Sprinter: Speeding Up High-Fidelity Crawling of the Modern Web

Ayush Goel[1] Jingyuan Zhu[1] Ravi Netravali[2] Harsha V. Madhyastha[3]
[1]*University of Michigan* [2]*Princeton University* [3]*University of Southern California*

Abstract—**Crawling the web at scale forms the basis of many important systems: web search engines, smart assistants, generative AI, web archives, and so on. Yet, the research community has paid little attention to this workload in the last decade. In this paper, we highlight the need to revisit the notion that web crawling is a solved problem. Specifically, to discover and fetch all page resources dependent on JavaScript and modern web APIs, crawlers today have to employ compute-intensive web browsers. This significantly inflates the scale of the infrastructure necessary to crawl pages at high throughput.**

To make web crawling more efficient without any loss of fidelity, we present Sprinter, which combines browser-based and browserless crawling to get the best of both. The key to Sprinter's design is our observation that crawling workloads typically include many pages per site and, unlike in traditional user-facing page loads, there is significant potential to reuse client-side computations across pages. Taking advantage of this property, Sprinter crawls a small, carefully chosen, subset of pages on each site using a browser, and then efficiently identifies and exploits opportunities to reuse the browser's computations on other pages. Sprinter was able to crawl a corpus of 50,000 pages 5x faster than browser-based crawling, while still closely matching a browser in the set of resources fetched.

1 Introduction

To make the most of the enormous trove of information available on the web, all of us today rely upon a range of efforts. Web search engines help users find pages relevant to their needs. Data from the web serves as input to smart assistants such as Siri and Alexa, and is used to train generative AI models that can answer our questions. Web archives store repeated snapshots of web pages to document changes over time and to preserve the content of deleted pages. Researchers continually study the web to help improve its performance and security.

A key enabler for all of the above is a capability that we take for granted today: the ability to crawl the web at scale. Web crawlers have traditionally crawled a page by first downloading the page's HTML, and then recursively fetching all embedded links to images, CSS stylesheets, scripts, etc. If one deploys many such so called static crawlers [32, 28] across a fleet of machines, the rate of crawling is limited by the network bandwidth of each machine.

Given that web crawlers have existed for over three decades, why revisit this topic now? Because, static crawlers no longer suffice. On today's web, the URLs of many of the resources on a page are determined at runtime, rather than being statically embedded in the page's source. To discover and fetch such resources, modern "dynamic" crawlers [5, 2, 4] leverage web browsers such as Chrome, Firefox, or Edge. However, due to the compute overheads associated with JavaScript (JS) execution and with browsers in general, the rate at which one can crawl pages drops by an order of magnitude relative to static crawling (Figure 1). Consequently, dynamic crawlers need to be deployed across a much larger number of servers in order to sustain the same crawling throughput as that feasible with static crawlers.

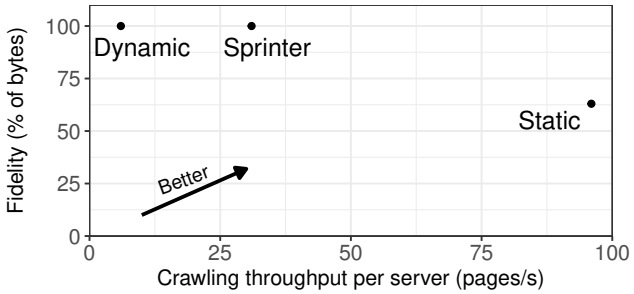


Figure 1: **Tradeoff between fidelity and performance with different crawlers.**

Thus, anyone seeking to crawl the web today has to make do either with the poor performance of dynamic crawlers or the incompleteness of static crawlers. Unfortunately, there is no easy fix. One could try to augment a static crawler with a lightweight JavaScript execution engine, but keeping up with constantly evolving web APIs is a challenge best left to the developers of widely used browsers. On the other hand, proposals that attempt to mitigate the impact of client-side web computations on user-perceived web performance have little utility in the context of crawling. For example, overlapping the browser's computations with its network activity [41, 24, 56] or parallelizing the browser's execution of JavaScripts [40] can reduce page load times, but crawling throughput remains unchanged since the total amount of client-side computation is the same.

We address this undesirable status quo with Sprinter, a new crawler which judiciously combines browser-based and browserless crawling. Sprinter crawls pages at a much faster rate than dynamic crawlers while matching them in the resources fetched. Our main observation is that large-scale web crawling workloads typically include many pages from each site and there is significant potential to reuse client-side computations across pages (§3.1).To exploit this property, our design of Sprinter is based on three key principles.

First, when Sprinter crawls a page using a browser, it strives to minimize the amount of JS code executed. For every script file on a page, Sprinter attempts to reuse the

browser's execution of that file on a previously crawled page. In user-facing page loads, execution of the same file is seldom exactly identical across multiple pages. In contrast, Sprinter can reuse JS execution at such a coarse granularity because it can skip executing a JS file as long as the URLs of the resources that file would fetch match those fetched during a prior execution of that file.

Second, even if none of the JS files on a page are executed, crawling the page with a browser imposes significant compute overhead. Therefore, on any site, Sprinter crawls the vast majority of pages on the site without a browser. To realize browserless crawling that does not sacrifice fidelity, we implement a lightweight page instrumentation framework that tracks the web APIs used on any page without support for executing these APIs. When it crawls a page without a browser, Sprinter uses this instrumentation to identify whether it can safely reuse JS executions from pages that it previously crawled with a browser.

Lastly, to maximize the fraction of pages that can be crawled without a browser, Sprinter crawls the pages on a site in a carefully chosen order. For any given site, Sprinter efficiently identifies a subset of pages such that most of the scripts seen on other pages are fetched as part of this subset. Sprinter crawls these pages first using a browser and captures the effects of JS executions. Most of the remaining pages can then be crawled without a browser, since Sprinter can identify all resources to be fetched on those pages without executing any JS code or web APIs.

We used Sprinter to crawl a corpus of 50,000 pages spread across a diverse collection of 100 sites. It offered a 5x speedup in crawling throughput compared to existing dynamic crawlers. When we recrawled the same corpus a week later, the rate at which Sprinter crawls pages improved by a further 78%. Importantly, Sprinter preserves almost all resource fetches issued by a browser-based crawler, and it is compatible with legacy web servers. Sprinter's source code is available at https://github.com/goelayu/Sprinter.

2 Background and Motivation

We begin by describing common web crawling workloads and quantifying the limitations of existing strategies for supporting these workloads.

2.1 Target workloads

Web crawlers take as input a seed list of URLs to pages that need to be crawled. The input configuration to the crawler can specify a range of options such as timeout per page, retry policy, politeness constraints (i.e., time gap between crawls of pages on the same site), and whether other pages discovered while crawling the seed list should also be recursively crawled. Some crawlers provide the option of saving page screenshots [2] and triggering user interactions (e.g., scrolling or clicking) on rendered pages [4]. In this work, we focus on supporting the common need for crawlers to save the content of resources that are fetched on every page that is crawled. To not make any assumptions about what the crawls will be used for, we aim to fetch and save all page resources requested by a browser such as Chrome, rather than a subset that may suffice for a particular use case.

We focus on supporting workloads where pages are crawled from a large number of sites. This is the case in any large-scale system that relies on web crawls, e.g., to support web search, ChatGPT, and Siri, their providers aim to crawl the entire web. Even in more focused crawls, it is common to crawl many sites and many pages in each site. For example, after every presidential term in the US, the Internet Archive captures a snapshot of 1.3 million government websites, crawling roughly 700 pages on average per site [14]. Similarly, research studies attempting to understand the web's security vulnerabilities [44] have crawled roughly 2500 pages per site. When pages are crawled from a single site (e.g., a research study of pages on Facebook), the rate at which pages can be crawled is constrained by the rate limits imposed by the site being crawled.

