2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

Characterization of Unnecessary Computations in Web Applications

Hossein Golestani, Scott Mahlke, Satish Narayanasamy
Department of Computer Science and Engineering
University of Michigan
{*hosseing, mahlke, nsatish*}*@umich.edu*

***Abstract*—Web applications are widely used in many different daily activities–such as online shopping, navigation through maps, and social networking–in both desktop and mobile environments. Advances in technology, such as network connection, hardware platforms, and software design techniques, have empowered Web developers to design Web pages that are highly rich in content and engage users through an interactive experience. However, the performance of Web applications is not ideal today, and many users experience poor quality of service, including long page load times and irregular animations.**

One of the contributing factors to low performance is the very design of Web applications, particularly Web browsers. In this work, we argue that there are unnecessary computations in today's Web applications, which are completely or most likely wasted. We first describe the potential unnecessary computations at a high level, and then design a profiler based on dynamic backward program slicing that detects such computations. Our profiler reveals that for four different websites, only 45% of dynamically executed instructions are useful in rendering the main page, on average. We then analyze and categorize unnecessary computations. Our analysis shows that processing JavaScript codes is the most notable category of unnecessary computations, specifically during page loading. Therefore, such computations are either completely wasted or could be deferred to a later time, i.e., when they are actually needed, thereby providing higher performance and better energy efficiency.

I. Introduction

Web applications play an important role in the daily life of many people, and they are widely used in both desktop and mobile environments for various purposes such as online shopping, navigation, and video streaming. Web pages are getting more and more complicated in order to provide content with a visually rich user experience. Although desktop and mobile processors have been constantly advancing in recent years, the quality of service delivered to Web users, especially in the mobile platform, is not satisfying yet as they may experience delays in showing the content of Web pages [14]. This is due to the fact that Web browsers are complex programs, which must process multiple languages (i.e., HTML, CSS, and JavaScript) and manage a wide variety of network transactions.

The quality of user experience depends on how fast the content of a Web page is displayed and how smooth one view transitions to another. In particular, both application designers (e.g., designers of Web browsers) and Web developers (i.e., Web page designers) should be aware that users' satisfaction relies on three distinct metrics: page load time, response time to user input, and animation smoothness [8]. Among these metrics, page load time is the most important one. In a study on more than 10,000 mobile Web domains [14], it was found that mobile websites load in 19 seconds on average with a 3G network and in 14 seconds on average with a 4G network. It was also observed that 53% of users left their browsing sessions if pages took longer than 3 seconds to load. This shows how deeply Web page load time affects user experience and highlights the need for performance improvement of Web applications.

Considerable effort has been put into improving the performance of Web applications both in academia and industry. Commercial Web browsers are continuously improved by leveraging complicated algorithms [9], [11], [15] and utilizing GPUs as accelerators [4], [5]. Web developers are also provided with advanced libraries and design tools [16], [17] for carefully managing services and ordering the resources. Prior academic work has tried to optimize Web browsers in different ways. [30] and [39] target Web page load time by prefetching and caching of resources and reordering of resources, respectively. Other proposals include enhancing or parallelizing the JavaScript engine [18], [25], [28], [29], proper scheduling of CPU cores [31], [36], [41], [42], [44], and designing specialized hardware [21], [22], [43].

In this work, we argue that in current Web applications–Web browsers in particular–there exists unnecessary computations, which are completely or most likely wasted. These unnecessary computations are caused by processing codes that are never used, pitfalls in the design of Web applications, or producing output that is never or most likely not noticed or used by the user. More details regarding potential sources of unnecessary computations are provided in Section II. Next, we develop a profiler that effectively identifies portions of Web browser computations that are important to the user (e.g., generating display pixels and network outputs), and analyzes the computations that do not belong to this portion (e.g., the unnecessary computations). The unnecessary computations are either completely useless, or done at improper time, so that they could be deferred to a later time when they are actually needed. Therefore, the designed profiler could be leveraged to both identify wasted computations and

978-1-7281-0746-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ISPASS.2019.00010

also reveal opportunities to optimize performance and energy efficiency of Web applications.

Our profiler is based on dynamic backward program slicing, and it works on the instruction and memory traces collected while a Web browser renders a Web page. The main slicing criteria are the pixels buffer at points where it contains the final values of pixels that are going to be put on the device display. While going backwards, the profiler identifies instructions whose execution has any effect on the values stored in the pixels buffer. Therefore, the instructions that do not belong to the calculated slice do not have anything to do with what is shown to the user. As an alternative to pixels buffer, system calls could be leveraged to define broader slicing criteria (Section IV-C), so that the profiler determines what instructions have any impact on the values communicated with I/O, including the network, display monitor, and audio device.

The profiling results show that only 45% of dynamically executed instructions on average contribute to the value of pixels in the process of rendering the Web pages in our benchmarks. We provide details of slicing percentage in important threads of the rendering process of the browser under test (Google Chromium). Moreover, by analyzing the the instructions which do not belong to the pixel-based slice (i.e., 55% of all instructions), we categorize potentially unnecessary computations and show that the most notable category is processing of JavaScript codes.

In the remaining sections of this paper, we first provide background on how Web browsers render Web pages and what the potential sources of unnecessary computations are. Next, in Section III, the design of the backward-slicing-based profiler is presented. We introduce the evaluation methodology in Section IV and describe how we leverage the profiler to identify unnecessary computations of different benchmarks. Then, we present and discuss the results in Section V. Finally, the paper is concluded in Section VII.

II. Background and Motivation

A. Rendering Pipeline of Web Browsers

For rendering a Web page, browsers follow a number of steps called the *rendering pipeline*. Figure 1 shows an overview of this pipeline, which is described below:

- First, the browser starts parsing an HTML file and generates a tree named the Document Object Model (DOM). This tree defines the hierarchical relationship between all the different elements available in the HTML file.
- Next, CSS files are parsed and a tree called CSS Object Model (CSSOM) is constructed. CSS files are complementary to the HTML file and define the exact style of the different elements in the HTML file.
- In the next step, the required JavaScript codes are executed which can arbitrarily modify or update the object model trees.

