WebOptProfiler: Providing performance clarity for Mobile Webpage Optimizations

Ghulam Murtaza
Brown University

Theophilus A. Benson
Brown University

ABSTRACT

Despite decades of research on mobile webpage optimizations, little is known about how these optimizations interoperate. Moreover, there has been little systematic work to understand the scenarios wherein combinations of these optimizations excel. Without a comprehensive understanding of how these optimizations compose with each other and under what conditions they excel, operators cannot determine which optimizations to adopt, and, similarly, developers do not know where to focus their efforts.

In this paper, we argue that developers should be required to evaluate and characterize the broader interactions between their proposed optimizations and other optimizations – this is in addition to demonstrating the potential benefits of their approach. To aide developers in characterizing these broader interactions, we propose an analytical model which decomposes web optimizations into virtual speedup functions that operate on well-understood browser processing phases (e.g., processing, rendering, layout, etc., for an object) and we present a web browser-oriented causal profiler which empirically explores interactions between optimizations by using their analytical models to speed up different parts of the Browser during a page load. Our system, WebOptProfiler, identifies and addresses practical issues in extending causal profiling to the webpage optimization domain and provides an algorithm for extracting an analytical model from readily available browser traces.

ACM Reference Format:
Ghulam Murtaza and Theophilus A. Benson. 2021. WebOptProfiler: Providing performance clarity for Mobile Webpage Optimizations. In *Proceedings of (HotMobile).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

Despite the growing ecosystem of mobile webpage optimizations [4–8, 14–18, 22–25, 27, 28], most efforts are often designed and developed in isolation. In particular, we have no avenues to answer the following questions: How do mobile webpage optimizations compose with each other? Are their interactions multiplicative or additive? Do they interact positively? Or, do they cancel each other out? Moreover, while we continue to churn out new mobile-centric optimizations, it is unclear whether the current optimizations have constructive or destructive interactions or whether, as a community, we are getting closer to the optimal scenario.

The most direct approach to answer these questions is to deploy the different optimizations and empirically analyze the different combinations. However, directly performing experiments will require the arduous task of getting many of these optimizations to interoperate with each other – a non-trivial task. Furthermore, this will require us to explore them on a huge corpus of websites.

A promising alternative is to analytically model the optimizations and explore the interactions between their models. While mobile webpage optimizations naturally lend themselves to analytical models [13, 19, 26], the browser, which ultimately loads the website and composes the results of different optimizations, does not necessarily lend itself to analytical modeling [26]. Specifically, while the browser sequentially processes web objects that interact with the DOM, the exact order is not well understood. Additionally, certain aspects are multi-threaded. Together, these observations imply that analytically modeling any browser is a cumbersome and error-prone endeavor.

Instead, we argue for a blend of analytical modeling with empirical analysis. Our combination builds on the following insights: the mobile webpage optimizations are generally meant to speed up the processing of one or more objects and be extracted by analyzing versions of a page with and without the optimizations. Given these speedup factors, we can empirically analyze the optimizations by using emerging causal profilers [11, 19] – tools that allow developers to reason about the impact of arbitrary speedups of specified functions on the application's performance.

Our system, WebOptProfiler, uses browser traces, e.g., DevLog, from webpage loads with (and without) the optimizations to construct an analytical model of the optimization. Different developers can then use these models to systematically evaluate interactions between various optimizations by using WebOptProfiler's causal profiler. Our approach allows engineers to thoroughly investigate and understand the payoffs before undergoing the more arduous task of re-implementing and interfacing with these optimizations with each other. In addition to enabling engineers, our tool has an added benefit for optimization designers; it provides them with insights that allow them to better adapt their optimization techniques to align with other optimizations.