2.2 Shortcomings of static crawling

As mentioned earlier, web crawling has traditionally relied on static crawlers, which identify all the resources to fetch on every page by extracting links embedded in the page's source. To demonstrate and quantify why static crawlers are now insufficient, we compile $Corpus_{10k}$, a collection of 10,000 pages comprising 100 randomly sampled pages from each of 100 sites: roughly 33 sites chosen at random from three ranges – [1, 1000], [1000, 100k], and [100k, 1m] – from Alexa's site rankings. This corpus spans a diverse collection of sites and is representative of real-world crawls in that it includes a large number of pages per site crawled.

On a server which has a 16-core 2.1 GHz Intel Xeon CPU, a 1 Gbps network connection, and a 500 GB SSD disk, we crawl every page in $Corpus_{10k}$ using a custom crawler which loads every page in Google Chrome but also fetches all URLs, both absolute and relative, that are embedded in text-based resources (i.e., HTML, CSS, and JS). We record all responses using a web record and replay tool [9]. We then separately crawl every page from our recorded copy once using Chrome and once using our custom static crawler (which mimics wget2 [12], a state-of-the-art static crawler), with network caching enabled in both cases, i.e., across all pages, each unique resource was only fetched once. Comparing the two types of crawlers in this manner eliminates any differences that might arise due to server-side nondeterminism [36].

First, the *"Dynamic - Static"* line in Figure 2 shows that a static crawler fails to fetch 32% of bytes on the median page. This is because, on a modern web page, which resources are served to a client are often determined only when the client executes the scripts included on the page. Since a static crawler can identify the URLs of a page's resources

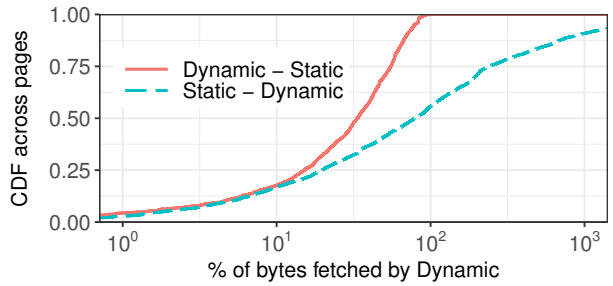


Figure 2: **Compared to a dynamic (i.e., Chrome-based) crawler, a static crawler both fails to fetch some resources and fetches many additional resources. Distribution shown is over 10,000 pages spread across 100 sites. Note logscale on x-axis.**

9297.js

```
var EA = fetch("crazyegg.com/usnews.com.json")
// json contents:  {
      script_url: "crazyegg.com/commonscripts/759.js"
}
```

utag.js

```
const n = document.createElement("script");
n.src = EA.script_url;
const r = document.getElementsByTagName("script")[0];
r.parentNode.insertBefore(n, r);
```

Figure 3: **Snippet of JS code from www.usnews.com. The browser first fetches a JSON file, and then requests a JS file referenced inside the JSON.**

only by parsing the source code of the page, it is blind to such resource fetches. Figure 3 shows an example.

Second, Figure 2's *"Static - Dynamic"* line shows that, on the median page, the static crawler fetches 93% more bytes than fetched by Chrome; on some pages, this overhead is as high as 200x. These extra resource fetches arise because, within a single page, web developers often embed resources that are applicable across a large number of client device types, expecting the client browser to download the resources applicable to it. Examples include multiple resolutions of the same image, or different font files for the same HTML text. To enable the client to pick the appropriate version of any particular resource, modern pages either use media queries [17] or CSS selectors [11]; see Figure 4 for examples. A static crawler is unable to evaluate media queries and does not know which CSS selectors are dynamically applied during JavaScript execution. Therefore, it offers no control on whether to fetch only resources applicable to the machine used for crawling a page or to fetch every resource that might be requested in any load of the page.

2.3 Compute overheads of browser-based crawling

Given the shortcomings of static crawlers, state-of-the-art web browsers are often employed to crawl pages [2, 5, 4]. We observe that Chrome is the most widely used in browser-based crawling frameworks because of its better support for web APIs [18] and for automation capabilities [8]. For this

index.html

```
<picture>
    <source srcset="ct.img/600x338" media="(min-width:768px)">
    <source srcset="ct.img/400x225" media="(min-width:0px)">
</picture>
```

style.css

```
.icon-calendar
      {font-family: {src: url("fonts/icomoon.woff")}}
```

widget.js

```
if (body.firstChild.hasAttr("data-widget")){
      var inode = document.createElement("i");
      inode.class = "icon-calendar";
      body.firstChild.insertBefore(inode)
}
```

Figure 4: **Code snippet from www.chicagotribune.com showing the two causes for a static crawler's extra resource fetches. (a) It will fetch both versions of the `ct.img` image, irrespective of the width of the client device's display. (b) It will fetch the font file `fonts/icomoon.woff`, whether or not the CSS selector `.icon-calendar` is used in the rest of the page. The CSS selector is only added if the HTML code contains a `data-widget` attribute.**

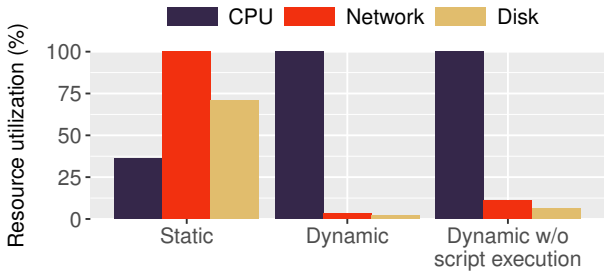


Figure 5: **A comparison of average CPU, network, and disk utilization by static and dynamic crawlers.**

reason, in the rest of this paper, we refer to Chrome[1] when discussing overheads of browser-based crawling.

We observe that the average number of pages that we could crawl per second with Chrome was only 12% of that achievable with the static crawler. The cause for this significant drop in crawling throughput is shown in Figure 5, which plots the average utilization of CPU, network, and disk with either crawler. Unlike the static crawler, which was limited by network bandwidth, the dynamic crawler ended up saturating all CPUs. If we were to use a 10 Gbps network, more than 5000 CPU cores would be necessary for the dynamic crawler to fully utilize the network, which is infeasible to accommodate on a single server.

We break down the reasons behind Chrome's high CPU usage using data from Chrome's in-built profiler [7]. We find three primary contributors: 1) the JavaScript engine, which is responsible for parsing and interpreting JS code, 2) inside the rendering engine, computation of the layout tree which

[1]We use Chrome in a headless mode as it is known to be more compute efficient [49, 21].

specifies on-screen positions for page content, and 3) time spent inside Chrome's internal code, into which the profiler has no visibility. Together, these three sources of computation account for 96% of the compute delays on the median page, with JavaScript execution alone accounting for about half. Given the complex inter-dependencies between these three tasks, none of them can be simply eliminated to reduce Chrome's computation overheads. For example, JavaScript execution queries layout information when scripts inspect the position of elements on the screen.

2.4 Minimizing browser's computation delays

The observation that the amount of client-side computation needed to load a web page has increased in recent times is not new. A large body of prior work [57, 42, 40, 41] has focused on addressing the impact of this overhead on user-perceived web performance. However, those solutions have little utility in the context of web crawling for two reasons.

First, many proposals for reducing the impact of client-side computation on page load times aim to either increase the overlap between the browser's use of the client CPU and network [41, 46] or parallelize the browser's execution of scripts on a page [40]. Such solutions can reduce the end-to-end latency of individual crawls, but the total amount of computation that the crawler needs to perform, and thus the crawling throughput, will remain unchanged.

