Smartphones

Editor: Nayeem Islam ■ Qualcomm ■ nayeem.islam@gmail.com

Concurrency in Mobile Browser Engines

Călin Caşcaval, Pablo Montesinos Ortego, Behnam Robatmili, and Darío Suárez Gracia, Qualcomm Research Silicon Valley

Web browsers are our main window into the wealth of information available on the Internet. All consumer computing platforms, including smartphones and tablets, rely on a browser to provide news, entertainment, and services. We use the term *Web apps* to refer to applications designed and implemented using Web technologies. Some Web apps require users to launch their Web browsers, while others appear to the user as native applications, even though they are just an API layer on top of a browser engine. Using Web technologies as the application back end is a convenient way of building portable applications across a variety of platforms.

However, this presents two main challenges. First, browsers must provide a smooth user experience—fast page load, satisfactory scroll and zoom performance, and uniform behavior regardless of the underlying hardware. The browsers' JavaScript engines thus must provide close-to-native application performance. The second challenge is that when running on mobile devices, browsers must adapt to the related energy and connectivity constraints.

As a result, browsers have been evolving to exploit the underlying hardware. Most current smartphones and tablets have systems on a chip (SoCs), with two to eight cores and powerful GPUs, and they rely on a plethora of techniques to maximize the performance/power ratio. Such techniques include power and clock gating, dynamic voltage and frequency scaling, and offloading work to specialized cores. On the network side, Long-Term Evolution (LTE) offers 100 Mbps bandwidth, yet network latency continues to be high. Web browsers must exploit all available capabilities to address performance and energy challenges.

Here, we focus in particular on how Web browsers can use concurrency to improve per-tab (or per-page) processing. We use the Zoomm browser engine[1] and its MuscalietJS JavaScript engine[2] to illustrate how parallel processing improves performance and hides network latency for faster page loads.

EXPLOITING CONCURRENCY

Desktop browsers, such as WebKit (www.webkit.org) and Firefox (www.mozilla.org/firefox), typically exploit multiple cores by running each tab as a separate collection of processes and relying on the OS scheduler to place processes on different cores. The Zoomm browser architecture was designed with a different goal: take advantage of multicore processing for each browser tab. This is in line with typical mobile device usage, and it lets a more constrained platform meet its performance and energy goals.

A Parallel Browser Architecture

A Web browser has several major components: parsers (HTML, CSS, JavaScript) that create the Document Object Model (DOM), a Cascading Style Sheets (CSS) engine to format and style the DOM, a layout engine to produce the image that will be displayed to the user, a rendering engine to display the page, and a JavaScript engine to enable interactivity and dynamic behavior.

Figure 1 shows the breakdown of execution time by component, excluding the network time. Our measurements, similar to other work,[3] show that the network time is 30–50 percent of the total execution time. As the Web evolves, we're seeing remarkable changes in complexity and dynamic behavior. For example, in 2010, Leo Meyerovich and Rastislav Bodík measured WebKit execution and observed that JavaScript took approximately 5 percent of the execution time.[4] One year later, the fraction of JavaScript execution increased to 30 percent, and for most webpages, it has since plateaued.

Even more significantly, we're observing a major trend to support application development using Web technologies such as HTML5, CSS, and JavaScript. Given this breakdown of computation, it is clear that to optimize the browser execution using concurrent processing, all major components must be addressed, because the gains from optimizing the components in isolation are bounded.

Our goal is to exploit concurrency at multiple levels: parallel algorithms

Published by the IEEE CS ■ 1536-1268/15/$31.00 © 2015 IEEE

for individual passes to speed up the processing of each component, and overlapping of passes to speed up the total execution time. In addition, we must respect the HTML and JavaScript semantics, even during concurrent execution. The main data structure used by all browser passes is the DOM. The DOM is a tree representing all HTML elements, including their content, relationships, styles, and positions. Web programmers use JavaScript to manipulate the DOM, producing interactive webpages and Web apps. Most communication between browser passes and components happens through the DOM. Unfortunately, even in a concurrent browser, access to the DOM tree (constructed by the HTML5 parser) must be serialized to conform to the HTML5 specification (see http://whatwg.org/html).