Although many of the components which comprise WebOptProfiler are readily available (i.e., web page optimization modeling and causal profilers), WebOptProfiler presents a unique combination of these techniques, which introduces new challenges. For example, while there is a growing body of work on causal profilers [11, 19], these approaches are optimized for uniform speedups of a single function. However, webpage optimizations impact objects differently, and when applied together, they may require speeding up multiple functions. Additionally, traditional causal profiling focuses

HotMobile, 2021

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI:10.1145/3446382.3449073

on the applications, whereas webpage optimizations speed up functions within the application (i.e., the browser) and the network. Our framework's novelty lies in extending causal profiling to (i) control interactions over the network by embedding a web page replay and record framework and (ii) scalably and efficiently provide fine-grained function-level control over virtual speedups.

This paper makes three key contributions to providing performance clarity on the growing space of web optimizations.

- First, we design a method for analytically modeling mobile webpage optimizations as a matrix that defines speedups for specific processing stages of loading web objects.
- Second, we develop a causal profiling framework for empirically analyzing the performance (i.e., page load time (PLT) changes) of different webpage optimizations by employing combinations of our analytical models to virtually speedup specific aspects of the page load process.
- Third, we create an integrated system that systematically extracts optimization models from page load traces and provides a platform to execute them to collect performance measurements.

2 BACKGROUND

In this section, we provide background on the web page loading process (§ 2.1), present an overview on current web optimizations (§ 2.2), and on causal profiling (§ 2.3).

2.1 Browser Workflow

In Figure 1, we present the high-level workflow of a browser and highlight the phases involved in a typical web page processing pipeline. The workflow is generally divided into four broad phases [13, 19, 26]:

(1) Networking phase, which includes fetching objects over the network using the Resource Loader module.
(2) Processing phase which deals with parsing and evaluating the objects and coordinates their interactions with the DOM tree. Note: As a result of processing an object, the browser may direct the ResourceLoader to fetch other web objects (e.g., processing an HTML object may lead to fetching images). The browser includes distinct modules for parsing the various objects.
(3) Layout phase, which consists of the construction of Render Tree and Layout Tree from the DOM tree.
(4) The painting phase, which includes compositing different layout layers in the Layout tree and painting them on screen. The browser's rendering engine generally does this.

The browser may parallelize the networking phase and portions of the processing phase for different objects. However, to maintain consistency and correctness, each object is processed sequentially during the other phases that involve DOM manipulation. This combination of sequential and parallel processing complicates browser modeling and performance analysis.

2.2 Mobile Web Optimizations

Web page performance optimizations can broadly be categorized into three categories; client-side, server-side, and network. These

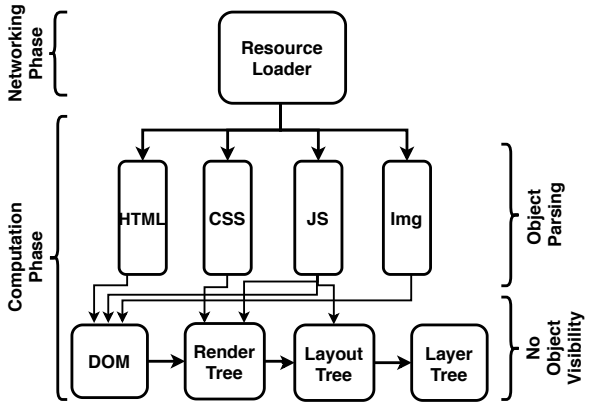

Figure 1: Browser workflow for page load process. Highlighting the different phases and level of visibility.

optimizations aim to improve web performance by altering the scheduling [8, 15], compression [6, 23], placement/location [18, 21, 22, 24], format [5], and pre/post-processing [4, 16, 27, 28] of one or more group of web-objects. We note that while all modern mobile webpage optimizations target web objects, their transformations are based on different assumptions about the critical path or the bottleneck resources. For example, compression optimizations [6, 23] assume the network is the bottleneck and consequently optimize network performance (e.g., transfer time).