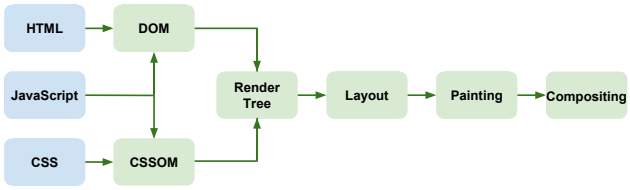


Figure 1: Rendering pipeline of a Web browser.

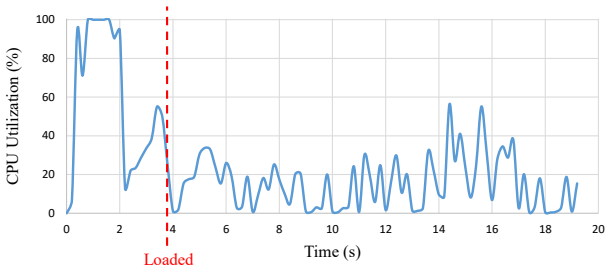


Figure 2: CPU utilization by the main thread of the tab process while browsing *amazon.com*.

- After running JavaScript codes, the browser merges the updated DOM and CSSOM and generates a new tree which then gets trimmed down to only contain objects that include visual context to the user. The resulting tree is called the Render Tree.
- Next, the exact position and size of different elements, which may be grouped in different layers, are computed in the layout stage. Then, the required graphical commands are generated in the paint stage, and according to the relative order of the layers computed in the compositing stage, the final view of the Web page is rendered in the user's display.

Note that the pipeline outlined above describes how a Web page is rendered during both load time and also the time when the page is modified based on user interactions (e.g., opening a menu) or dynamics of the page (e.g., an animation). However, the computations of load time are much more intensive because the whole page is rendered from the ground up, while once it is completely loaded, changes made to the page by user interactions or dynamics only affect a few elements of the page. To illustrate this behavior, Figure 2 shows the percentage of CPU utilization in a fairly short browsing session, where the *amazon.com* website is loaded, the user scrolls down and up a little bit, clicks to see the next two photos in a photo roll, and finally opens a menu. The utilization percentage corresponds to the main thread of the tab process, in which the most critical computations, such as calculation of styles and execution of JavaScript code, are performed. Note that compositing is done in a separate thread (More details about the architecture of the Chromium browser are provided in Section V).

Table I: Unused JavaScript and CSS code bytes.

Website		Amazon	Bing	Google Maps
Only Load	Unused bytes	955 KB	103 KB	1.9 MB
	Total bytes	1.6 MB	199 KB	3.9 MB
	Percentage	58%	52%	49%
Load and Browse	Unused bytes	882 KB	82.5 KB	2.0 MB
	Total bytes	1.6 MB	206 KB	4.6 MB
	Percentage	54%	40%	43%

B. Unnecessary Computations in Web Browsers

In the rendering pipeline of Web browsers, there may be unnecessary computations. We categorize them into three main groups:

Unused JavaScript and CSS codes: There are various JavaScript and CSS libraries that Web developers tend to use–such as jQuery [7], Bootstrap [2], and React [10]–in order to reduce development time. Not all these codes, when imported, are really used, meaning that processing them is a useless computation. Table I shows the percentage of unused JavaScript and CSS code bytes after loading three different websites–that is, Amazon, Bing, and Google Maps–and also after browsing them for 30 seconds in a typical way. As can be seen, about 40-60% of JavaScript and CSS codes are unused, and even by browsing the websites, not all these codes are used. Moreover, in the case of Bing and Google Maps, more code bytes are downloaded while browsing, which adds to the total bytes, and may add to the number of unused bytes, as compared to the load time.

Browser design pitfalls: Web browser designers have been constantly trying to improve the performance of Web browsers by leveraging complicated methods and algorithms. Although the improvement in the performance of Web browsers could be easily observed by comparing their earlier versions to their state-of-the-art ones, there are a number of optimizations, some of which are done speculatively, that have not been fully verified to work all the time or in the common case. For example, in the compositing algorithm of the Chrome browser [3], multiple elements of the page are grouped together as different layers, and to avoid repainting their contents, each layer has its own backing store/cache. However, this is expensive in terms of memory requirements; moreover, the computations and memory space related to the layers that are only rendered once and will not be required to be repainted (e.g., because they are always on top of other layers or they are always invisible) are wasted. The compositing algorithm of Chrome blindly accepts these overheads and potentially unnecessary computations. Other examples include multi-threaded rasterization, which may invalidate some pixel-based optimizations done at the early stages of the rendering pipeline [9], and the JavaScript JIT compiler deoptimizations, which are done because of wrong assumptions of the compiler about object types [37].

Imperceptible computations: A Web page consists of many layers, which may overlap each other, and elements, which may never be noticed or utilized by the users. For example, a layer that is overlapped by another layer may most likely remain invisible while the user interacts with the Web page. Similarly, a button element that is placed at the bottom of the page may never be clicked by the user. Therefore, the calculation of their styles and layouts, or compilation of the JavaScript code that corresponds to their event handlers (e.g., the code for handling the *onclick* event) is imperceptible to the user. Existence of Web analytics tools that could even track user clicks and scrolls enlightens the fact that not all the elements in the Web page have the same importance level.

C. Detection of Unnecessary Computations

A program slice contains instructions whose execution affects the values of a set of variables at a specific point in the program execution. The pair *(program point, set of variables)* is called *slicing criterion* [38]. Program slicing is typically done by starting from the *program point* given by the *slicing criterion* and going backwards toward the beginning of the program. Hence, this is called backward program slicing. Program slicing could be done either statically or dynamically. In static program slicing, no assumption is made on program inputs, while in the dynamic approach, slicing is done on the dynamic instruction trace of a sample execution. Static program slicing is less precise in that it has to make conservative assumptions on program inputs. Thus, our choice for the profiler is dynamic program slicing.