This is the biggest limitation Zoomm must contend with, and it significantly influenced the design. In our architecture, we manage access to the DOM through a dispatcher. Most passes have their own private concurrent data structures to allow for greater parallelism inside components, and they send asynchronous DOM updates to the dispatcher for processing. Figure 2 shows the architecture's high-level components, discussed in more detail next.

Zoomm Browser Components

The Zoomm browser consists of a number of loosely coupled subsystems, all of which were designed with concurrency in mind. With the exception of the browser global resource manager and the rendering engine, all subsystems are instantiated once for each page (shown as a separate tab in the user interface).

Resource manager. The resource manager is responsible for managing and preprocessing all network resources, including fetching resources from the network, providing cache management for fetched resources, and notifying other browser components when data from the network arrives.

In our first implementation, all resources are fetched in the order in which they appear, without imposing any priorities. In addition, the resource manager includes other components, such as the HTML prescanner and image decoder. The HTML prescanner quickly determines all external resources in an HTML document, requests their downloading, and, depending on the type of resources, requests further processing. The image decoder component consists of a thread pool that decodes images for later use as the resource manager receives them. These operations are fully concurrent, because each image decode is an independent task.

DOM engine. In Zoomm, each page (tab) instantiates a DOM engine that consists of the DOM dispatcher, HTML parser, CSS parsing and styling, and timers and events. The DOM dispatcher thread schedules DOM updates and serves as the page event loop. It serializes access to the DOM and manages the interaction between components.

The rest of the browser infrastructure dispatches work items to the concurrent DOM dispatcher queue, and the items are then handled one at a time. Work items represent browser passes as well as events from timers and the user interface. The HTML parser receives incoming (partial) data chunks for an HTML document via a DOM dispatcher work item and constructs the DOM tree by executing the HTML5 parsing algorithm. The parser adds external resources (referenced from the HTML document) to the resource manager's fetch queue. The parser also initiates the execution of JavaScript code by calling the JavaScript engine at appropriate times during parsing. The CSS engine calculates the look and feel of the DOM elements for the later layout and rendering stages. Similar to image decoding, the resource manager hands off CSS stylesheets to the CSS engine for parsing and for discovering new resources to request.

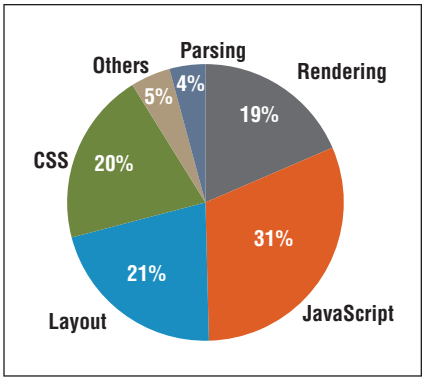

Figure 1. Browser processing times by component, excluding network load time. Profiling results obtained using the WebKit browser on a four-way ARM Cortex-A9 processor. Results are an aggregate of the top Alexa 30 sites as of March 2010.

Rendering engine. Whenever the DOM or CSS stylesheets change—because the fetcher delivered new resources, the HTML parser updated the DOM, or as a result of JavaScript computations—this change needs to be reflected on the screen so that the user can view and interact with it. The layout engine is responsible for transforming the styled DOM tree into geometry and content, which the rendering engine can turn into a bitmap. Ultimately, this bitmap is displayed on the screen by the user interface as a viewable webpage. Normally, the layout and rendering engine takes a snapshot of the DOM information it needs and performs the rest of the work asynchronously; however, it can also be invoked synchronously when JavaScript use APIs that query layout information.

JavaScript engine. The Zoomm employs a novel JavaScript engine, MuscalietJS, for executing all JavaScript code. The engine's design is presented in detail elsewhere (http://github.com/mcjs/mcjs.git).[2] In particular, our engine exploits concurrency by compiling multiple scripts in parallel, as well as compiling scripts asynchronously with the rest of the browser passes.