Second, others [42, 57, 1] rely on server-/proxy-side support to ship processed versions of pages so as to minimize the amount of JavaScript that clients need to execute. Notwithstanding the fact that such solutions are not usable until they are adopted by millions of domains, we estimate their best case utility by crawling pages in $Corpus_{10k}$ with script execution in Chrome disabled. A comparison of *"Dynamic"* and *"Dynamic w/o script execution"* in Figure 5 shows that the latter marginally reduces the gap between CPU and network utilization. However, client-side computation remains a significant bottleneck, thereby limiting crawling throughput to still be only 17 pages per second.

Alternatively, one could attempt to build a lightweight browser from scratch which only supports crawling, but does not enable users to visit web pages, i.e., has no graphical interface, does not support user interactions, etc. However, significant engineering effort would be required to constantly keep up with updates in HTML, CSS, and JavaScript APIs. For example, when we load the landing pages of the top 1000 Alexa sites using a version of Chrome from five years ago (v65), it fails to fetch 16% of the resources fetched by the most recent version of Chrome (v114). This is because certain JavaScript APIs that are commonly used today were not supported by Chrome v65, e.g., support for optional chaining [15] was only added in v80. It would be best for web crawlers to rely on widely used browsers which are well-maintained, instead of having to replicate the effort in a lightweight browser dedicated to crawling.

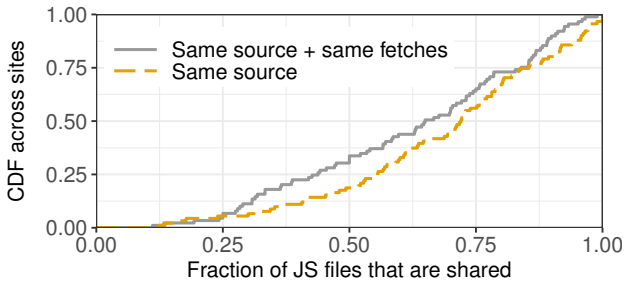


Figure 6: **For the sites in $Corpus_{10k}$, most JavaScript files appear on multiple pages and a script typically fetches the same resources on all the pages which include that script.**

3 Overview

The takeaway from the previous section is that, today, operators of web crawlers are stuck with having to choose between two less than ideal options: use static crawlers and miss out on some resources, or make do with the poor performance of dynamic browser-based crawlers. We seek to resolve this quandary by enabling high-fidelity crawling at high throughput. We do so while respecting two constraints. First, we make sure to crawl all the resources on a page that a browser would fetch, but make it configurable whether to crawl only the resources relevant to the machine on which the crawler is executed. Second, to make our crawler compatible with the legacy web, we require no changes to web pages and the servers that host them.

3.1 Observations and approach

The high-level observation that guides our approach is that, on any site, there typically is significant overlap across pages both in the JavaScript code that they include and JavaScript-initiated fetches when a browser loads them. Figure 6 demonstrates this property on the pages in $Corpus_{10k}$.

First, for every site, out of all the unique JS files seen on at least one of the 100 pages on that site, we compute the fraction which are included in multiple pages; here, we consider the combination of a file's URL and a hash of its source code to be a unique identifier for a file. The *"Same source"* line plots the distribution of this fraction across the 100 sites in our corpus. For the median site, 72% of JS files were shared across multiple pages.

Next, we examine the likelihood that a JS file fetches the same set of resource URLs when it is executed on different pages. For this, we consider a script file's execution uniquely by the file's URL, the hash of its source, and the set of URLs it fetches. When we consider only those executions which result in at least one fetch, the *"Same source + same fetches"* line in Figure 6 shows that, on the median site, 65% of unique file executions – at least with respect to resource fetches – are repeated across multiple pages.

The takeaway from these observations, coupled with the property that web crawling workloads typically crawl a large number of pages per site (§2.1), is that there exists signifi-

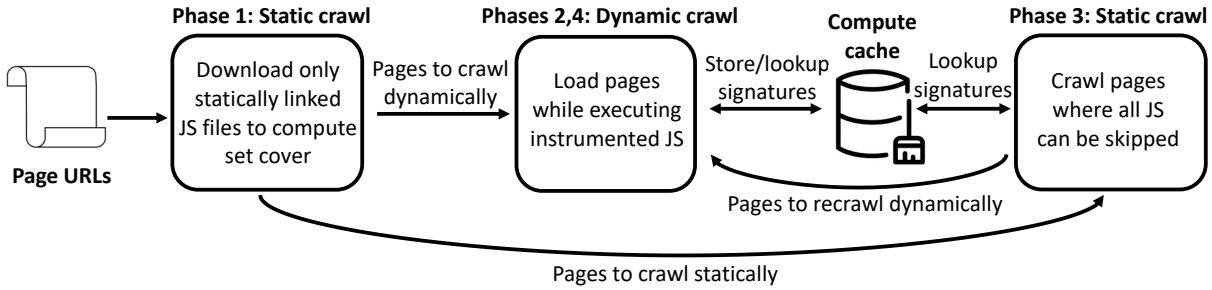


Figure 7: **Sprinter crawls the pages on any site in four phases which alternate between browserless and browser-based crawling.**

cant redundancy in a dynamic crawler's execution of JS files. When the browser used by the crawler executes a JS file that it had previously executed on a different page on the same site, the numbers from Figure 6 indicate that the same set of resources are often requested as on the previous page. The browser's network cache will ensure that it does not have to waste network bandwidth in re-downloading those resources. But, the browser will still execute every JS file in its entirety just to identify these resources.

To improve crawling performance by reducing the crawler's computations, our approach aims to first eliminate redundant execution of JS files. Specifically, whenever our crawler, Sprinter, crawls any page, it skips executing a JS file if a) it has already executed that file while crawling a different page on the same site, and b) it identifies that, if executed, the file will fetch the same resources as it did on the previously crawled page. However, as observed earlier (§2.4), a browser imposes high compute overhead even when it is used to load pages with execution of scripts disabled. Therefore, second, on pages where it can reuse the executions of *all* JS files, Sprinter does not even employ a browser to crawl those pages. Put together, Sprinter uses a browser to crawl only a small subset of pages in each site and minimizes the browser's execution of JavaScripts.

3.2 Challenges

Realizing the above approach requires us to answer the following three questions.

- Whenever a script appears on multiple pages, it is not guaranteed to initiate the same resource fetches on all pages; in our corpus, 48% of repeated scripts had at least one execution where they fetched a different set of URLs than what they fetched in their first execution. Prior to executing a script, how can Sprinter efficiently identify that the script's execution will match a prior execution, and it is safe to skip executing it?
- Classic memoization involves storing the results of execution and using them to skip future executions of the same code in the same runtime context. In contrast, when Sprinter crawls a page without a browser, how can it reuse the browser's prior computations on other pages? Mimicking the entire browser runtime will significantly increase complexity and degrade performance.
- Finally, on each site, which subset of pages should Sprinter crawl using a browser? To minimize Sprinter's compute overheads, it is key that the subset be small. However, for Sprinter to crawl all the remaining pages on the site without a browser, we must ensure that the script executions on the pages crawled using a browser suffice to skip executing all the JS files on the remaining pages.

4 Design

As shown in Figure 7, Sprinter crawls a corpus of pages from any particular website in four phases. In the first phase, Sprinter identifies the subset of pages that need to be crawled with a browser. It crawls those pages in the second phase while skipping JS executions whenever feasible. Next, Sprinter crawls the remaining pages on the site using its augmented static crawler. Finally, it recrawls some of the pages from the third phase using a browser. We present our design of Sprinter by first describing its operation in phases 2 (§4.1) and 3 (§4.2), and lastly, phases 1 and 4 (§4.3).

4.1 Memoizing JavaScript execution

Sprinter maintains a *compute cache* in order to take advantage of the opportunities to reuse JS executions across the pages on a site. On any page that Sprinter crawls with a browser, prior to executing JS on the page, the browser looks up the compute cache to determine whether the execution can be skipped. Upon a cache miss, the browser executes the JS code and logs a summary of its execution in the compute cache, for use on other pages.