A profiler based on dynamic backward program slicing can theoretically identify all wasted computations mentioned in Section II-B. If the slicing criteria are defined in a way to include all the "necessary" variables at exact program points, execution of whatever instructions that are not part of the calculated slice is unnecessary. However, these necessary variables should be carefully specified, which may not be practical or even possible. If such criteria intuitively cover what the user cares about–that is, visual contents shown to them and page objects with which they interact–the computations related to processing unused JavaScript and CSS codes, layers that are invisible, and page elements that are not important to the user will be discovered.

In the next section, we describe our slicing-based profiler, and then in Section IV, we explain how slicing criteria are chosen to effectively identify unnecessary computations.

III. Profiler Design

The profiler implemented and used in this work is based on dynamic backward program slicing. Figure 3 shows an overview of the profiler design and how it works. The profiler performs dynamic backward program slicing on a trace of dynamically executed machine instructions. In other words, it does not do slicing at the C/C++ source code level; rather, it tracks back machine-level instructions from

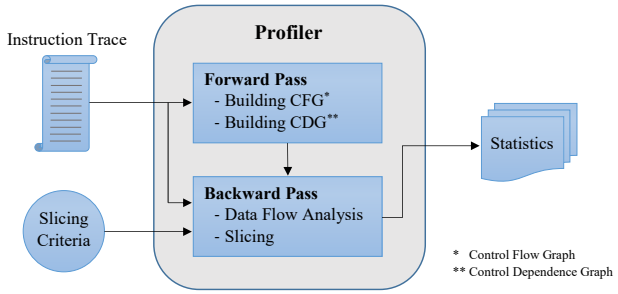


Figure 3: Profiler design overview.

the end of the instruction trace to the beginning and marks each instruction as being part of the slice or not based on the slicing criteria as it goes backwards. The slicing criteria essentially determine what the target variables (i.e., memory locations) are at what points in the instruction trace. The output of the profiler includes statistics about the calculated slice, such as distribution of instructions of the slice among all instructions at function-level or thread-level. Note that unlike other slicers that only focus on a specific aspect of a Web application, such as JavaScript [40], our profiler treats the browser as a whole program rendering a page.

Traditionally, program slicers perform slicing on a program dependence graph, which is a combination of the data dependence graph and control dependence graph [38]. In our profiler, we construct the control dependence graph in a forward pass, as displayed in Figure 3. However, we do not explicitly construct a data dependence graph. As will be explained in Section III-B, data dependencies are discovered through a liveness analysis meanwhile the profiler goes backwards and performs slicing. Since the input trace contains exact memory addresses accessed by the browser, the profiler does not suffer from the memory aliasing problem in capturing data dependencies.

In the rest of this section, we go through the details of the forward and backward passes. Then in Section IV, we describe how the slicing criteria should be chosen so that the unnecessary computations of a trace collected while a Web browser renders a Web page are effectively identified.

A. Forward Pass

In a single forward pass, the profiler first builds a Control Flow Graph (CFG) for each function/procedure from the trace of dynamically executed instructions. Boundaries of functions/procedures are identified through matching call and return instructions. Note that since the profiler works on machine-level instructions, it is necessary to build the CFGs from the trace of dynamic instructions in that the target(s) of indirect branches could not be found statically (i.e., from the instruction opcode). Also, all CFGs have their own specific *entry* and *exit* nodes.

In the next step, the Control Dependence Graph (CDG) of the instructions is built. CDG shows on what branches each instruction is dependent. For building the CDG, we first need to determine the postdominators of each instruction. In a CFG, a node n postdominates a node m if and only if every directed path from m to *exit* contains n. Algorithms for computing postdominators of each node in a CFG and subsequently, computing the CDG are not very complicated, and could be derived from basic compiler books and articles [19], [23]. Note that the calculated CDG could be stored in stable storage, so that it can be re-used multiple times in the backward pass for different slicing criteria.

B. Backward Pass

In the backward pass, data dependence analysis and slicing are done concurrently through liveness analysis. Conceptually, in our slicing method, there is a set of live variables, which is updated based on two distinct factors: slicing criteria and operation of instructions. As Figure 3 illustrates, slicing criteria–which are pairs of *(program point, set of variables)* (Section II-B)–are given to the backward pass analyzer of the profiler as input. When the profiler reaches to any *program point* specified in a slicing criterion, it puts the corresponding *set of variables* into the live set.

The second factor, based on which the live variables set may be updated, is operation of instructions, which also determines whether or not instructions should be part of the slice. If an instruction writes into a variable that is a member of the live variables set, that variable is taken out of the live variables set, and variables which are read by the instruction, if any, are put into the live variables set. Moreover, the instruction becomes part of the slice. As an example, if the slicer reaches the pseudo-instruction $c = a + b$, and c is a member of live variables set, it removes c from it, puts a and b into it, and finally puts this instruction into the slice.

Control dependencies also play an important role in putting instructions into the slice or not. When an instruction becomes part of the slice based on the described liveness analysis above, all branches on which this instruction is dependent should also be put into the slice. Therefore, these branches are put into a pending list, so that when the backward pass reaches a branch in the pending list, it is put into the slice. Moreover, the way branches update the live variables set differs from how regular instructions do so in the way described in the previous paragraph: when a branch must become part of the slice, its condition variable is put into the live variables set. For example, when the profiler reaches the pseudo-instruction $if(c)$ (c is the condition variable) which is in the pending branch list, c is put into the live variables set, and the branch is put into the slice and removed from the pending branch list.