SMARTPHONES

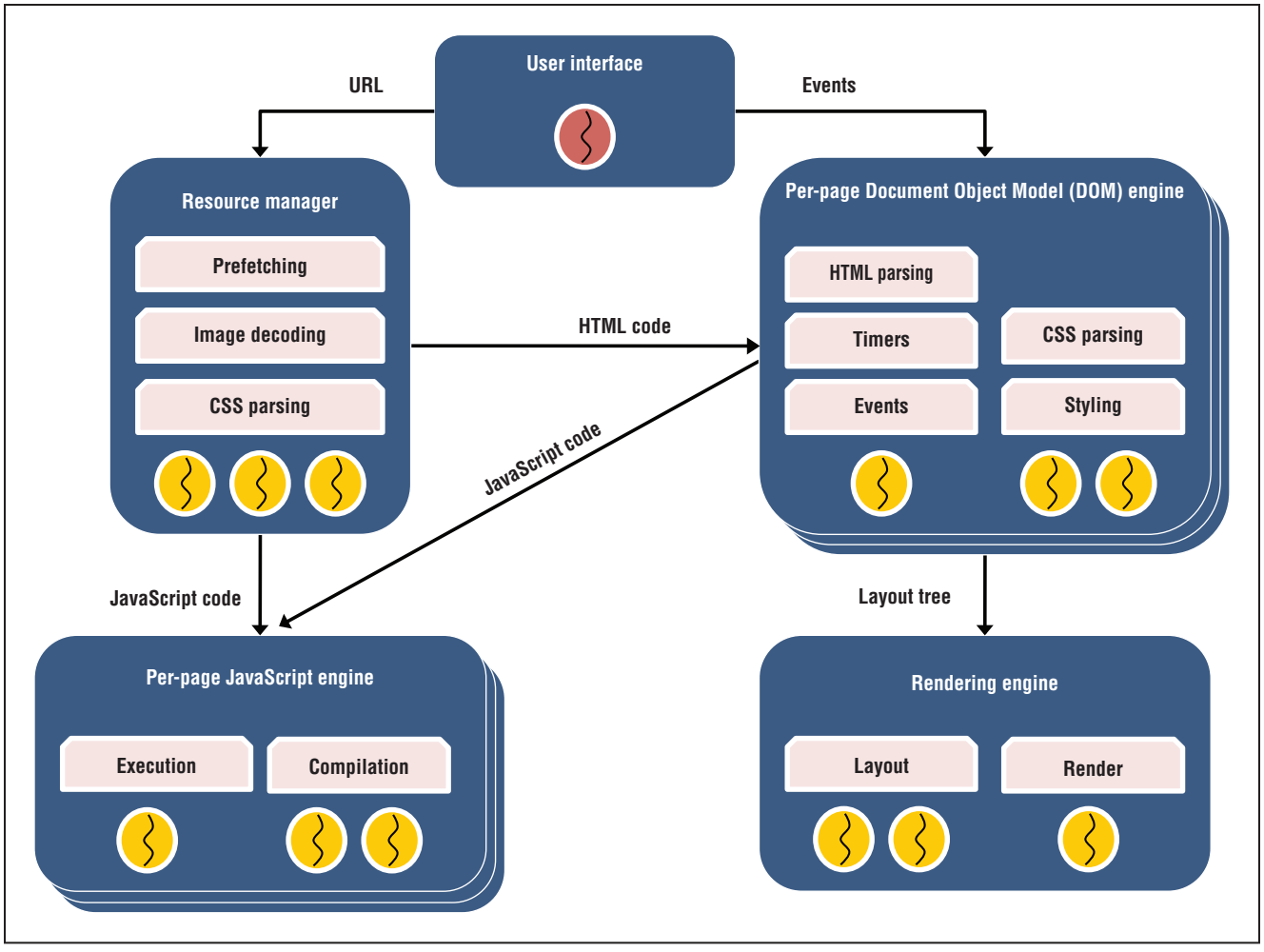

Figure 2. The Zoomm browser architecture. Concurrency is exploited both across components and within each component.

To achieve this, the JavaScript engine uses a thread pool and the just-in-time compiler uses a separate state stored in the metadata of each script. Due to JavaScript semantics, the execution of scripts is performed sequentially in the main engine thread. When the HTML parser or DOM dispatcher (for example, for user interface events) requests the execution of a JavaScript script that has not been compiled already, compilation is initiated. In either case, the engine waits for the compiled result and then executes the script. The goal of the engine is to use available resources on the platform to improve the generated code for JavaScript execution.