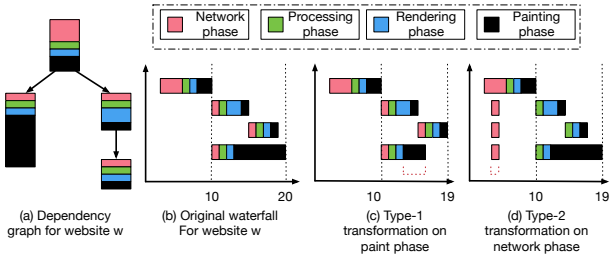

Figure 2: Illustration of transformations on an abstract webpage.

In Table 1, we present a representative list of these optimizations and highlight the objects that they modify/transform and the specific resource they focus on. Abstractly, we can think of each of these optimizations as functions, which takes as input (1) an abstract version of a website, (2) client conditions (e.g., network or mobile device type), and alters the semantic or the syntactic properties of one or more types of objects. These functions produce as output an optimized set of web objects to be loaded by the browser.

The transformations can be classified into three types:

- Type-1 performs semantic transformations on individual objects, e.g., compression, transcoding, minification, or duplicate code elimination. Figure 2 (c) presents an example of one such transformation altering the page load performance by reducing the processing time of a specific phase (in this case, the painting phase).

Table 1: Web Optimizations

Optimization Type	Algorithms	Resource	Optimization Sumary
Assisted Rendering	Opera Mini [4], Cumulus [17], Prophecy [16], Shandian [27], Fawkes [14], Snapshot Based Acceleration [28]	CPU Utilization	Objects precomputed on a remote server, to save CPU on client device.
Compression Proxy	Flywheel [6], Flexiweb [23], Weblight [5]	Network Utilization	All objects are compresses to save network bandwidth
Object Scheduling	Polaris [15], Klotski [8]	CPU/Network Idle Time	Objects scheduled in a way to minimize CPU and Network Idle times.
Object Push Proxy	Watchtower [18], APPx [10], WebPro [22], Vroom [21], Nutshell [22]	Network Idle Time	Dependent objects pushed to reduce network Idle times and to eliminate dependencies between Network and CPU.

- Type-2 modifies the scheduling of specific phases of an object. The most popular alter the scheduling phase of when the "network" processing of an object takes place, e.g., H2 server-push, H2 priorities, client-prefetch, caching, and dependency modifications (e.g., polaris [15],VROOM [21]). Figure 2 (d) presents an example of one such transformation altering the page load performance by allowing the networking phase to overlap, increasing concurrency, and speeding up the entire page load process.
- Type-3 is a special case of Type-1 where multiple objects are semantically transformed together, e.g., server-side rendering or server-side preprocessing of javascript (e.g., Prophecy [16]), which merges the pre-computed JS with base HTML. In essence, we can classify any mobile web optimization (client, network, or server-side) into one of the three types mentioned above.

Takeaway: Based on the categorization of web optimizations, we infer the following key insights about web optimizations: The impact of any given optimization is localized to a well-defined class of objects or well-defined phases of the page load process. For example, the compression optimizations discussed earlier impact the networking phase and object parsing phase. Moreover, the browser's combination of sequential and parallel processing implies that reasoning about different optimizations' behavior is non-trivial. In particular, combining optimizations is not a strictly linear combination of their speedups, and understanding the benefits of optimization may be non-trivial.

2.3 Causal profiling

Causal profiling [11, 19] is a new form of software profiling that differs from traditional profiler in that while conventional profilers capture the amount of time spent in different parts of the code, causal profilers selectively speed up different parts of code to estimate the benefits of optimizing them. In essence, traditional profilers help developers understand performance characteristics, while causal profilers enable developers to speculate on the implications of optimizing different code pieces.

Causal profilers speed up code by employing a novel technique, called *virtual speed ups*, which systematically slows down other code snippets. Specifically, this technique identifies other code segments or functions which run concurrently as the target code snippets but on other threads. The technique subsequently slows down these identified code-segments by adding pauses that are proportional to the anticipated speedup. These slowdowns are effective because slowing down everything else except the optimized function (or code) has the same impact as optimizing the target function to run faster.