In practice and at machine-level instructions, variables are, in fact, registers and memory locations. Therefore, in a single-threaded program, the live variables set actually consists of a live memory set and a live registers set. On

the other hand, Web applications are typically multi-threaded programs, and thus, it is required that our profiler also works for multi-threaded programs. The profiler assumes that even for a multi-threaded program, it is given a single instruction trace, which means that it requires that different threads are executed sequentially during the instruction trace collection phase. This makes the design of the profiler simpler because there is no need to handle synchronization between threads, and data dependence of instructions of different threads through shared memory can be easily identified by the liveness analysis described above. Finally, since the architectural context of the CPU changes when it switches the execution between threads, the profiler needs to keep a separate live registers set for each thread. Note that we should not have separate live memory sets for different threads because each thread has a distinct address space for local memory (i.e., heap and stack).

IV. Evaluation Methodology

In this section, we utilize the proposed profiler to identify unnecessary computations in rendering real websites. We implemented the profiler in C++ based on the descriptions in the previous section. Our test Web browser is Google Chromium, which is an open-source program [13]. For collecting instruction traces, we attach Intel's dynamic binary instrumentation tool, that is, Pin [6], to a specific tab of Chromium (each Chromium tab has its own separate process). Using a Pin tool written by us, we obtain the required information about the execution of instructions and store it in stable storage. In the rest of this section, we first explain the details of our Pin tool. Then, we describe the benchmarks and how slicing criteria are designated.

A. Dynamic Binary Instrumentation

Pin [6] is Intel's dynamic binary instrumentation tool, which can inspect and even manipulate dynamically executed instructions using only the program binary. The task of instrumentation and inspection/manipulation could be customized through writing Pin tools.

We wrote a Pin tool that collects static and dynamic information about the executed instructions. Static information includes the required data that could be extracted from the instruction opcodes, such as whether an instruction is a call, return, or direct/indirect conditional/unconditional branch, and which registers it accesses. Dynamic information includes data that are available at runtime, such as the addresses of memory locations accessed by an instruction, the ID of the thread where it is executed, and the system call number if the instruction is *syscall*.

System calls need special attention. Pin only instruments user-level code and does not inspect operating system instructions. System calls may change the value stored in registers and memory, thereby affecting the procedure of our liveness analysis. In order to solve this issue, we determined the record of all system calls that Chromium executes. We looked in the Linux kernel manual to understand how each of these system calls manipulate memory. For example, the syntax of *sendto* system call is as follows:

```
ssize_t sendto(int sockfd, const void *buf,
               size_t len, int flags,
               const struct sockaddr *dest_addr,
               socklen_t addrlen);
```

When our Pin tool reaches a *sendto* system call, it indicates in the trace file that memory locations pointed by *buf* and *dest_addr* are read accesses. How registers are manipulated by a system call is specified in a CPU's ABI (Application Binary Interface). Our profiler takes care of this issue based on the standard specified in the Intel's x86-64 (i.e., AMD64) ABI, which is the processor architecture used in our experiments.

B. Benchmarks

We use the Chromium browser, as was briefly mentioned earlier, to generate real-world benchmarks. We collected four instruction trace sets from different websites: Amazon in desktop view, Amazon in emulated mobile view, Google Maps, and Bing. We chose these three websites because their appearance and user interface totally differ from each other. Moreover, the desktop and mobile views of Amazon are considerably different. The instruction traces of the first three benchmarks include the load time of the corresponding websites (i.e., Amazon and Google Maps); that is, the trace is collected from entering the URL to when the Web page is completely loaded. However, the last benchmark, i.e., Bing, includes the instructions of loading the Web page and browsing it in a typical way. The browsing is composed of several user actions: opening and closing the top right menu, clicking on a button to roll the news pane in the bottom of the page, and typing a term in the search bar.

In Chromium, each tab is actually a separate process composed of multiple threads. Before starting to collect the instruction trace of a tab of Chromium, we set affinity of the corresponding process to one, so that all the threads of that process are sequentially executed on only one CPU core. This requirement, as explained in Section III-B, is imposed by our profiler. Next, we attach our Pin tool to the tab's process to start collecting the trace of instructions, and we enter the URL of a website. Benchmarks are generated using Chromium v58 that was run on an Ubuntu 14.04 desktop with 8 GB of RAM and an Intel Xeon E31230 CPU; note that Pin only supports Intel CPUs.

As will be explained later in this section, for the slicing criteria that we use, we need to know the address of pixels buffer and the points in the trace at which they contain values that are going to be put on the screen. In order to achieve this knowledge, we studied the source code of Chromium and found the point in the code (which is inside the *RasterBufferProvider::PlaybackToMemory* function) where

the final value of pixels (i.e., bitmaps) are written into a special buffer which corresponds to a tile of the screen (tiles are typically squares of 256×256 pixels). We put a unique instruction marker, that is, "*xchg* $\%r13w$, $\%r13w$", in a proper point in this function. We also modified the code of this function so that whenever Chromium executes it, the address of the tile buffer and its size are stored in an external file. This file and also the special instruction marker are, in fact, a set of slicing criteria provided to the profiler.

C. Choice of Slicing Criteria for Web Applications

As mentioned in Section II-B, in order for our profiler to effectively discover unnecessary computations of a Web application, slicing criteria should be carefully designated. Ideally, slicing criteria should contain all variables at exact program points that are somehow valuable and important to the user. Defining such criteria is a difficult task because relating user satisfaction in all possible executions to machine-level variables may not be practical or even possible. Therefore, we try to designate slicing criteria that closely match the ideal case. In this work, we use two types of slicing criteria: pixels buffer and system calls.

Pixels buffer. We define our first set of slicing criteria as the values of the pixels buffer that are shown to the user during rendering the page. The values of pixels of the display containing the Web page are actually the endpoint result of the application computations. Therefore, whatever that does not have any visible effect by no means–such as unused JavaScript and CSS codes, invisible layers, and page elements located at the very end of the page that are not shown on the first view of the Web page–will not be part of the calculated slice.

System calls. System calls are, in fact, means by which a process communicates with the outside world, including the network and display monitor. Therefore, we define our second set of slicing criteria as the values used by any system calls. Note that the slice computed by this set of slicing criteria must be inclusive of that of the pixel-based criteria, and the reason that we also use such criteria is to capture important computations to the user that do not have any visual effect, such as bank transactions through the network or audio playback.