Similar to other modern JavaScript engines, MuscalietJS is a multitier execution engine. When the number of times a function has been executed exceeds a certain threshold (in other words, it's "hot"), the engine will promote the function and recompile it at a higher optimization tier. Different tiers include an interpreter, a baseline compiler, and a full compiler. The baseline compiler generates suboptimal code quickly. The full compiler, on the other hand, generates more optimized code for hot functions by performing adaptive JavaScript-specific optimizations, including hidden classes, property lookup, type specialization, and restricted dataflow analysis.

User interface. The Zoomm browser is implemented in platform-agnostic C++. For concurrency, we use a custom asynchronous task library (Qualcomm Multicore Asynchronous Runtime Environment; http://developer.qualcomm.com/mare), optimized for mobile execution. On Android, a thin Java wrapper is used to create the user interface. User interactions, such as touching a link on the display, are translated into Java Native Interface method calls, which ultimately create work items in the DOM dispatcher. Drawing to the display is performed using the Android Native Development Kit, which provides direct access to Android bitmaps. On Linux and Mac OS X, a similar wrapper is implemented in C++ using the Qt interface toolkit (www.qt.io/developers). Although our deployment

TABLE 1
Combined HTML and CSS prefetching initiates the download of most external resources ahead of their discovery by the HTML and CSS parsers with high accuracy ("correct prefetch") and small error ("missed/mistaken prefetch"). "Total resources" denotes the number of referenced resources in a webpage.

	Correct prefetch						Mistaken prefetch					
	HTML		CSS		Missed prefetch		HTML		CSS		Total resources	
Website*	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes
cnn.com	34	979,695	52	409,377	2	372	0	0	5	3,371	93	1,392,815
bbc.co.uk/news	54	610,479	24	407,819	16	468,371	0	0	1	1,277	95	1,487,946
yahoo.com	44	672,595	13	264,603	2	2,016	1	0	0	0	60	939,214
guardian.co.uk	49	1,018,738	14	92,997	7	102,087	1	0	3	11,305	74	1,225,127
nytimes.com	73	1,046,636	9	73,487	13	228,162	1	10,837	1	89	97	1,359,211
engadget.com	128	2,023,135	84	651,030	5	104,320	0	0	9	34,824	226	2,813,309
qq.com	45	485,264	22	167,078	7	39,361	0	0	0	0	74	691,703

*The websites are from the Vellamo benchmark.

targets are Android devices, the Qt implementation allows much easier debugging and testing on desktop-based machines, and the ability to evaluate concurrency beyond what Android devices currently offer.

PARALLEL EXECUTION FOR RESOURCE PREFETCHING

Mobile devices commonly experience high latency when requesting the resources that form an HTML document. To reduce the overall time taken to load a page, fetching all of the dependencies from the network as early as possible is very important.

HTML Prescanning

Due to idiosyncrasies in the HTML5 specification, the HTML5 parser must wait for <script> blocks to finish executing before it can continue parsing. So, if a webpage references an external resource after a script element, fetching the resource can't be overlapped with the waiting. This could delay the completion of page loading.

The Mozilla Firefox browser mitigates such situations by speculatively parsing ahead of script blocks to discover new resources. (It might then be forced to throw away some of that work if, for example, JavaScript inserts new content into the DOM tree via the document.write() API.) Once resources are discovered, network latency can be masked by requesting multiple resources to be fetched in parallel. This strategy also helps use all available bandwidth, and it reduces the overall time spent waiting for resources to arrive.

In Zoomm, we favor concurrency to achieve the same goal by running an HTML *prescanning* component in parallel with a (nonspeculative) HTML parser. The main objective of the HTML prescanner is to quickly determine all external resources in an HTML document and trigger their fetching from the network. The most commonly referenced resources are images, CSS stylesheets, and JavaScript sources. In addition, stylesheets and JavaScript sources can themselves reference further external resources. Furthermore, the prescanner obtains all id, class, and style attributes used in the document.

As network packets of an HTML document arrive, they are given to the prescanner and the actual HTML parser independently. The prescanner can run ahead of the HTML parser because it only has to approximately parse HTML to find resources, thus skipping the complex DOM tree construction phase. More importantly, the prescanner doesn't have to wait for the execution of <script> blocks to finish.