Execution signatures to enable reuse. When JS code runs within a browser, it can read from or write to the JavaScript heap and HTML DOM object. It can also read the return values from various web APIs. Therefore, to enable reuse of JS executions without violating correctness, we associate the execution of every block of JS code with a *signature* which includes the values at the start of executing that block of code for all state from the heap or DOM that is read within that block. When the browser executes any block of JS code, its execution is guaranteed to result in the same externally visible effects (i.e., writes to the DOM and heap, and URL fetches) as a prior execution which had the same signature, if the block does not invoke any non-deterministic APIs (e.g., `Date`, `Random` or `Performance`). Figure 8 shows an example block of code and the corresponding signature.

vendor.js

```
var gKey = window.grumi.key; // “bfd2-4adc”
fetch(`https://www.geoedge.com/${gKey}/grumi-ip`)
var NYTD = {PageViewID:‘mubjhislka7867’};
window.NYTD = NYTD;
```

$\text{Signature}_{\text{vendor.js}}$

```
{
      Reads: [“window.grumi.key”,”bfd2-4adc”],
      Writes: [“window.NYTD”,”{PageViewID:‘mubjhislka7867’}”]
      Fetches: [“https://www.geoedge.com/bfd2-4adc/grumi-ip“]
}
```

Figure 8: **JavaScript code from www.nytimes.com which reads a global variable using the *window* object and, based on the property read, fetches a URL. It also writes to the window object. Signature for this includes the global state read and written (both the keys and the values) and the fetches initiated.**

However, to construct code signatures, modern browsers provide no APIs to extract the necessary runtime information about JavaScript execution. To remedy this, Sprinter uses a custom JS instrumentation framework, similar to the ones used in prior work [31, 40, 42]. This instrumentation framework runs inside a man-in-the-middle (MITM) proxy which sits in front of the browser. For every new JS file requested by the browser, the proxy statically analyzes the code in the file and rewrites it by injecting code that tracks the state and APIs that are accessed when the file is executed. Sprinter's instrumentation tracks variables on the heap which are in either 1) the global scope, which is accessible using the `window` object, or 2) the closure scope, which is created within a function but persists after the function's execution if there exists a nested function declared in the same enclosed scope. For the DOM object, Sprinter tracks all APIs that can read from (e.g., `getElementById`) or write to (e.g., `appendChild`) the DOM.

Once the browser finishes loading a page, Sprinter's injected JS code compiles signatures for the scripts on the page and stores them in the compute cache which is co-located with the proxy. These signatures include both the above-mentioned information needed to identify the opportunity for reuse, and the writes to the heap and DOM that need to be executed when the corresponding code is skipped, along with any fetches initiated; see Figure 8. When a previously cached JS file is fetched in future page loads, the proxy embeds stored signatures for the code in this file directly into the file. When processing each JS file, the browser uses the embedded signatures to determine if any code within the file can be skipped.

Granularity of JS execution reuse. Given our results from §3.1, it is natural to try and reuse the browser's JS executions at the granularity of entire files, i.e., prior to processing any script file, the browser uses cached signatures for that file to determine whether to skip all the code in the file or execute all of it. However, as shown in the *"Full signature"* line in Figure 9, the cache hit rate is pretty poor. On the median site in $Corpus_{10k}$, only 40% of JS file executions can be skipped.

Figure 9: **Cache hit rate for JavaScript files that initiate fetches for other URLs.**

To improve the hit rate, our key insight is that, unlike in user-facing page loads, we do not need to restrict Sprinter's skipping of a JS file's execution only when it is guaranteed to execute in a manner *exactly* identical to a previous execution of the same file. Rather, as long as we can guarantee that the code will fetch the same resource URLs, we can skip it. A crawler does not need to preserve other aspects of JavaScript execution, such as visual changes by modifying the DOM or functional changes by adding event handlers that allow users to interact with the page.

This observation enables us to trim file signatures and only include state that influences resource fetches. To identify this state, we turn to dynamic taint tracking [47]. Our instrumentation of any JS file marks all statements that initiate URL fetches (such as `XMLHTTPRequest.send`) and all DOM nodes with a *src* property as sinks. We also mark all control-flow statements as sinks. At the end of any file's execution, we include in the file's signature only those reads which propagate values to any of the sinks.

The *"Trimmed signature"* line in Figure 9 shows that trimming the signatures stored in Sprinter's compute cache improves the cache hit rate on the median site to over 80%. This is because a large fraction of reads performed by JS on the web does not influence the set of URLs fetched. Furthermore, we see that the cache hit rate with Sprinter is close to the best achievable hit rate, which we obtain via post-hoc analysis of JS executions to identify when the set of URLs fetched matched a prior execution. The gap between *"Trimmed signature"* and *"Oracle"* is due to Sprinter's conservative tracking of all control-flow dependencies, instead of only the ones that influence the URLs fetched.

4.2 Statically crawling pages

So far, we have discussed how Sprinter reuses execution across pages. However, as mentioned in §2.3, JS execution is only a part of the total compute overhead of web browsers. To maximize Sprinter's performance, we now discuss how it crawls pages without a browser in phase 3.

Crawling without a browser. We observe that the primary utility of crawling a page within a browser is its implementation of the JavaScript heap and the DOM object, and its

support of various APIs. However, if we are able to skip executing a file, we only need to compile the read state in its signature, for which we need a log of all the writes performed by previously executed or skipped JS files. We do not need to apply these writes to the browser's heap and DOM since there are no user interactions at the time of crawling.

Based on this insight, Sprinter's static crawler maintains a shadow heap, which is a key-value map from the properties of the heap to the corresponding values. It also maintains a shadow DOM, which it constructs by parsing the page's HTML at the start of every page load and offers the same read and write APIs as the ones provided by the browser.

For every page that it statically crawls in phase 3, Sprinter fetches the page's HTML, extracts all embedded resource URLs, and recursively fetches them. For every JS file fetched, the static crawler looks up the shadow heap and DOM to construct the file's signature. Upon a successful cache hit, Sprinter logs the writes included in the file's signature to the shadow heap and shadow DOM, and issues any resource fetches included in the signature. It repeats this process until all resources on the page have been fetched. Whenever there is a cache miss for a JS file, the static crawler is unable to execute the file, and it defers these pages for browser-based crawling in phase 4 (§4.3).

Handling additional fetches. Crawling pages as described above has the downside of fetching additional resources that a browser would not (as described in §2.2). For *Corpus*$_{10k}$, this increases the total number of bytes fetched by 3.5x. Unlike during dynamic crawling, when the network is severely underutilized (Figure 5), these additional fetches significantly degrade overall throughput when crawling without a browser.

If the input configuration to Sprinter specifies that only the resources relevant to the machine executing the crawler be downloaded, its static crawler does so by leveraging the browser's processing of pages crawled earlier in phase 2. First, during every page load executed within a browser, Sprinter adds to its compute cache the media queries evaluated and the corresponding value (true or false). For any media query encountered during browserless page loads, the static crawler fetches the corresponding URL if the compute cache either returns a true value or does not contain any entry for that media query. Similar to our observation of similarity in JS executions across pages, we find that, for the median site in *Corpus*$_{10k}$, 92% of all media queries occur on more than one page. Second, the static crawler uses the cached signatures for JS files to identify which selectors were applied when the browser executed those files. It fetches only the URLs contained within these selectors.

4.3 Scheduling page crawls

Given the high compute overhead of loading pages in a browser and extracting signatures, we must minimize the number of pages that Sprinter crawls using a browser. However, Sprinter can crawl a page without a browser only if it is able to skip executing *every* JS file on that page. Hence, the subset of pages on any site that Sprinter crawls without a browser in phase 3 should ideally be such that all of the JS files that appear on any of these pages also appear in at least one of the pages previously crawled with a browser in phase 2. This does not guarantee that the static crawler will find a compute cache entry with a matching signature for every JS file, but at least makes it possible.