Both types of slicing criteria described above are browser-independent. Particularly, in the case of pixels buffer, we only need to locate in the browser's source code where this buffer is filled with the final value of the pixels. In other words, how the values stored in the pixels buffer are calculated, which may differ from one browser to another, does not affect the way the profiler performs slicing.

For the benchmark related to a complete browsing session–that is, loading and browsing the Web page for a while–the instructions that do not belong to the calculated slice through either of the mentioned types of slicing criteria specify computations that were not necessary for rendering the page in that particular session. On the other hand, such instructions for the benchmarks that only contain loading a Web page denote either computations that are unnecessary (similar to the complete browsing session case), or computations that would be useful if the user started browsing the page, e.g., computations that are responsible for preparing the state of the application for the interactions of the user with the page which do not have any visible effect at load time (such as pre-compiling JavaScript code that would be fired as soon as the user starts interacting with the page). Our results, however, show that the latter item includes a very small percentage of instructions, and almost all the instructions that do not belong to the calculated slice in the benchmarks that only contain the load time could be treated in a similar way to the benchmark containing both loading and browsing the page.

V. Results and Discussion

In this section, we present the output results of our profiler regarding doing pixel-based slicing on the collected instruction traces from different websites. Our results show that slicing based on either pixels buffer or system calls leads to almost the same slice. Hence, only results of pixel-based slicing are presented and discussed.

A. Calculated Slice

Table II contains the statistics of the pixel-based slicing approach. The results show that the pixels slice is, on average, composed of **45%** of dynamically executed instructions in the four different benchmarks, which is an interestingly small percentage number. This implies that there is a good opportunity to identify useless computations in more than 50% of instructions. Note that in the Amazon benchmarks, the length of the trace in the mobile view (2.9 billion instructions) is so much smaller than that of the trace in the desktop view (6.2 billion instructions), which is because the first view of the Amazon Web page is much simpler in mobile displays as compared to desktop displays.

For the Bing benchmark, we also performed backward slicing starting from the time when the page was completely loaded back to the beginning time, which is composed of 1.7 billion instructions. The total slicing percentage for this experiment is 49.8%. On the other hand, when slicing is done starting from the end of the full trace, i.e., when the browsing session is complete, 50.6% of instructions that correspond to the load time are part of the calculated slice. This implies that browsing the Web page only makes about 1% more instructions of load time become useful.

Table II also includes statistics of three important thread types: main thread, compositor, and rasterizers. The main thread is mainly responsible for processing HTML, CSS, and JavaScript codes. The compositor thread handles the order of the layers containing the elements of the Web page and is also in charge of handling user inputs and animations. User

Table II: Slicing statistics of pixel-based approach for all instructions and important threads.

Threads	Amazon (desktop view): Load		Amazon (mobile view): Load		Google Maps: Load		Bing: Load + Browse	
	Pixels slice	Total instructions	Pixels slice	Total instructions	Pixels slice	Total instructions	Pixels slice	Total instructions
All	**46%**	6,217 M	**43%**	2,861 M	**47%**	4,238 M	**43%**	10,494 M
Main	52%	2,173 M	59%	764 M	61%	1,382 M	44%	3,499 M
Compositor	34%	1,711 M	35%	1,135 M	35%	1,698 M	34%	3,702 M
Rasterizer 1	55%	199 M	14%	76 M	78%	32 M	71%	617 M
Rasterizer 2	60%	66 M	13%	88 M	74%	29 M	52%	345 M
Rasterizer 3	54%	191 M	-	-	-	-	-	-

inputs that do not cause any major change to the rendered page, such as scrolling, are handled in the compositor thread, but for other inputs, such as a mouse click to open a menu, the compositor thread notifies the main thread to render the changes. Moreover, the compositor thread also notifies the main thread when a new animation frame must be rendered. Chromium might launch a different number of rasterizer threads for each website. These light-weight threads translate graphical objects (e.g., lines and circles) into pixels. In our benchmarks, Amazon with desktop view had three rasterizer threads, while other benchmarks had only two rasterizers.

The slicing percentage of the compositor thread is almost the same across all the benchmarks, while that of the main and rasterizer threads varies and is website-specific. This is reasonable because HTML, CSS, and JavaScript codes of different websites, which are processed by the main thread, are not the same, and what will finally be rasterized and displayed on the screen completely depends on the website content. On the other hand, the responsibilities of the compositor thread are not dependent on the details of the website content. Calculating the correct order of the layers and determining whether or not they are visible; handling user inputs and forwarding them to the main thread if necessary; and notifying the main thread to render a new animation frame are generic, website-independent tasks performed by the compositor thread.