The processing of prefetched resources works as follows. Images are fetched concurrently with the rest of the page processing. Once downloaded, image data is given to a thread pool for decoding concurrently. The decoded image is added to the DOM dispatcher queue, which updates the corresponding img tree node. Then the image is removed from the set of pending images.

CSS Prefetching

CSS stylesheets are dispatched to a thread pool responsible for parsing CSS concurrently. If a CSS rule contains additional external resources, the parser decides whether to initiate prefetching for them, based on the likelihood that they're actually referenced in the HTML document.

It's crucial to download just enough of the referenced resources. Downloading too little means that new resources are discovered only when styling the DOM tree later on, which incurs additional latency penalties. It's common practice among websites to reference many more resources than are actually needed for any given document—for example, by using a site-wide common style file. Downloading

SMARTPHONES

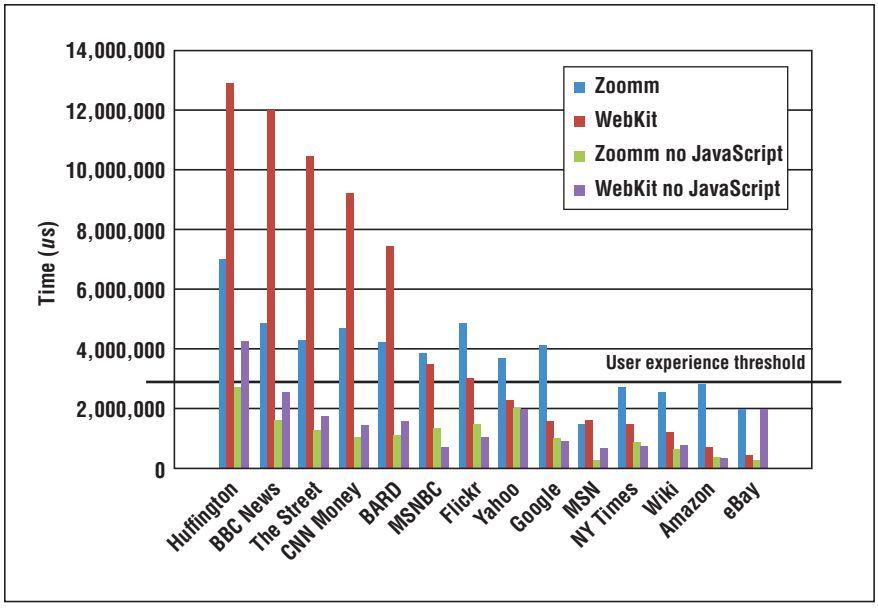

Figure 3. Page load time for several popular sites. Note that users typically expect pages to load in less than 3 seconds.

all resources invariably consumes too much bandwidth and slows down page loading.

In Zoomm, the CSS parser employs the id and class attributes discovered by the HTML prescanner to determine if a rule is likely to be matched. If all attribute values referenced in a CSS rule selector have been seen by the HTML prescanner, we assume that the rule will match at least one DOM tree element and initiate downloading its resources. This heuristic is simple but effective (see Table 1). Note that wrong decisions don't affect correctness; any missed resources will be discovered during the styling phase, at the cost of additional latency.

Table 1 shows the number of resources that are successfully requested by the prefetching stage, and the number of resources are missed due to use of JavaScript. Note that resources would also count as "missed" if the prefetching algorithms would fall behind the actual HTML and CSS parsers. However, this was never the case in all our experiments. The prefetching components were always fast enough to finish much earlier than the parsers.

Despite the heuristic nature of some of the prefetching decisions, they're quite accurate. In our experiments, 80–95 percent of all externally referenced resources in a document were prefetched correctly, with only a small error rate. Due to bandwidth and power considerations, our heuristics were still conservative—that is, they tend to prefetch too little rather than too much. The "missed prefetch" (not prefetched, but needed for rendering the webpage) numbers were higher than "mistaken prefetch" (prefetched, but not needed for rendering) numbers.