Figure 10: **Approximate set cover captures a large fraction of JS files ("JS"), while the number of pages in the set cover ("Pages") are a small fraction of the total corpus size.**

Need for scheduling. Since the set of JS files on any page is not known apriori, Sprinter could use a browser to crawl the pages on any site in a random order and switch to browserless crawling once the set of JS files converges, i.e., once the union of JS files remains unchanged for n consecutive pages crawled. But, we find that there is no value of n that offers a good tradeoff between compute overheads and coverage of JS files. For example, with $n = 2$, we would need to crawl only 8% of pages on the median site in *Corpus*$_{10k}$ with a browser, but only 49% of the JS files seen across all the pages on this site appear on those pages. With $n = 10$, the fraction of JS files covered by browser-based loads increases to 82%; however, 38% of pages now need to be crawled using a browser.

Efficient identification of set cover. Sprinter takes an alternate approach of carefully selecting which subset of pages on each site to crawl using a browser in phase 2. Though we cannot predict which JS files are on the remaining pages, we leverage our finding from §3.1 that the same JS file often fetches the same resources across pages of a site. Therefore, instead of finding a subset of pages that includes *all* the JS files used on that site, we find a subset that includes all the JS files that are statically embedded in the remaining pages. When these files are executed as part of the browser-based loads, all the JS files that are dynamically fetched on this site's pages will likely be fetched and executed.

Thus, in phase 1, Sprinter crawls all pages using a static crawler which only fetches the JS files that are directly linked. We then have a set of JS files for every page, and Sprinter computes the set cover, i.e., the subset of sets whose union matches the union of all sets. Since computing the op-

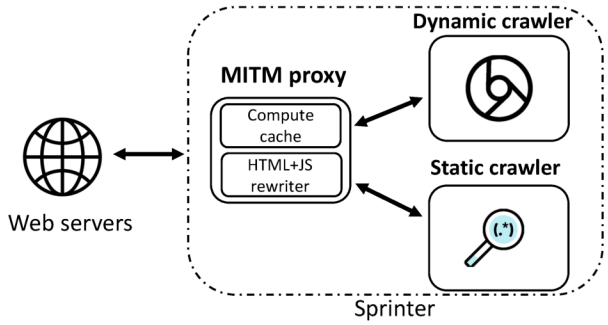


Figure 11: **Overview of our Sprinter implementation.**

timal set cover is NP-complete [29], we use a greedy heuristic which runs in polynomial time and is known to closely approximate the optimal [50]. Figure 10 shows that, on the median site in $Corpus_{10k}$, Sprinter selects only 7% of pages to be crawled using a browser, yet these pages cover over 80% of all the JS files seen across all pages on the site.

Given this methodology for choosing which pages to crawl with a browser in phase 2, there are multiple reasons why Sprinter's static crawler may not find a matching compute cache entry for every JS file that it fetches. First, since the set cover is only based on statically linked JS files, some dynamically fetched JS files may not have been encountered in the browser-based loads. Second, even for files that were executed by the browser, those executions may not have had the same signature as that expected by the static crawler. If the static crawler runs into either issue on any page, Sprinter recrawls that page using a browser in phase 4.

5 Implementation

Our implementation of Sprinter has three components, which work together as shown in Figure 11.

Dynamic crawler. Sprinter's dynamic crawler is written in 1150 lines of NodeJS code. This dynamic crawler allows Sprinter to load pages using Chrome in phases 2 and 4. To automate Chrome, the crawler uses the Puppeteer library [16]. At the end of every page load, it uses Chrome's DevTools protocol [8] to collect runtime information, and it then sends the per-file signatures that it compiles to the man-in-the-middle (MITM) proxy.

Static crawler. The static crawler, used in phases 1 and 3, is written in 920 lines of Golang. It uses the goquery library [13] to create a virtual DOM for every HTML file and the cascadia library [6] to parse and query CSS selectors.

MITM proxy. The MITM proxy is an HTTP proxy written in 350 lines of Golang. It intercepts requests and responses for every resource fetched by the static and dynamic crawlers. The proxy also statically analyzes and instruments JS files, for which it uses a static analyzer written in 1200 lines of NodeJS code. The static analyzer uses Babel [3], a JS transpiler, to create the abstract syntax tree for every JS file. While instrumenting JS files, the static analyzer enables tracking of the JS heap and DOM. Ideally *all* web APIs should be tracked to achieve perfect correctness since their return values can potentially influence URL fetches. However, our evaluation shows that the subset of APIs that our implementation currently supports – all values accessible from the `window` object directly, e.g., `window.navigator.userAgent` – largely suffices. Our finding is in accordance with prior work which studied the impact of web APIs on URL fetches [36]. The proxy also runs a gRPC server to receive signatures from the dynamic crawler.

6 Evaluation

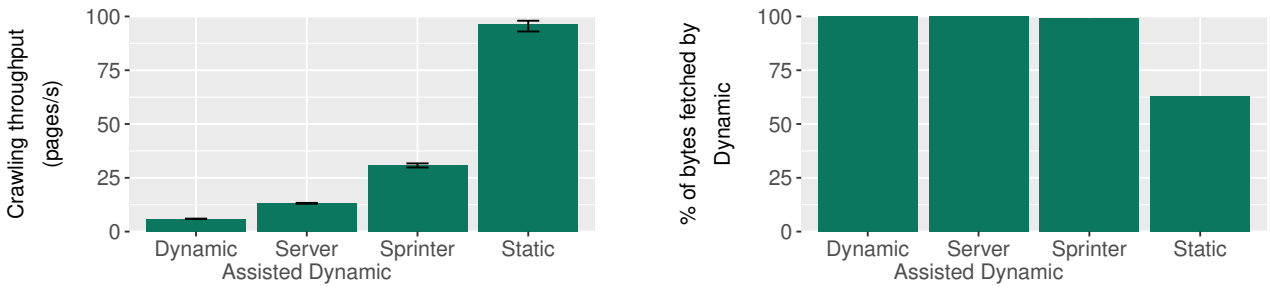
We evaluate Sprinter with respect to the fidelity with which it crawls pages and its performance in terms of crawling throughput. We also estimate the effort that would be required to maintain its implementation over time as web APIs evolve. We compare it to various options that exist for web crawling today. Our key findings are as follows:

- When compared to dynamic browser-based crawlers, Sprinter improves crawling throughput by 5x while preserving over 99% of the bytes fetched.
- Even in comparison to prior web accelerators that rely on assistance from web servers to reduce in-browser computations when loading pages, Sprinter delivers 2.4x higher crawling throughput.
- Sprinter's performance benefits significantly improve as more pages are crawled per site, e.g., its crawling throughput increases by 2.1x when the number of pages per site goes up from 100 to 500.
- Sprinter's performance improves as the same corpus is crawled repeatedly over time, e.g., the second crawl of our corpus is 78% faster than the first run 1 week earlier.

6.1 Evaluation setup

Workload. We expand our corpus of pages to include 500 randomly sampled pages in each of 100 sites. This new corpus, which we refer to as $Corpus_{50k}$, retains the same properties as $Corpus_{10k}$: diverse set of sites, and representative of real-world crawling workloads in having a large number of pages per site. As described in §2.2, we crawl every page in this new corpus using a custom crawler which fetches the resources downloaded by either dynamic or static crawlers, and we use a record-replay tool [9] to record all request-response headers along with the corresponding payloads.

Hardware configuration and crawling methodology. We store the recorded pages in an SSD drive of a Linux server which hosts 450 web servers to concurrently service HTTP requests with the appropriate recorded content. The crawlers run on a different Linux server with a 16-core 2.1 GHz Intel Xeon CPU, 128 GB RAM, and a 1 Gbps Ethernet connection to the server housing recorded pages. Crawling performance in this setup matches what we see when crawling pages from the live web, but eliminates the impact of any server-side effects on our evaluation of performance and fidelity.



