code reuse but exposes ample inter-event (i.e., across events) code reuse.
- **Limited code locality leads to front-end bottlenecks.** The instruction cache, branch predictor, and instruction TLB are extremely inefficient in delivering instructions, limiting performance, as compared to SPEC CPU 2006 applications that are also single-threaded and widely used to stress the CPU's capabilities and drive microarchitecture innovations.
- **Microarchitectural optimizations can regain the lost code locality.** By utilizing two innovative hardware designs, a cache insertion policy and an instruction prefetcher, tailored specifically to extract different sources of locality, the I-cache performance can be satisfactorily improved.

On servers, the event-driven programming paradigm is perhaps best exemplified by *Node.js* [2], which we use as our experimental platform. *Node.js* has achieved rapid adoption in industry, and companies relying on warehouse-scale computing capability (such as PayPal [3] and LinkedIn [4]) for its scalability and responsiveness [1]. We introduce and use for study the first *Node.js* workload suite, consisting of diverse and representative open-source applications (*Etherpad Lite*, *Let's Chat*, *Lighter*, *Mud*, *Todo*, and *Word Finder*), which are available at https://github.com/nodebenchmark. Our insights can have immediate and practical real-world impact.

*The original article appears under the title "Microarchitectural Implications of Event-driven Server-side Web Applications," published in the 48th International Symposium on Microarchitecture, Dec. 2015.

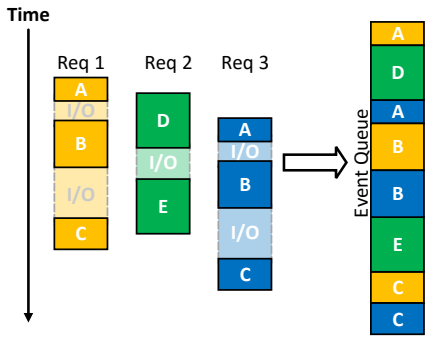


Figure 1: The responses to incoming requests can logically be divided into computational pieces, separated by idle periods, where long-latency I/O events are waiting to complete. The event-driven programming model overcomes these idle periods by placing the processing of all events—their event callback functions—into a single-threaded event queue. Node.js on the server-side and Android on the client-side are prevalent examples of event-driven programming.

2. Lost Locality

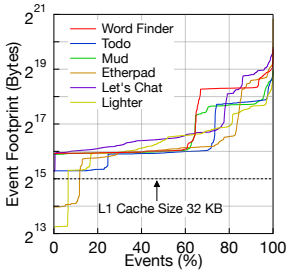
The event-driven *Node.js* applications violate one of the most fundamental assumptions in CPU design: code locality. In this section, we describe in greater detail how the unique properties of *Node.js* applications combine to limit locality (Sec. 2.1). We then show the measurable impact of this low locality on processor performance (Sec. 2.2). And finally, we show that the event-driven program characteristics preclude simple structure scaling as a viable solution (Sec. 2.3).

2.1. Intra-event Locality

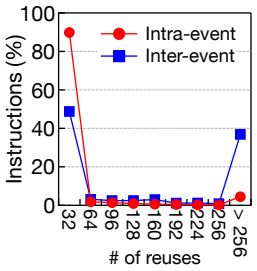
Where conventional single-threaded applications, such as those in SPEC CPU 2006 [5], tend to be centered around limited tight, hot loops that dominate execution time, the applications of *Node.js* exhibit two striking deviations from this tendency: (1) event callback functions have extremely large code footprints, and (2) those same functions tend to be relatively flat. These properties combine to ill effect in *Node.js*.

Callback Footprints Fig. 2a shows the large code footprints of the event callback functions. Of note is the fact that most footprints are larger than the I-cache. This extremely large footprint likely arises, at least in part, from programming in JavaScript with all its virtual machine overhead.

Code Reuse Event callback functions also exhibit relatively little code reuse. Fig. 2b shows, as an example, the instruction reuse counts for *Etherpad Lite*. Within callback

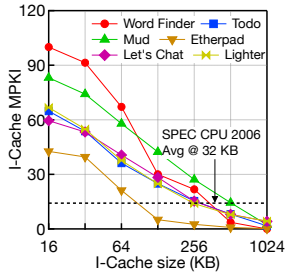


(a) Most event callback functions have instruction footprints larger than the I-cache itself.

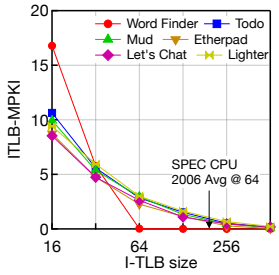


(b) The intra-event instruction reuse is limited. Only in inter-event reuse is it significant.