JAVASCRIPT PARALLEL PROCESSING

In modern pages, a significant number of resources (style sheets, images, and other scripts) are dynamically constructed using JavaScript. It's advantageous to discover these resources ahead of time, such that their download doesn't block the page load. HTML5 introduces two attributes for scripts: *async* and *defer* to allow out-of-order processing of scripts. When the HTML parser encounters one of these attributes, it can farm out its compilation and execution to the JavaScript engine immediately. MuscalietJS takes advantage of the asynchronous semantics and compiles and executes these scripts in parallel.

Another technique for exploiting multicore processing for JavaScript is parallel compilation. Almost all current browsers use parallel compilation to either compile multiple scripts concurrently or run an enhanced compiler in a separate thread.[5–7]

Overall, using these parallelization techniques, Zoomm loads pages about twice as fast as WebKit, as shown in Figure 3.

Exploiting parallelism in browsers promises performance and power savings. We believe that Zoomm is just a first step in that direction, and hiding network latency using ahead-of-time processing removes a bottleneck in loading webpages that is beyond the control of browser clients, thus improving the user experience. Optimizations explored in Zoomm and the MuscalietJS engine are being adopted by commercial browsers: the Mozilla Servo project (https://github.com/servo/servo) is using a parallel language (RUST) to implement a concurrent browser architecture similar to Zoomm's. That project puts a larger emphasis on the layout engine to handle all the corner cases of the HTML5 specification, which presents a significant challenge and opportunity.

Browsers such as Chrome and Internet Explorer are implementing parallel JavaScript processing, and recently Chrome has decoupled JavaScript parsing into a concurrent thread.[5] Other researchers are looking at architectural aspects of enabling more concurrency in the browser.

Finally, Web standards are evolving to allow webpage designers to exploit

concurrency. These include asynchronous and deferred script processing directives in HTML, Web workers, and several efforts to express concurrency in JavaScript. In addition, the declarative nature of CSS makes it ripe for exploiting parallelism through concurrent implementations.

ACKNOWLEDGMENTS

We thank Nayeem Islam and the Qualcomm Research Executive team for the opportunity to build the Zoomm and MuscalietJS engines. We thank Mehrdad Reshadi, Michael Weber, Wayne Piekarski, Seth Fowler, Vrajesh Bhavsar, Alex Shye, and Madhukar Kedlaya for their contributions.

REFERENCES

1. C. Caşcaval et al., "ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices," *Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP), 2013, pp. 271–280.
2. B. Robatmili et al., "MuscalietJS: Rethinking Layered Dynamic Web Runtimes," *Proc. 10th ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments* (VEE), 2014, pp. 77–88.
3. Z. Wang et al., "Why Are Web Browsers Slow on Smartphones?" *Proc. ACM Int'l Workshop on Mobile Computing Systems and Applications*, 2011, pp. 91–96.
4. L.A. Meyerovich and R. Bodík, "Fast and Parallel Webpage Layout," *Proc. Int'l Conf. World Wide Web*, 2010, pp. 711–720.
5. M. Hölttä and D. Vogelheim, "New JavaScript Techniques for Rapid Page Loads," blog, 18 Mar. 2015; http://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html.
6. J.-D. Dalton, G. Seth, and L. Lafreniere, "Announcing Key Advances to Javascript Performance in Windows 10 Technical Preview," blog, Oct. 2014; http://blogs.msdn.com/b/ie/archive/2014/10/09/announcing-key-advances-to-javascript-performance-in-windows-10-technical-preview.aspx.
7. J. Ha et al., "A Concurrent Trace-Based Just-in-Time Compiler for Single-Threaded JavaScript," *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures* (PESPMA), 2009, pp. 47–54.

Călin Caşcaval is a senior director at Qualcomm Research Silicon Valley. Contact him at cascaval@qti.qualcomm.com.

Pablo Montesinos Ortego is a senior staff engineer/manager at Qualcomm Research Silicon Valley. Contact him at pablom@qti.qualcomm.com.

Behnam Robatmili is a staff research engineer at Qualcomm Research Silicon Valley. Contact him at behnamr@qti.qualcomm.com.

Darío Suárez Gracia is a staff engineer at Qualcomm Research Silicon Valley. Contact him at dgracia@qti.qualcomm.com.