Our goal is to understand the broader interactions between mobile webpage optimizations, which effectively speed up processing within specific browser functions. We believe that causal profilers stand out as ideal candidates for correctly analyzing their impact on the page load process. Unfortunately, causal profilers are limited in the following ways:

First, they operate at the granularity of an entire application, e.g., arbitrary speedups for all objects processed by the web browser's target function(s); however, web optimizations, in contrast, provide speedups for specific objects (i.e., speedups are dependent on the inputs). Second, while traditional speeds up are constant and fixed, the speedups for web optimizations are a function of the object's properties. This implies that the code we would want to speedup is also a function of the inputs and their properties. Third, the existing profilers only operate on software (in our case, the browser) and have no control over the network (i.e., object transfers). This implies that causal profiling is insufficient to analyze the broader range of optimizations that target the network. Finally, they only speed up one function at a time; whereas, single web optimizations may impact multiple functions.

To illustrate these limitations, consider a type-1 optimization (specifically compression-proxy [6, 23]). While the proxy may try to compress all objects, each object has a different compression factor, and hence a different reduction in its network transfer time. To faithfully analyze this with a causal profiler, we need the profiler to support per-object speedups with varying speedup values per object.

Takeaway: In conclusion, these limitations imply a need for a broader and finer-grained causal profiling system that provides: (1) The flexibility to selectively speed up code phases, (2) A new virtual speedup logic that allows potentially every piece of code to have distinct speedup factors, and (3) A new method for controlling interactions with the network to apply speedups on the networking phase.

3 WEBOPTPROFILER

In this section, we discuss WebOptProfiler our framework for scalable profiling and analyzing the interactions between mobile web optimizations on arbitrary webpages. In Figure 3, we present the workflow of our framework. WebOptProfiler operates on Chrome's DevTools logs [2] which Chrome generates while loading websites. These traces provide timing information about when phases start and end and for the subtasks within each phase. WebOptProfiler analyzes a pair of traces from loading a website with and without the web optimization, to extract a model (§ 3.1.2) which captures the performance profile of the optimization. Given such models for different mobile web optimizations, WebOptProfiler enables operators to analyze their cumulative impact by virtually speeding up appropriate sections as dictated by the model. To support client-side (i.e., browser) speedups, WebOptProfiler extends causal profiler to address challenges identified in § 2.3, and to support network speedups,

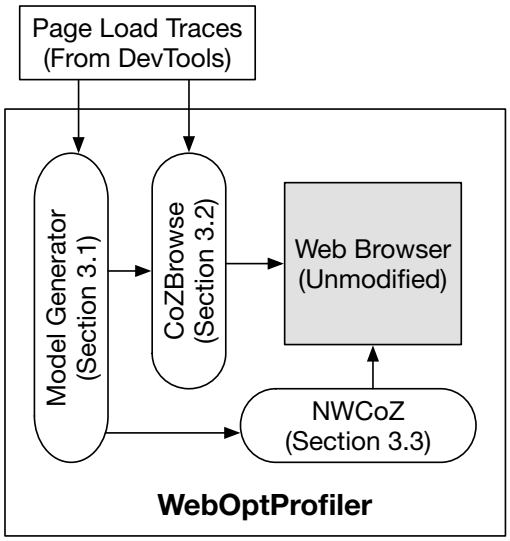

Figure 3: WebOptProfiler Architecture.

the profiler is enhanced with a proxy-based web record and replay tool to provide network-centric speedups (§ 3.3). The novelty in our framework are: first, the insight that optimizations can be analyzed using a fusion of empirical and analytical modeling and second, enhancements to the causal profiling framework to enable selective and arbitrary speedups of both compute (§ 3.2) and network (§ 3.3).

3.1 Model Generator

To effectively speed up a website, WebOptProfiler requires a model (§3.1.1) for abstracting a webpage into an analytical representation of the interactions between the different phases of the page load process. Such a model enables WebOptProfiler to express optimizations as speedups of different phases of different objects (§3.1.2): this abstraction naturally lends itself to WebOptProfiler's underlying causal profiling engine.