(a) (b)

Figure 12: **Comparison of (a) crawling throughput and (b) fidelity of Sprinter against the three baselines.**

Baselines. Our primary baselines represent existing static and dynamic crawlers. For the static approach, we port wget2, a popular open-source crawler, to be compatible with our proxy-based setup; we verified that our static crawler is identical to wget2 in terms of fetched content. For the dynamic approach, we first considered three popular open-source crawlers: Archivebox [2], Browsertrix [4], and Brozzler [5]. However, our benchmark results for each revealed substantial performance drawbacks, likely because their primary goal was high fidelity, not necessarily high throughput. Specifically, undue overheads stem from spawning a new browser instance for each crawled page, using a CPU-intensive MITM proxy, and relying on an outdated Chrome automation framework. Therefore, we instead built an in-house Chrome-based crawler that achieves 20%, 33%, and 250% higher throughput than Archivebox, Browsertrix, and Brozzler, respectively. We verified that our custom crawler fetches the same set of resources as Archivebox when used to crawl the landing pages for the 100 sites in $Corpus_{50k}$.

Our third baseline is representative of prior server-/proxy-assisted solutions to reduce client-side computations in user-facing page loads [42, 57]. To the best of our knowledge, none of these systems are open sourced, and we are unaware of any domains that have adopted these techniques. Therefore, to evaluate Sprinter against this prior work, we consider the best case outcome of these systems, where *all* client-side JS execution is eliminated. We mimic such a scenario by using our Chrome-based crawler to crawl a version of every page wherein we include links to all the resources fetched by JS files in the page's main HTML. The browser loads this modified HTML with JS execution disabled. We refer to this baseline as *server assisted dynamic* crawling.

Metrics. We measure the crawling throughput of each crawler as the average number of pages it can crawl per second on a single server. For each crawler, we run a sufficiently large number of instances so as to saturate either the CPU or the network. We expect crawling throughput to linearly increase with the number of servers. We run 5 trials for each experiment and plot the median value, with error bars plotting the minimum and the maximum values.

We consider the default goal of crawling to be to match a Chrome-based crawler. Therefore, we measure the fidelity offered by a crawler as the fraction of bytes it fetches of all the resources fetched by Chrome when crawling the same pages. When the goal is to crawl all resources that are relevant to any client device, we measure fidelity as the fraction of bytes fetched out of the union of the resources fetched by the static and dynamic crawlers.

6.2 Throughput and Fidelity

6.2.1 Comparison with baselines

To compare Sprinter with the three baselines, we load pages in $Corpus_{50k}$ using each of the four crawlers separately. We monitor the resources fetched by each crawler on every page, and the total time taken to finish crawling the entire corpus. We also monitor the CPU and network utilization to identify the bottleneck for each crawler.

Figure 12(a) plots the crawling throughput achieved with each crawler, and Figure 12(b) shows the fidelity achieved.[2] Static crawler achieves the best crawling throughput by far of 96 pages per second. However, it misses out on 37% of the bytes fetched by the dynamic crawler. In contrast, the dynamic crawler could only crawl at a rate of 6 pages per second. Since CPU utilization was at 100% throughout the entirety of the crawl with the dynamic crawler, throughput increased to 13 pages per second with the server-assisted dynamic crawler, which does not execute any JS.

Sprinter offers a significant additional speedup, improving crawling throughput to 31 pages per second, a 5x improvement relative to the dynamic crawler. Importantly, it does so without requiring any changes to the web and while preserving 99.2% of the bytes fetched by the dynamic crawler. The 0.8% of bytes that went unfetched stem from the incomplete support for all web APIs in our current implementation. 50% of these unfetched bytes correspond to JavaScript files, 27% to images, and 17% to HTMLs, with the remaining accounted for by CSS and other content types. While no resources went unfetched on the median page, the 90th percentile page was missing 1 resource.

[2] We see no variation across runs in the resources fetched by each crawler because all of our crawls rely on one snapshot of every page crawled from the live web.

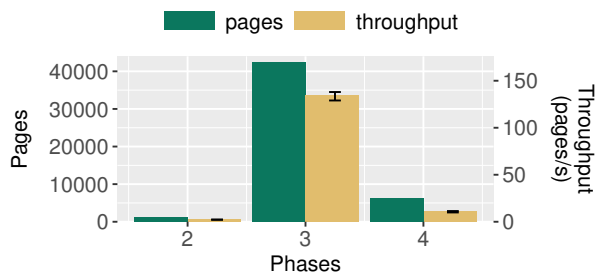


Figure 13: **Number of pages crawled during each of the different phases of Sprinter and the corresponding throughput achieved in each phase.**

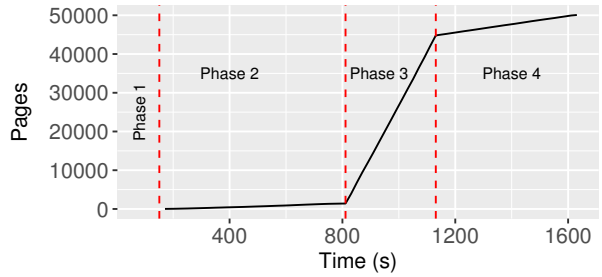


Figure 14: **A timeline of Sprinter's crawl of *Corpus$_{50k}$*, showing the duration and number of pages crawled in each phase.**

6.2.2 *Throughput in each phase*

Sprinter's crawling throughput varies widely across phases. Figure 13 plots the number of pages crawled in each phase and the corresponding throughput. Whereas, Figure 14 shows a timeline of how Sprinter's crawling of the pages in *Corpus$_{50k}$* proceeds over time.

- No page is fully crawled in phase 1; Sprinter only statically crawls the HTML files and embedded JavaScript files for every page so as to identify the subset of pages to be crawled with a browser in phase 2. Therefore, phase 1 finishes in 151s, the quickest of all four phases.
- Phase 2 is the slowest since Sprinter not only has to crawl pages with a browser, but it also has to incur the overheads of statically analyzing and rewriting every JavaScript file, executing these instrumented files inside Chrome, and processing the information it collects to generate and store per-file signatures. In this phase, Sprinter crawls 1413 pages in 620s, resulting in a crawling throughput of a little over 2 pages per second.
- Sprinter crawls the vast majority of pages in phase 3: 42497 pages in 316s. The average throughput of 135 pages per second in this phase is even higher than what a static crawler can achieve (96 pages per second, as shown in Figure 12(a)). This is because, unlike a static crawler, Sprinter leverages browser-based execution of media queries and CSS selectors in phase 2 to eliminate fetches of resources relevant only for other client types.
- In Phase 4, Sprinter recrawls the remaining 6090 pages with a browser; about a quarter of these are because they contained a JS file not executed in phase 2, and the remaining pages incurred at least one compute cache miss. The crawling throughput of 11 pages per second in this phase is better than in phase 2 because significantly fewer JS files need to be instrumented.

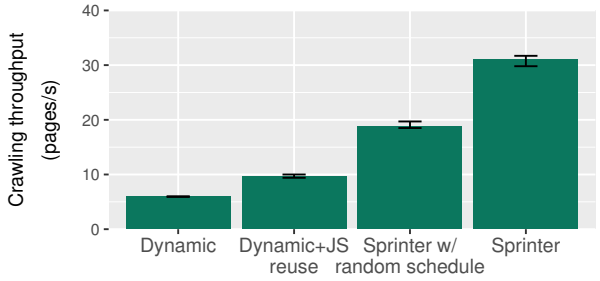


Figure 15: **Incremental benefit offered by each of the techniques used in Sprinter.**

At the end of phase 4, Sprinter's compute cache had 3089 entries. The cache hit rate of 95.6% is the key enabler of Sprinter's throughput improvements as it could crawl a large fraction of pages in phase 3, without requiring a browser. We cannot further reduce the total crawl time by immediately spawning a browser to crawl any page that incurs a cache miss in phase 3 because both phases 3 and 4 are bottlenecked by the CPU.