Figure 2: The code reuse distance combined with code footprint size destroys most locality.



(a) I-cache MPKI remains much higher in *Node.js* than SPEC until the cache is increased to 512 KB.



(b) I-TLB MPKI can be reduced to SPEC levels only with 256 TLB entries.

Figure 3: Simply scaling the sizes of the I-cache and I-TLB would be too expensive a solution.

functions (*intra-event*; shown in red), very few instructions are used more than 32 times. This limited intra-event code reuse is a consequence of the event-driven programming model. To ensure responsiveness in the single-threaded event queue processing, programmers understand they must avoid time-consuming and computationally-intensive loops, instead writing flat code.

Making matters worse, however, is that this intra-event code reuse is diffuse and hard to utilize. Because it is not found in hot loops, it must instead be spread across the entire event callback function, which limits the locality even further.

Working Set Size The ultimate measure of locality is the working set size. We compare the working set sizes in *Node.js* with SPEC CPU 2006 applications, since both are single-threaded applications. *Node.js* applications are orders of magnitude larger than those of SPEC applications. Capturing the entire working set requires an I-cache of size 512 KB, orders of magnitude larger than SPEC (Fig. 7 in the original paper).

2.2. Performance Implications

The *limited locality* and *large working set size* manifest themselves in the CPU pipeline structures' performance. The front-end suffers from particularly low efficiency. We demonstrate the effect on the I-cache, I-TLB, and branch predictor.

The I-cache and I-TLB both show miss rates that are in line with those usually seen in the D-cache and D-TLB. Fig. 3 shows the misses per kilo-instruction (MPKI) of each structure across a range of sizes. The default sizes are 32 KB and 64 entries, respectively. At those sizes, *Node.js* manages 61.2 MPKI and 2.29 MPKI compared to SPEC's 14.4 MPKI and 0.03 MPKI for the cache and TLB, respectively. The differences are orders of magnitude off—4.25x and 76.3x!

On the branch predictor side, the *Node.js* applications are predicted far worse than the SPEC applications. Using a 4 K-prediction tournament predictor, *Node.js* manages an average misprediction rate of 8.7% compared to SPEC's 3.6%.

2.3. Structure Scaling

The obvious reaction to address the severe problem—simply increase the sizes of these three structures—is an untenable solution. The unique characteristics of event-driven programs make simple scaling an ineffective approach (see Sec. 2.1).

To reach the I-cache and I-TLB rates of typical SPEC performance, *Node.js* would require structures of 512 KB and 256 entries, respectively (see Fig. 3). Accurate branch prediction using existing common predictors would require similarly overblown structures. Even given an unlimited hardware budget, the increased latency of accessing each structure would likely outweigh the benefits gained from higher hit rates.

3. Capturing Code Reuse

Though event-driven programs have less easily-utilized locality than other programs, it is still possible to exploit some of that and regain performance; furthermore, Fig. 2b shows that there is ample code reuse across event callback functions (*inter-event*; shown in blue). We show one approach to capitalizing on these two different kinds of reuse on the I-cache. Our insight is to combine two existing techniques intelligently: a cache insertion policy to capture intra-event code reuse and an instruction prefetcher to capture inter-event code reuse.

Cache Insertion Policy We analyze the instruction stream and observe similarities to streaming data. We use the LRU Insertion Policy (LIP) [6] to capture intra-event code reuse, preserving the reused instructions from the otherwise flat event callback functions. LIP was invented to handle streaming data workloads. Instead of newly fetched lines being inserted into the most-recently used (MRU) position of the cache, they are inserted in the least-recently used (LRU) position. Only once a line in the LRU position has a subsequent hit is it promoted to the MRU position. This prevents streaming data (or, in our case, streaming instructions) from polluting the cache.

Prefetching For inter-event code reuse, we rely on the Temporal Instruction Fetch Streaming (TIFS) prefetcher [7]. It is built on the observation that many I-cache miss sequences are repeatable. It operates by recording every access to the I-cache that results in a miss. When subsequent cache misses are found to have been recorded, it begins prefetching lines in the same sequence as the misses previously recorded, which suits our observation of ample code reuse across callback functions.

Evaluation Combining LIP and TIFS reduces I-cache MPKI by 88%. LIP improves the average I-cache miss rate for *Node.js* applications from 61.2 MPKI to 18.2 MPKI, a 70% improvement. The TIFS prefetcher provides an additional 60% reduction, bringing the miss rate down to 7.22 MPKI.