3.1.1 Analytical Modeling of Websites. We abstractly model a website w as a directed dependency graph $G = (O,E)$, were $o_i \subset O$ are the objects, and an edge $e_i = (o_i,o_j)$ implies that o_i must be processed first before o_j can be processed. The objects are typed: for example, $O_{css} \subset O$ is the set of CSS object in w. [1] Moreover, each object, $o_i \subset O$, can be characterized by four distinct functions: one for each of the four phases in the browser page load pipeline. We define $Proc_{s,i}$ to be the amount of time that a phase s (e.g., network loading, processing, rendering, or painting) uses to process object i (o_i).

We construct this directed dependency graph G by analyzing Chrome's DevTools logs [2], which captures fine-grained tracing information regarding the time spent by the browser to process each object across all the page load phases. Devtools logs also specify each object's requestor (i.e., the object that triggers another object) – this information can be used to construct dependencies. Note, the logs also include the processing time for functions in each phase, and we can aggregate this information to determine the phase processing times, $Proc_{s,i}$.

[1] Similarly, O_{html}, O_{JS} are, respectively, the set of HTML and JS objects in w.

3.1.2 Modeling Web Optimizations. To create a Web optimization model, we must capture and compare two graphs, G_{base} with $G_{optimized}$, which are models of a page with and without the optimization of interest. The output of this comparison is a matrix that indicates the relative speedup for each phase of each object in the website.

3.2 CoZBrowse

CoZBrowse is an enhanced version of the traditional CoZ+ profiler [19] with support for fine-grained profiling. In particular, CoZBrowse enables us to selectively speed up different objects (§ 3.2.2) using different combinations of speedup values (§ 3.2.1).

3.2.1 Different Speedups for Phases. The current architecture can only virtually speed up one function (or code segment). In the current setting, the slowdown factor is the same as the virtual speedup factor just applied in reverse order – thus, (in Figure 4) to speedup function 'a()' by 2ms, a causal profiler slows down 'b()' and 'c()' by 2ms. To virtually speedup multiple functions with different speedup factors (e.g., in Figure 4 to speedup only functions 'a()' and 'b()' by 1ms and 3ms respectively), we need to redesign the slowdown calculation function. Intuitively, virtual speedups, and by extension, slowdowns are cumulative: we can offset the relative speedups. In Figure 4, 'b()' is only being sped up by a factor of two relative to 'a()', and, thus, to speed up 'b()', 'a()' must be slowed down even though it is also being sped up. The key to correctly calculating the slowdown factor is to carefully analyze the virtual speedups, determine their relative speedups, and convert this into slowdowns used by the profiler. We achieve this in three steps:

(1) First, we analyze all the virtual speedups to identify the function with the max speed (in Figure 4, this is 'a()').
(2) Second, we determine the relative speedups of the identified max function relative to the other functions (in Figure 4, the relative speedup 'a()' relative to 'b()' and 'c()' are 2 and 3 respectively).
(3) Finally, we convert this into slowdown factors (thus, 'b()' and 'c()' are slowed down by 2 and 3, respectively Figure 4(c)).

3.2.2 Individual Object speedups. The most intuitive method to provide selective and conditional speedups is to perform a quick check to determine which object is being processed and only perform a virtual speed up if the object being processed is affected by the optimization. For performance reasons, causal profilers operate at a much lower level, and thus they do not have access to object names. Alternatively, we could maintain a mapping from object name to the memory location where the object is stored and instead perform our checks by examining the functions' memory addresses. Unfortunately, this approach falls short when objects are copied.