6.2.3 *Contribution of techniques*

To understand the performance benefits of each of the techniques used in Sprinter, we incrementally add them to the dynamic crawler and measure crawling throughput.

First, we evaluate the benefits of only using JS memoization (§4.1) in Chrome, loading all pages in the corpus in a random order. Figure 15 shows that *"Dynamic+JS reuse"* provides a roughly 66% speedup over *"Dynamic"*.

Next, we crawl some of the pages with a browser and the rest using Sprinter's augmented static crawler (§4.2). To determine which pages to crawl using a browser, we consider the strawman approach (§4.3) wherein we transition to browserless crawling once the union of JS files remains unchanged for n consecutive pages. For *Corpus$_{50k}$*, we observe that $n = 25$ results in browser-based loads fetching the same fraction of all JS files as that covered by Sprinter's chosen set cover. Even this unsophisticated combination of dynamic and static crawling – *"Sprinter w/ random schedule"* in Figure 15 – roughly doubles the crawling throughput.

Finally, by efficiently choosing a carefully chosen subset of pages to crawl with a browser, Sprinter crawls 88% fewer pages using a browser in phase 2, resulting in a further 1.6x improvement in throughput.

6.3 Sensitivity to crawling parameters

We evaluate the impact of the following three configuration parameters on Sprinter's crawling throughput: 1) the number of pages crawled per site, 2) the time gap between repeated

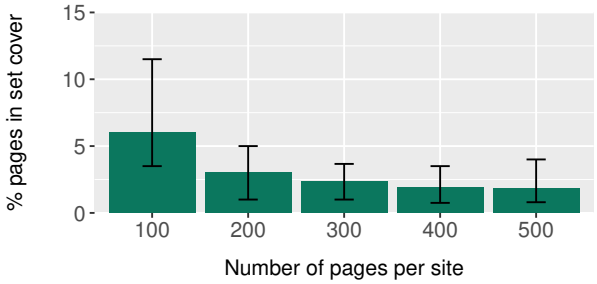


Figure 16: **Percentage of pages selected by Sprinter for browser-based crawling as a function of number of pages crawled per site. Bars show value for median site, with error bars for the 25th and 75th percentiles.**

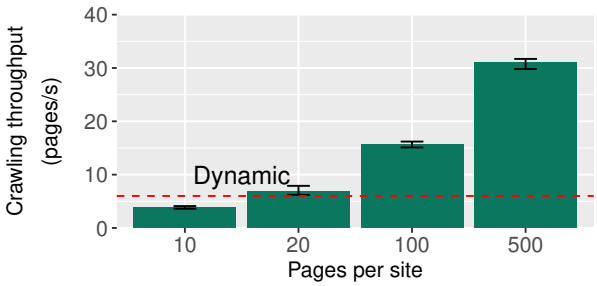


Figure 17: **Sprinter's crawling throughput as a function of the number of pages per site.**

crawls, and 3) whether fetching all statically embedded resource URLs is desired.

6.3.1 *Number of pages per site*

The key to Sprinter's high crawling throughput is its judicious partitioning of pages, crawling a small fraction using a browser and the remaining without. We examine how the fraction chosen for browser-based crawling varies as a function of the number of pages being crawled per site. For 5 different values of the number of pages per site, Figure 16 plots this fraction for the 25th, median, and 75th percentile sites. The percentage of pages in Sprinter's carefully selected "set cover" for the median site goes down from 6% with 100 pages per site to 1.6% with 500 pages per site. As a result, Sprinter is able to crawl a corpus of 10k pages at an average rate of 15 pages per second. But, for a 50k page corpus, its throughput improves to 31 pages per second (Figure 17). Akin to how a static crawler benefits more from network caching with more redundant resource fetches, Sprinter's compute cache enables it to reuse more client-side computations when it crawls more pages per site.

On the flip side, lower the number of pages per site, lower Sprinter's throughput. Figure 17 shows that, with 10 pages per site, Sprinter crawls 4 pages per second on average, which is slower than the dynamic crawler. For Sprinter to offer any benefit, we see that it must be asked to crawl at least 20 pages per site. As a result, workloads that only crawl landing pages of sites [10] will not benefit from Sprinter.

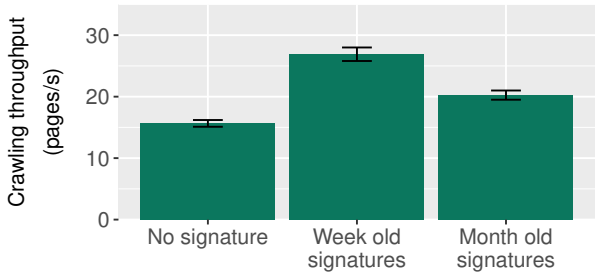


Figure 18: **Sprinter can crawl pages faster by leveraging signature information from previous crawls of the same corpus.**

6.3.2 *Repeated crawling*

In many web crawling workloads, the same corpus of pages is repeatedly recrawled. For example, a web search engine must ensure that its search index reflects the latest content on every page, and web archives must track changes to page content over time. In such cases, Sprinter will crawl the entire corpus in 4 phases the first time. However, when the corpus is recrawled, Sprinter can directly jump to crawling pages statically in phase 3, leveraging JS execution signatures from the previous crawls. Pages where no compute cache entry was found for at least one JS file would have to be recrawled with a browser in phase 4.

To measure the crawling throughput with Sprinter when the same corpus is crawled multiple times, we recrawl $Corpus_{10k}$ once three weeks after our initial crawl, and again a week later. We then use Sprinter in our replay setup to crawl pages from our last copy of the corpus. We run Sprinter once starting with an empty compute cache, once using signatures from the crawl a week before, and once using signatures from the crawl a month before.

Figure 18 shows that reusing signatures from a week ago improves Sprinter's throughput by 78% as compared to when no prior crawl existed. Reusing month-old signatures also speeds up Sprinter. But, since the compute cache entries are more stale and more previously unseen JS files are fetched, the benefits are significantly lower.

6.3.3 *Preserving static fetches*

Thus far in our evaluation, we have considered the goal of crawling to be to fetch the same resources on every page as a dynamic crawler. However, in some cases, it might be desirable to also crawl all resources that would be fetched by a static crawler. For example, web archivists might want to preserve all versions of every image on a page, so as to be able to accurately render the preserved page irrespective of the client device used to visit this page in the future.

In these cases, Sprinter can be configured to not eliminate fetches using the techniques mentioned in §4.2. The resultant throughput of Sprinter drops to 28 pages per second which, though 9% lower than when it only tries to match the dynamic crawler, is still 4.6x faster than the dynamic crawler. This drop in throughput is because of Sprinter's static crawler having to fetch additional bytes in phase 3.

Chrome version	Lightweight browser		Sprinter	
	# of APIs added	# of files added/-modified	# of APIs added	LOC added
v108	4	41	1	9
v109	3	70	1	13
v110	4	58	1	6
v111	7	109	0	0
Total	18	278	3	28

Table 1: **Comparison of number of APIs that need to handled by Sprinter and a lightweight browser.**

Note that the impact of this configuration option on Sprinter's throughput depends on the number of pages crawled per site. With more pages per site, phase 3 is able to achieve higher crawling throughput due to the benefits of network caching.

6.4 Maintainability

Web APIs and their specifications are constantly updated [38]. Web crawlers need to be correspondingly updated over time to ensure that web pages using the latest APIs are accurately crawled. Dynamic crawlers leveraging web browsers such as Chrome and Firefox simply need to update to the latest version of the browser, as these browsers are well-maintained and constantly updated to support most of the latest web APIs.