4. Long-term Impact

Software evolution necessitates architecture innovation. New programming paradigms give rise to new architectural bottlenecks that obstruct otherwise efficient program execution. In order to sustain these software innovations, it is important to understand their execution and optimize future architectures.

Our paper identifies an emerging programming paradigm—server-side event-driven programming—and takes the first important steps in analyzing its execution on modern hardware to identify performance bottlenecks. Though other researchers have operated with the understanding that there are instruction fetch and branch prediction bottlenecks in client-side event-driven applications [8,9], this is the first paper to quantitatively measure the bottlenecks on server-side event-driven applications and, more importantly, to identify their root causes in the context of the event-driven programming model.

4.1. Wide-ranging Applicability

While our work focuses on a particular use case—server-side event-driven programs—it is applicable to a wider range of programs. Event-driven programming, though not a new concept, is showing up in new application domains: it is present in nearly all GUI programming, as program actions are triggered by user interaction; it partially drives the Android operating system [10]; and, perhaps most importantly, event-driven programming is inherent to Web browsers themselves [11] and, by extension, to any Web-based mobile application. Hence, with both server and client running event-driven software, much of our interaction with today's technology is event-driven.

The key observation of this work is that instruction locality is lost within event-driven applications. The flip side of this observation is that inter-event code reuse is plentiful. This is inherent to the event-driven programming paradigm, i.e., developers tend to write flat and short-running event callbacks. We believe this to be true of any event-driven program. Consequently, we expect that I-cache hit rates will be suboptimal and branch mispredict rates will be uncomfortably high in many, if not all, event-driven applications.

4.2. Catalyst for Cross-layer Research

The primary message from this paper—combining large callback instruction footprints with the flat event callback functions of event-driven programming leads to extremely low locality—has important implications for processor architecture, compiler, and software researchers and designers.

Architecture Architects must continue to improve microarchitectural efficiency. The unique and yet widely used event-driven programming paradigm is fundamentally different from traditional workloads and so merits dedicated research into code reuse patterns, branch prediction opportunities, and specialized hardware to match these programs' needs.

For example, though the combination of LIP and TIFS was shown to provide an 88% reduction in I-cache MPKI, the MPKIs of some *Node.js* applications are still higher than some CPU 2006 applications. LIP and TIFS do not leverage event-specific knowledge. Our results indicate that specializing front-end structures based on a deeper understanding of event-level characteristics can be a promising new research direction.

Runtime The locality issue is also rife with opportunity for event-driven runtime research. Because of the inter-event code reuse, event-level scheduling shows promise as a software technique to improve efficiency. Reordering callbacks in the event queue to group event callbacks of the same type (which will have similar instruction streams) may reduce the reuse distance in the presence of large per-event footprints.

In addition, the large event callback function footprints of *Node.js* applications may prove, in part, to be the result of JavaScript compilation techniques that are well optimized for Web browsers but suboptimal for event-driven servers. As *Node.js* grows in popularity, JavaScript and other language designers will have to consider the impact their decisions may have on event-driven execution (e.g., instruction footprint). Reducing the footprint may prove to be a greater performance boon than whatever optimizations exist in the larger footprints.

Software Our locality findings also have critical implications for application developers. They will not only have to make sure that event callback functions execute quickly but also that they have small instruction footprints. We will have to educate developers and introduce programming conventions that reduce instruction footprints, such as splitting a flat callback into multiple smaller callbacks and chaining them through asynchronous calls. Smaller event callbacks combined with the event-level reordering and scheduling techniques discussed above will likely reduce instruction reuse distance and therefore improve instruction delivery efficiency.

4.3. Citation from 2026

In their landmark paper on event-driven server-side Web applications, the authors pioneered the work on recovering code locality in event-driven applications. Since then, numerous studies have proposed hardware and software techniques to capture and improve locality. From the original paper, an entire body of research was spawned and event-driven programming has become pervasive due to its flexibility and performance.

References

[1] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler, "A design framework for highly concurrent systems," in *TR UCB/CSD-00-1108*, 2000.

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

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

[4] "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/.

[5] Standard Performance Evaluation Corporation, "SPEC CPU 2006." https://www.spec.org/cpu2006/.

[6] 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.

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

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

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

[10] E. L. Mañas, "Event-driven programming for Android (part I)." https://medium.com/google-developer-experts/event-driven-programming-for-android-part-i-f5ea4a3c4eab, January 2015.

[11] E. Swenson-Healey, "The JavaScript Event Loop: Explained." http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/, October 2013.