In our design of CoZBrowse, we explore a different approach that builds upon the web domain's unique properties. Namely, web replay techniques, i.e., NWCoZ, are deterministic, and the browser sequentially processes most objects to ensure that the DOM remains consistent. Given these insights, we record the processing order of objects and use this recorded ordering to determine when to perform a speedup (and when not to), e.g., on the third call to 'a()', it

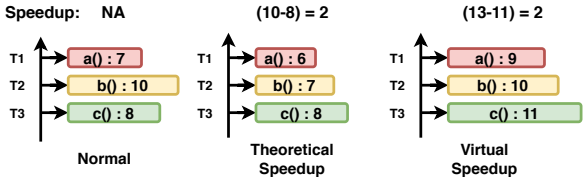

Figure 4: Multiphase speedup example: By our method, max slowdown is 3 seconds (a:1, b:3, c:0), so a is slowed down by $3-1=2$ seconds, b is slowed down by $3-3=0$ seconds and c by $3-0=3$ seconds. After applying this slowdown, our program executes in 11 seconds. We compute speedup by subtracting 11 from 13 which is actual runtime + MaxSlowDown.

processes 'test.css', thus we speed up 'a()' by 2ms on its third call. While primitive, this approach is simple, efficient, and correct.

3.3 NWCoZ

To extend causal profiling into the network, WebOptProfiler needs to virtually speed up network transfers, which requires altering a combination of bandwidth and latency. To realize this, WebOptProfiler includes a proxy-based webpage record and replay framework called NWCoZ. By recording and storing a web page on the proxy, NWCoZ can apply arbitrary speedups or alter the network transfer times during the replay process. While many webpage replay techniques, e.g., MahiMahi [3, 9, 17], exist, they do not provide object-level control over networking conditions during replay. On the other hand, existing Linux network emulation tools, e.g., Linux TC, are too fine-grained, i.e., they operate at the packet level, not the object level. The fundamental challenge in extending existing tools to provide object-level control is to ensure efficiency and fidelity while being flexible. Our observation is that the single largest overhead of traffic profiling arises from per-packet traffic classification. Thus, we avoid performing per packet classification by having the replay tool's network stack specify the train of packets associated with the object. This can be done, for example, by adding special bits into pre-existing packet headers. We observe that object-level classification is appropriate for our purposes as the web optimizations operate on the object level (§ 2).

4 PRELIMINARY EVALUATION

We have designed a preliminary version of WebOptProfiler for Chrome. We implemented the part of the model generator module in python with 300 lines of code, which constructs a matrix of speedups for each object by comparing the run time of each stage of each object. Similarly, we extended MahiMahi with 200 lines to support optimization models' ingestion and apply per object network speedups. We use our partially developed system to emulate a content minification optimization, which is a subset of standard compression optimizations [6, 23], because we need an optimization whose impact is strictly confined to the networking phase. We collect our measurements on the top four websites from Scratchpad [1], which contains a repository of publically hosted HTTP websites. We run each website 10 times with a 100ms delay and 2Mbit bandwidth.

We summarize our findings in Figure 5. As the figure shows, our system can accurately represent a simple minification optimization.

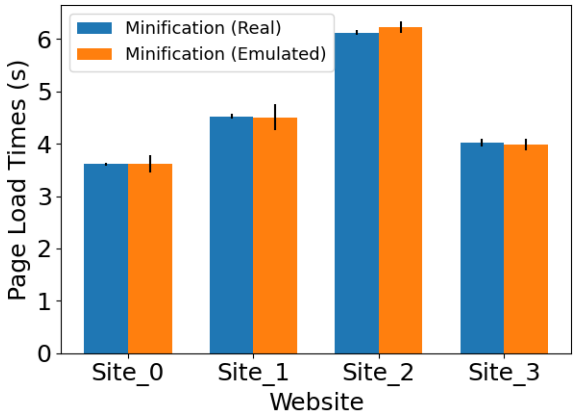

Figure 5: Analysis WebOptProfiler's accuracy

5 RELATED WORK

The closest related work is on the causal profiling of distributed systems [11] and web browsers [19]. The key distinctions are that while prior work explores causal profiling at the system's granularity, we explore at a fine-granularity, namely at the granularity of the system's inputs (i.e., web objects). This shift in granularity requires reifications of traditional causal profiling tools and developing a composable web optimizations model, which allows us to map existing optimizations to our enhanced causal profiling tool systematically.