To get a measure of the effort that would be needed to maintain Sprinter or a lightweight browser such as phantomJS, we look at all the APIs added in the 4 most recent versions of Chrome (v108 to v111). For each API, we manually read its specification. Only a subset of these would need to be implemented by a lightweight browser designed for the purpose of crawling, e.g., any API that takes effect only during user interactions (such as $webRTC$ APIs to enable video conferencing or $navigator.credentials$ API to enable secure logins) would not have to be handled. Sprinter's instrumentation of JS code would need to keep track of an even smaller subset of APIs, only those which influence execution signatures, i.e., any API that can read from or write to the global state.

Table 1 compares the number of APIs that need to be tracked and implemented by Sprinter versus a lightweight browser designed for crawling. Across the four versions, a lightweight browser would be required to implement 18 APIs; in Chrome's source, these APIs touch 278 files (Chrome's commit history only shows files added/modified, not the number of lines of code). In contrast, Sprinter needs to handle only 3 of these APIs, requiring 28 lines of code.

7 Related Work

Scalable web crawling. The engineering issues associated with web crawling are well studied [33, 58, 19, 37, 22, 23]. Some of these crawlers [37, 23, 22] are able to achieve a crawling throughput of upwards of 1000 pages per server. However, all of these crawlers only download the HTML file for every page URL. In contrast, Sprinter downloads all the resources which would be fetched by browser-based crawlers such as Archivebox [2], Brozzler [5], and Browsertrix [4].

Incremental crawling A large amount of prior work [27, 39, 25, 52] has focused on incremental web crawling, i.e., how to efficiently recrawl pages. These techniques are helpful only when the same set of pages are crawled multiple times. Sprinter, on the other hand, eliminates redundant computations across pages even within a single crawl.

Resource bottlenecks of large-scale distributed systems. Prior work has studied the bottlenecks in scaling various distributed data processing workloads such as sorting [45], data analytics [43], and distributed deep learning [54, 48, 20]. These efforts first identify the hardware resource (CPU, GPU, network, or disk) that constrains overall performance, and then propose solutions to optimize the utilization of that resource. To the best of our knowledge, we are the first to study the compute bottleneck in browser-based web crawling and propose a solution to reduce its impact.

Web performance optimization. The negative impact of a web browser's computations on user-perceived latency while loading web pages is well-known [55, 56]. As discussed earlier (§2.4), proposals to lower page load times either do not reduce the total amount of computation that web clients need to perform [41, 46, 40] or require server-side changes [42, 57]. Sprinter is backward compatible with the legacy web and reduces the total amount of client-side processing by memoizing and reusing computations across pages on the same site.

Compute memoization. Memoization is widely used across different kinds of application. Prior work has leveraged such techniques to reduce compile-time latency [51, 34], improve runtime performance [30, 53], minimize scheduling overheads [26], and enable faster auditing of web applications [35]. Sprinter uses similar memoization techniques to reduce the amount of client-side computation required to crawl pages, and it maximizes its benefits by selectively identifying the state that influences URL fetches.

8 Conclusion

Over the years, crawling web pages with high fidelity has evolved from a workload that is limited by network bandwidth to a CPU-intensive one. In this paper, we showed that the key to mitigating this new bottleneck is to strategically minimize the use of the web browser and its execution of JavaScripts. Our design of Sprinter does so by efficiently identifying and exploiting opportunities to safely reuse the browser's computations across the pages on any site. We hope that our work will spur a new wave of innovation in scalable web crawling, a task that underlies many important systems in today's society.

REFERENCES

[1] Amazon silk. https://docs.aws.amazon.com/silk/latest/developerguide/what-is-silk.html.

[2] Archivebox. https://github.com/ArchiveBox/ArchiveBox.

[3] Babel. https://babeljs.io/.

[4] Browsertrix crawler. https://github.com/webrecorder/browsertrix-crawler.

[5] Brozzler. https://github.com/internetarchive/brozzler.

[6] Cascadia. https://github.com/andybalholm/cascadia.

[7] Chrome cpu profiler. https://developer.chrome.com/docs/devtools/performance//.

[8] Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/.

[9] Chrome web page replay. https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/README.md.

[10] Common crawl. https://commoncrawl.org/.

[11] CSS selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

[12] Gnu wget2. https://github.com/rockdaboot/wget2.

[13] Goquery. https://github.com/PuerkitoBio/goquery.

[14] Internet archive end of term 2020 web crawls. https://archive.org/details/EndOfTerm2020WebCrawls.

[15] Optional chaining. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining.

[16] Puppeteer. https://pptr.dev/.

[17] Using media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries.

[18] Web APIs. https://caniuse.com/?compare=chrome+114,firefox+113&compareCats=all.

[19] F. Ahmadi-Abkenari and A. Selamat. An architecture for a focused trend parallel web crawler with the application of clickstream analysis. *Information Sciences*, 2012.

[20] T. Akiba, K. Fukuda, and S. Suzuki. Chainermn: Scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351*, 2017.

[21] A. S. Bale, N. Ghorpade, S. Rohith, S. Kamalesh, R. Rohith, and B. Rohan. Web scraping approaches and their performance on modern websites. In *International Conference on Electronics and Sustainable Communication Systems*, 2022.

[22] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 2004.

[23] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses. *ACM Transactions on the Web*, 2018.

[24] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI*, 2015.

[25] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB*, 2000.

[26] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[27] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *WWW*, 2001.

[28] M. Erdélyi, A. A. Benczúr, J. Masanés, and D. Siklósi. Web spam filtering in internet archives. In *International Workshop on Adversarial Information Retrieval on the Web*, 2009.

[29] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 1998.

[30] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha. Rethinking client-side caching for the mobile web. In *HotMobile*, 2021.

[31] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *OSDI*, 2022.

[32] G. Gossen, E. Demidova, and T. Risse. ICrawl: Improving the freshness of web collections by integrating social web and focused web crawling. In *JCDL*, 2015.

[33] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. In *WWW*, 1999.

[34] M. Johnson. Memoization of top down parsing. *arXiv preprint cmp-lg/9504016*, 1995.

[35] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *OSDI*, 2012.

[36] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating page loads using symbolic execution. In *NSDI*, 2021.

[37] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: scaling to 6 billion pages and beyond. *ACM Transactions on the Web*, 2009.

[38] J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service API evolution affect clients? In *IEEE International Conference on Web Services*, 2013.

[39] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[40] S. Mardani, A. Goel, R. Ko, H. Madhyastha, and R. Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *OSDI*, 2021.

[41] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.

[42] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.

[43] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[44] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, 2016.

[45] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *NSDI*, 2011.

[46] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *SIGCOMM*, 2017.

[47] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[48] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[49] S. M. Shariff, H. Li, C.-P. Bezemer, A. E. Hassan, T. H. Nguyen, and P. Flora. Improving the testing efficiency of selenium-based load tests. In *International Workshop on Automation of Software Test*, 2019.

[50] P. Slavík. A tight analysis of the greedy algorithm for set cover. In *STOC*, 1996.

[51] A. Suresh, E. Rohou, and A. Seznec. Compile-time function memoization. In *International Conference on Compiler Construction*, 2017.

[52] Q. Tan and P. Mitra. Clustering-based incremental web crawling. *ACM Trans. Inf. Syst.*, 2010.

[53] Y. Tang and J. Yang. Secure deduplication of general computations. In *USENIX ATC*, 2015.

[54] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, N. Prajapati, P. McCormick, J. Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *OSDI*, 2022.

[55] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *NSDI*, 2013.

[56] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[57] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with Shandian. In *NSDI*, 2016.

[58] Q. Zheng, Z. Wu, X. Cheng, L. Jiang, and J. Liu. Learning to crawl deep web. *Information Systems*, 2013.