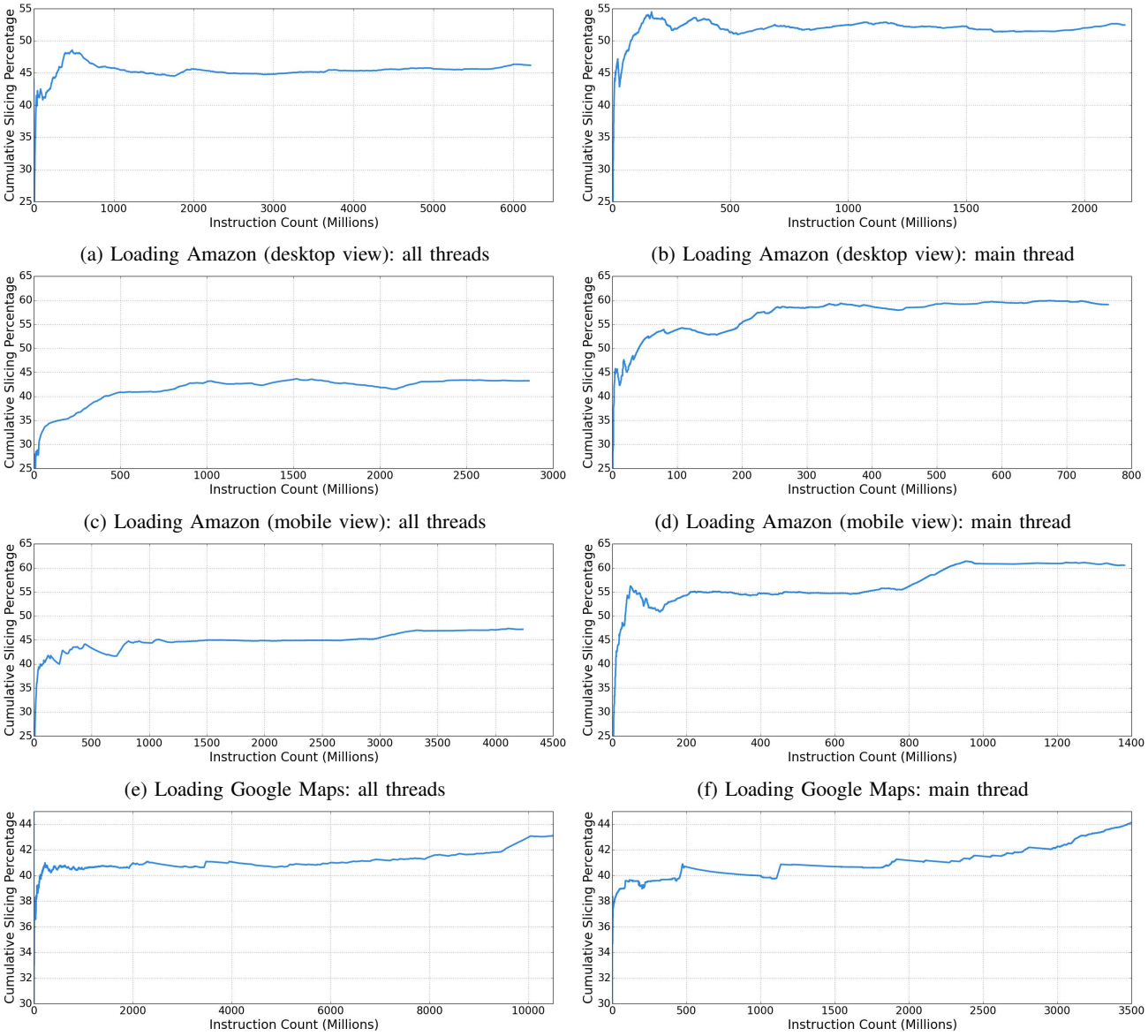
In the Amazon benchmark with mobile view, the slicing percentage of the rasterizer threads is very small. Note that for this benchmark, we emulated a mobile display using the Developers Tool of Chromium. The emulated display has a 360×640 resolution, which does not actually contain a large number of pixels. Therefore, these threads' effort to rasterize the content seems to be not quite useful as it is reflected on a few pixels.

The slicing percentage of the compositor thread in all the benchmarks is also small. As mentioned in Section II-B, in the compositing algorithm of Chrome/Chromium, a backing store/cache is specified to each layer, either when the layer is visible or not, so that if the order of layers changes and some layers become visible, the correct content is displayed quickly. While this idea may bring performance, it may also lead to useless computations in case of the backing stores whose contents are never used because some layers are fully or partially overlapped during the whole browsing session. The low slicing percentage of the compositor thread indicates that more smart compositing algorithms could provide both performance and energy efficiency.

Figure 4 shows how the slicing percentage changes in the backward pass for the pixel-based slicing criteria on different benchmarks. The x-axis in these charts shows the progress in the backward pass; therefore, the starting point on the x-axis corresponds to the time when the Web page is loaded or the browsing session is done, and the last point is related to the time when the Web page URL is entered. The y-axis shows the percentage of instructions of the slice for a specific point on the x-axis (aggregated from the starting point) in the instructions analyzed up to that point. The results are shown both for the instructions of all threads and also for the instructions of only the main thread. We can see that the changes in the overall slicing percentage of all threads in the backward pass is almost constant in large intervals. This implies that the distribution of instructions of the slices among all instructions is fairly even overall. However, the range of changes in the slicing percentage of the main thread is more in contrast to all threads. This means that computation regions that do or do not contribute to the pixel values are more conspicuous in the main thread as compared to other threads. It is also interesting to notice that for the main thread in the Bing benchmark (Figure 4h), there are some points where the slicing percentage suddenly increases (i.e., $x = 400$, $x = 1100$, and $x = 1800$), and then there is a gradual decrease in it. These points correspond to the user interactions that make the main thread render the imposed changes, such as rolling the news pane. Moreover, near the end of the chart (i.e., $x = 3000$), there is another considerable increase in the slicing percentage, which is related to loading the page. All in all, whenever rendering or re-rendering happens, the overall slicing percentage increases in that it leads to changes in the pixel values.

B. Categorization of Unnecessary Computations

Now that the slice of instructions that determine the value of pixels is calculated, we categorize unnecessary computations by analyzing the instructions that are not part of the calculated slice ($\sim$55% of all instructions). We closely examined the functions that each dynamically executed instruction belongs to using the symbol table stored in the



(a) Loading Amazon (desktop view): all threads

(b) Loading Amazon (desktop view): main thread

(c) Loading Amazon (mobile view): all threads

(d) Loading Amazon (mobile view): main thread

(e) Loading Google Maps: all threads

(f) Loading Google Maps: main thread

(g) Loading and browsing Bing: all threads

(h) Loading and browsing Bing: main thread

Figure 4: Changes of slicing percentage over the backward pass. $x = 0$ indicates the Web page is loaded or the browsing session is done, and the last point on the x-axis corresponds to entering the Web page URL.

application binary and used the namespace of the functions as the basis for categorization.