Most attempts to analyze and perform "what-if" analysis within the webspace [13, 26] explore a static analysis of the browser either by instrumenting the browser [26] or by modeling the browser's traces [13]. Regardless, the browser's multithreaded-ness and parallelism limits their applicability.

6 DISCUSSION AND OPEN CHALLENGES

This paper lays out a vision for providing performance clarity to the growing jungle of mobile web optimizations by offering an initial approach for modeling and analyzing these optimizations: In a sense, we provide necessary mechanisms for analysis. However, to meaningfully use our new mechanisms, we need to address the following issues: developing carefully balanced benchmarks for comparative and representative analysis (Section 6.1), defining policies for composing optimizations (Section 6.2), and designing a useful model for device speedups (Section 6.3). Next, we elaborate on these open challenges.

6.1 Representative Benchmarks

Our work's key strength is that it allows developers to create and share analytical models; however, such models are meaningless if they are generated using arbitrary websites. On the one hand, generating models using arbitrary websites introduces bias, and also the lack of standardization implies that models may be incongruent with each other. On the other hand, generating models for every website is wasteful and prohibitively expensive.

We plan to generate a *representative* benchmark of websites, i.e., WebBench, that provides *coverage* over defining features of modern website. We envision that similar to TPC-C, our benchmarks will capture various aspects of a webpage and provide different subcategories for different types of webpages, e.g., a benchmark for progressive webpages versus a benchmark for AMP-specific pages. The key challenge in realizing these benchmarks is defining appropriate features and subfeatures. While the main features, e.g., object type and webpage structure, are obvious, the subfeatures are less obvious. For example, while the subfeatures for images naturally include resolution and encoding format, the obvious subfeatures for javascript, e.g., lines of code or cyclometric complexity, are less relevant as they do not meaningfully impact many (or any) optimizations. Given appropriate features (and subfeatures), we can determine a compact benchmark of websites that provide coverage and representativeness by investigating and tailoring an appropriate clustering algorithm to our domain.

6.2 Composing Optimization Models

Theoretically, we can combine, or compose, two optimizations by merging their matrices. However, the appropriate method for merging is dependent on domain knowledge about the semantics of the optimizations being composed. In particular, merging can be direct and straightforward, provided that the optimization operates on non-overlapping phases. For example, composing an image transcoding optimization and an HTTP/2 push optimization can be as simple as combining their matrices because they operate on different aspects; while the former impacts transfer time, the latter impacts the start time of the transfer. However, when the optimizations operate on overlapping phases, we need semantic information to understand how to combine their matrices. For example, composing an image transcoding optimization with a compression optimization is non-trivial as they are likely to operate on overlapping phases, and we need additional semantic information to determine an appropriate method for composition. We can automatically infer this semantic information by analyzing the interactions between this composition on our representative benchmarks (Section 6.1); however, this requires the design of algorithms to detect correlations. Alternatively, we can provide several explicit composition operators, e.g., "min(), max(), sum()", which the developer must select to allow our framework to perform composition. We plan to begin with the more direct approach, i.e., explicit operators, and then explore the model-based approach upon completion of our benchmarks.

6.3 Device Models

With the growing move to extend mobile performance to developing regions [12, 20], understanding how optimizations operate on a broad range of devices has become an important task. However, systematically testing on such a broad range is not always feasible. We plan to address this by developing device models that allow developers to reason about how differences in compute speed, memory size, and specialized hardware accelerators (i.e., domain-specific DSPs) impact an optimization's effectiveness. While compute speed is easy to model because it speeds up or slows down phases, memory and specialized hardware are trickier.

First, hardware accelerators are tailored for specific algorithms or code patterns and thus operate on a limited set of objects, e.g., domain-specific DSPs are designed to accelerate pre-specified image formats or video codecs. Thus, to model this hardware, we must design techniques first to infer the type of objects and then determine the speedup factor. Given a robust enough benchmarking suite, we can more directly infer models for such hardware.

Second, memory size indirectly impacts performance. On mobile devices, e.g., on Android, the size of memory impacts the frequency and duration of garbage collection events, and, unfortunately, during garbage collection events, the applications, i.e., browser, are paused, and pausing an application inflates its processing time. Thus modeling device memory requires understanding events that are outside the scope of the browser. These events are significantly harder to model because they occur at random times, and modeling at fine-grained to capture them can be cumbersome. We plan to investigate the design of statistical models based on correlations that leverage OS-level and application-level semantics to infer the relation between memory size and performance impacting events, e.g., garbage collection.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Nirupam Roy, for their invaluable comments. This work is supported by NSF grant CNS-1814285.

REFERENCES

[1] 2017. Sites list. (Mar 2017). http://scratchpads.eu/explore/sites-list
[2] 2020. Chrome DevTools | Google Developers. (2020). https://developers.google.com/web/tools/chrome-devtools
[3] 2020. mitmproxy is a free and open source interactive HTTPS proxy. (2020). https://mitmproxy.org/
[4] 2020. Opera Web Browser: Faster, Safer, Smarter. (2020). https://www.opera.com/
[5] 2020. Web Light: Faster and lighter mobile pages. (2020). https://support.google.com/webmasters/answer/6211428?hl=en
[6] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 367–380.
[7] Alessandro Baricco and Ann Goldstein. 2019. Silk. (2019). https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html
[8] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 439–453.
[9] Catapult-Project. [n. d.]. catapult-project/catapult. ([n. d.]). https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md
[10] Byungkwon Choi, Jeongmin Kim, Daeyang Cho, Seongmin Kim, and Dongsu Han. 2018. APPx: An Automated App Acceleration Framework for Low Latency Mobile App. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 27–40. https://doi.org/10.1145/3281411.3281416
[11] Charlie Curtsinger and Emery D. Berger. 2018. Coz: Finding Code That Counts with Causal Profiling. *Commun. ACM* 61, 6 (May 2018), 91–99. https://doi.org/10.1145/3205911
[12] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R. Das, and Michael Ferdman. 2018. Impact of Device Performance on Mobile Internet QoE *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3278532.3278533
[13] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. 2010. WebProphet: Automating Performance Prediction for Web Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, USA, 10.
[14] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *17th USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 20). USENIX Association, Santa Clara, CA, 879–894.

[15] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. USENIX Association, Santa Clara, CA.

[16] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 249–266.

[17] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, USA, 417–429.

[18] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. 430–443. https://doi.org/10.1145/3307334.3326104

[19] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. 2019. What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3 (06 2019), 1–23. https://doi.org/10.1145/3341617.3326142

[20] Ihsan Ayyub Qazi, Zafar Ayyub Qazi, Theophilus A. Benson, Ghulam Murtaza, Ehsan Latif, Abdul Manan, and Abrar Tariq. 2020. Mobile Web Browsing under Memory Pressure. *SIGCOMM Comput. Commun. Rev.* 50, 4 (Oct. 2020), 35–48. https://doi.org/10.1145/3431832.3431837

[21] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 390–403. https://doi.org/10.1145/3098822.3098851

[22] Ali Sehati and Majid Ghaderi. 2016. Network assisted latency reduction for mobile web browsing. *Computer Networks* 106 (2016), 134 – 150. https://doi.org/10.1016/j.comnet.2016.06.026

[23] Singh and et al. 2015. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*.

[24] Ashiwan Sivakumar and et el. 2017. NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 448–461. https://doi.org/10.1145/3117811.3117827

[25] Ingvar Stepanyan. 2019. Faster script loading with BinaryAST? (Aug 2019). https://blog.cloudflare.com/binary-ast/

[26] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, USA, 473–486.

[27] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, USA, 109–122.

[28] Jihwan Yeo, Changhyun Shin, and Soo-Mook Moon. 2019. Snapshot-Based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution. In *The World Wide Web Conference (WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2215–2224. https://doi.org/10.1145/3308558.3313575