The categories of potentially unnecessary instructions by this namespace analysis are: JavaScript, Debugging, Inter-Process Communication (IPC), Multi-threading, Compositing, Graphics, CSS, and Other. Note that when compiling the Chromium source code, all debugging options were turned off, and the Debugging category reflects the default debugging mechanisms built in Chromium. IPC corresponds to the communication of the tab process with browser's main process. In Chromium, there is a single main process which manages the views of different tabs and other things such as browser extensions. Each process in Chromium is multi-threaded, and the Multi-threading category mainly consists of PThread code, which enables thread communication and synchronization. The Compositing category relates to the operations of the compositor thread, which is also the last stage shown in Figure 1. The Graphics category basically corresponds to the Paint stage of the rendering pipeline (Figure 1), and the CSS category is related to style and layout calculation in the rendering pipeline. The Other category mainly consists of event scheduling; note that all

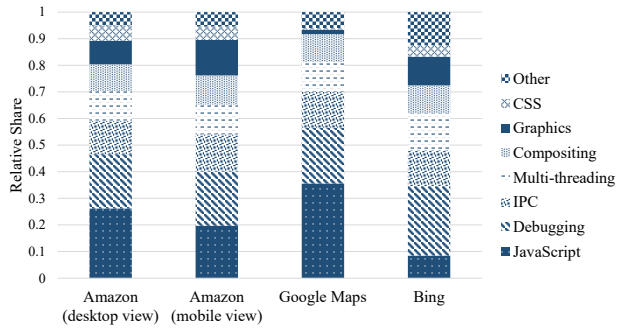


Figure 5: Categorization of potentially unnecessary computations and their distribution through analysis of instructions that do not belong to the pixel-based slice.

threads in Chromium are event-driven in nature, and event scheduling deals with managing an event queue, which holds events that should be executed.

Distribution of the categories of potentially unnecessary instructions through the namespace analysis is illustrated in Figure 5. Note that through this methodology, not all instructions could be categorized because not all functions have a specific namespace. The results shown in this figure include 74%, 59%, 53%, and 61% of the Amazon in desktop view, Amazon in mobile view, Google Maps, and Bing benchmarks respectively.

Figure 5 shows that most of the potentially unnecessary instructions belong to the first three categories, which are JavaScript, Debugging, and IPC. Presence of JavaScript in this list is not surprising. Also, it is reasonable that debugging codes are detected as unnecessary in that their execution has nothing to do with what is displayed on the screen. However, the IPC category needs more inspection because execution of instructions belonging to this category might have useful effect on the browser's main process; this is left as future work. It is interesting that in the Bing benchmark, which includes both loading and browsing the page, the JavaScript category has a smaller share as compared to other benchmarks, which only include loading the page. This implies that, generally, loading is the most intensive time in terms of processing JavaScript codes, not all of which are useful in a browsing session. Therefore, deferring processing of JavaScript codes to a time when they are really needed could provide better performance in Web applications. It is also worth mentioning that because of the noticeable presence of the Multi-threading category in Figure 5, and also because the share of the Other category, which mainly has to do with event scheduling, increases by browsing the page, assignment of tasks to different threads and scheduling mechanism of Chromium need reconsideration.

VI. Related Work

A. Workload Characterization of Web Applications

Prior work on characterization of Web applications mainly focused on JavaScript [32], [34], [35]. In contrast, in this work, we essentially characterize the whole JavaScript and rendering engines and determine computations that are useful for users. [32] and [34] characterize dynamic behavior of JavaScript workloads in terms of functions and objects, events and event handlers, and memory allocation. [32] concludes that JavaScript behavior of real Web applications and available benchmarks differ, and the benchmarks are not representative of real-world websites. [34] points out common misunderstandings of the behavior of JavaScript programs mainly caused by the available benchmarks. As a result, benchmarks inspired by real user actions have been developed [12], [33].

B. Performance Optimization of Web Applications

Many techniques have been proposed in prior work to improve performance of Web applications targeting various components of them. These techniques mainly enhance the JavaScript engine or improve the load time of Web pages.

JavaScript. Much prior work has focused on improving the JavaScript JIT compiler and execution engine. [18] enhances object type prediction of a JavaScript compiler by decoupling prototypes and method bindings from the object type. [25] uses server-side profiling to reduce deoptimizations done at client-side JavaScript engines. WebAssembly [24] is low-level, high-performance code compiled from C/C++ which could be utilized in Web applications through specific JavaScript APIs. Prior work also tried to bring parallelization to the JavaScript engine. [29] proposes offloading runtime checks of the JavaScript JIT compiler to a separate thread. [28] tries to parallelize loops in compute-intensive JavaScript applications.

Web page load time. The load time of Web pages has also received lots of attention in prior work due to its high impact on user experience. [27] proposes a coupled design of a server, which decomposes Web pages into sub pages on-the-fly, and a Web browser, that processes the sub pages in parallel. [26] leverages a machine learning model to predict future Web accesses of a user and prefetch the Web content. [30] decreases the load time of Web pages by caching and re-using JavaScript objects across browsing sessions. [20] and [39] dynamically reprioritize the content of a Web page to improve the load time of the Web page and sooner deliver resources that are critical to user experience.

C. Energy-efficient Mobile Web Applications

Energy efficiency of Web applications is a critical matter in mobile devices such as smartphones. Prior work mainly focused on frequency/voltage scaling of heterogeneous multiprocessors [31], [36], [41], [42], [44]. In [42], statistical models are achieved to estimate the time and

energy consumption of loading Web pages based on their characteristics–such as, number of HTML tags, number of CSS rules, and content size. Based on these models, proper frequency/voltage of Arm big.LITTLE cores [1] are found after parsing the Web page. [31] characterizes the energy consumed in different processes and threads of a Web browser and proposes several power management policies on heterogeneous multiprocessor platforms. [41] and [44] propose energy-efficient schedulers of a heterogeneous mobile architecture based on the QoS requirements of users, which is, respectively, determined by automatic reasoning based on intensity and latency, and two novel CSS language extensions provided for Web developers.

D. Architectural Support for Web Applications

Due to widespread use of Web applications, prior work also proposed specialized hardware and architectures for them. [43] identifies fine-grained parallelism in applying styles to HTML elements and proposes a specialized hardware unit for it. It also proposes a specific cache for the document object model tree since its content is heavily reused while rendering a Web page. In [21], a specialized prefetcher is designed that takes advantage of long latency cache misses to bring to cache data and instructions required for future events that are in the event queue. [22] accelerates JavaScript object accesses through a hardware table similar to a branch target buffer.

VII. Conclusion

The performance of today's Web applications is often unsatisfactory to users, and in this work, we argued one of the reasons for it is that there are unnecessary computations occurring in Web applications which could be avoided or scheduled in a better way. We designed a profiler that effectively identifies computations that are important to the user. To the best of our knowledge, this is the first work that quantitatively characterizes unnecessary computations of Web applications. The profiler detects instructions contributing to what is shown to the user on the device display during rendering a Web page. We showed that only 45% of dynamically executed instructions in the rendering process of the browser under test are useful for calculating the value of the pixels displayed to the user on average. By analyzing the rest of the instructions, we revealed inefficiencies of the Web browser (e.g., the compositing algorithm) and provided a categorization of computations that are either completely wasted or could be deferred to a more appropriate time (e.g., compiling a piece of JavaScript code when it is really needed), thereby providing opportunities for higher performance or reduced energy consumption.

Acknowledgment

The authors would like to thank anonymous reviewers for their comments and feedback. We also appreciate Salar Latifi's assistance during this work. This research was supported by the National Science Foundation grant SHF-1527301.

References

[1] "Arm big.LITTLE technology," https://developer.arm.com/technologies/big-little.

[2] "Bootstrap," http://getbootstrap.com/.

[3] "Compositing in Blink/Webcore," https://tinyurl.com/yd5nwmy3.

[4] "Firefox's Hardware Acceleration," https://support.mozilla.org/en-US/kb/performance-settings.

[5] "GPU Accelerated Compositing in Chrome," https://tinyurl.com/no64sem.

[6] "Intel's Pin," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[7] "jQuery," https://jquery.com/.

[8] "Measure Performance with the RAIL Model," https://developers.google.com/web/fundamentals/performance/rail.

[9] "Multi-threaded Rasterization in Chrome," https://tinyurl.com/yaksfwz8.

[10] "React," https://reactjs.org/.

[11] "Speculative Parsing," https://tinyurl.com/yxta9zyu.

[12] "Speedometer 2.0," https://browserbench.org/Speedometer2.0/.

[13] "The Chromium Web Browser," https://www.chromium.org/.

[14] "The need for mobile speed (DoubleClick by Google)," https://bit.ly/2eqaC8z.

[15] "Using Request Idle Callback," https://tinyurl.com/ybyseoo5.

[16] "Vue.js: The Progressive JavaScript Framework," https://vuejs.org/.

[17] "Webpack: Build System and Module Bundler," https://webpack.js.org/.

[18] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving javascript performance by deconstructing the type system," in *PLDI*, 2014, pp. 496–507.

[19] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[20] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices," in *USENIX NSDI*, 2015, pp. 439–453.

[21] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating asynchronous programs through event sneak peek," in *ISCA*, 2015, pp. 642–654.

[22] J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas, "Shortcut: Architectural support for fast object access in scripting languages," in *ISCA*, 2017, pp. 494–506.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[24] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *PLDI*, 2017, pp. 185–200.

[25] M. N. Kedlaya, B. Robatmili, and B. Hardekopf, "Server-side type profiling for optimizing client-side javascript engines," in *DLS*, 2015, pp. 140–153.

[26] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "Pocketweb: Instant web browsing for mobile devices," in *ASPLOS*, 2012, pp. 1–12.

[27] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos, "A case for parallelizing web pages," in *USENIX HotPar*, 2012.

[28] M. Mehrara, P. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *HPCA*, Feb 2011, pp. 87–98.

[29] M. Mehrara and S. Mahlke, "Dynamically accelerating client-side web applications through decoupled execution," in *CGO*, 2011, pp. 74–84.

[30] J. Oh and S. Moon, "Snapshot-based loading-time acceleration for web applications," in *CGO*, Feb 2015, pp. 179–189.

[31] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford, "Web browser workload characterization for power management on hmp platforms," in *CODES*, 2016, pp. 26:1–26:10.

[32] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *USENIX WebApps*, 2010.

[33] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of javascript benchmarks," in *OOPSLA*, 2011, pp. 677–694.

[34] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *PLDI*, 2010, pp. 1–12.

[35] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *ESEC/FSE*, 2013, pp. 488–498.

[36] D. Shingari, A. Arunkumar, B. Gaudette, S. Vrudhula, and C. Wu, "Dora: Optimizing smartphone energy efficiency and web browser performance under interference," in *ISPASS*, April 2018, pp. 64–75.

[37] G. Southern and J. Renau, "Overhead of deoptimization checks in the v8 javascript engine," in *IISWC*, Sept 2016, pp. 1–10.

[38] F. Tip, "A survey of program slicing techniques." Amsterdam, The Netherlands, Tech. Rep., 1994.

[39] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with shandian," in *USENIX NSDI*, 2016, pp. 109–122.

[40] J. Ye, C. Zhang, L. Ma, H. Yu, and J. Zhao, "Efficient and precise dynamic slicing for client-side javascript programs," in *SANER*, March 2016, pp. 449–459.

[41] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *HPCA*, Feb 2015, pp. 137–149.

[42] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *HPCA*, Feb 2013, pp. 13–24.

[43] ——, "Webcore: Architectural support for mobileweb browsing," in *ISCA*, 2014, pp. 541–552.

[44] ——, "Greenweb: Language extensions for energy-efficient mobile web computing," in *PLDI*, 2016.