WebAssembly-based Delta Sync for Cloud Storage Services

JIANWEI ZHENG, ZHENHUA LI, YUANHUI QIU, HAO LIN, HE XIAO, YANG LI, and YUNHAO LIU, Tsinghua University

Delta synchronization (sync) is crucial to the network-level efficiency of cloud storage services, especially when handling large files with small increments. Practical delta sync techniques are, however, only available for PC clients and mobile apps, but not web browsers—the most pervasive and OS-independent access method. To bridge this gap, prior work concentrates on either reversing the delta sync protocol or utilizing the native client, all striving around the tradeoffs among efficiency, applicability, and usability and thus forming an "impossible triangle." Recently, we note the advent of WebAssembly (WASM), a portable binary instruction format that is efficient in both encoding size and load time. In principle, the unique advantages of WASM can make web-based applications enjoy near-native runtime speed without significant cloud-side or client-side changes. Thus, we implement a straightforward WASM-based delta sync solution, WASMrsync, finding its quasi-asynchronous working manner and conventional In-situ Separate Memory Allocation greatly increase sync time and memory usage. To address them, we strategically devise sync-async code decoupling and streaming compilation, together with Informed In-place File Construction. The resulting solution, WASMrsync+, achieves comparable sync time as the state-of-the-art (most efficient) solution with nearly only half of memory usage, letting the "impossible triangle" reach a reconciliation.

CCS Concepts: • **Information systems** → **Information storage systems**; *Storage architectures*; **Cloud based storage**;

Additional Key Words and Phrases: Cloud storage service, delta synchronization, web browser, WebAssembly

ACM Reference format:
Jianwei Zheng, Zhenhua Li, Yuanhui Qiu, Hao Lin, He Xiao, Yang Li, and Yunhao Liu. 2022. WebAssembly-based Delta Sync for Cloud Storage Services. *ACM Trans. Storage* 18, 3, Article 24 (September 2022), 31 pages.
https://doi.org/10.1145/3502847

1 INTRODUCTION

Recent years have witnessed the enormous popularity of cloud storage services, such as Dropbox, Google Drive, iCloud Drive, and Microsoft OneDrive. They have not only provided a convenient and pervasive data store for billions of Internet users, but also become a critical component of other online applications. Their popularity brings a large volume of network traffic overhead to both the client and cloud sides [18, 38]. Thus, a lot of efforts have been made to improve their network-level

This work is supported in part by the National Natural Science Foundation of China (NSFC) under grants 61822205, 61902211 and 62202266, as well as Microsoft Research Asia.

Authors' address: J. Zheng, Z. Li (corresponding), Y. Qiu, H. Lin, H. Xiao, Y. Li, and Y. Liu, Tsinghua University, Beijing; emails: zhengjw19@mails.tsinghua.edu.cn, lizhenhua1983@gmail.com, qiuyh0924@163.com, linhaomails@gmail.com, xiaoh16@gmail.com, liyang14thu@gmail.com, yunhaoliu@gmail.com.

1553-3077/2022/09-ART24 $15.00
https://doi.org/10.1145/3502847

efficiencies, such as batched sync, deferred sync, delta sync, compression, and deduplication [12, 13, 17, 38, 39, 57]. Among these efforts, delta sync is of particular importance for its fine granularity (i.e., the client only sends the changed content of a file to the cloud, instead of the entire file), thus achieving significant traffic savings in the presence of users' file edits [19, 40, 41].

Unfortunately, today delta sync is only available for PC clients and/or mobile apps in state-of-the-art commercial cloud storage services (as detailed in Section 2), but not for the web—the most pervasive and OS-independent access method. Taking Dropbox as an example, after a file f is edited into a new version f' by a user, Dropbox's PC client will apply delta sync to automatically upload only the altered bits to the cloud; In contrast, Dropbox's web interface requires users to manually upload the *entire content* of f' to the cloud.[1] This gap significantly affects web-based user experiences in terms of both sync speed and traffic cost [19, 46, 47].

Web is a popular access method for cloud storage services [20, 30, 32]: All major services support web-based access, while only providing PC clients and mobile apps for a limited set of OS distributions and devices. One reason is that many users do not want to install PC clients or mobile apps on their devices to avoid the extra storage and CPU/memory overhead; in comparison, almost every device has web browsers. Specially, for the emerging cloud-oriented systems and devices (e.g., Chrome OS and Chromebook) web browsers are perhaps the only option to access cloud storage.

To understand the fundamental obstacles of web-based delta sync, we implement a delta sync solution, WebRsync, using state-of-the-art web techniques including JavaScript, WebSocket, and HTML5 File APIs [31, 45] to achieve fast file I/O and transfer. WebRsync implements the algorithm of `rsync` [49] and works with all modern web browsers. Our experiments show that the sync speed of WebRsync is much lower than that of PC client-based `rsync`, since WebRsync is severely affected by the low execution efficiency of JavaScript inside web browsers.

Thus, our first approach to optimizing WebRsync is to leverage the native client [25], a sandbox for efficiently and securely executing compiled C/C++ code in a web browser. We embody this approach by developing WebRsync-native, and the experiments show that it can achieve comparable sync time as PC client-based `rsync`. Nevertheless, using the native client requires the client side to download and install plug-ins for web browsers, which substantially impairs the usability and pervasiveness of WebRsync-native.

Driven by above observations, our second effort towards efficient web-based delta sync is to "reverse" the WebRsync process by handing computation-intensive operations over to the server side. The resulting solution is named **WebR2sync (Web-based Reverse `rsync`)**. It significantly cuts the computation burden on the web client, but brings considerable computation overhead to the server side. To this end, we make additional two-fold efforts to optimize the server-side computation overhead. First, we exploit the locality of users' file edits, which can help bypass most (up to ~90%) chunk search operations in real usage scenarios. Second, by leveraging lightweight checksum algorithms SipHash [9] and Spooky [11] instead of MD5, we can reduce the complexity of chunk comparison by ~5 times. The solution is referred to as WebR2sync+.

WebR2sync+, however, breaks the original workflow of `rsync` due to the reverse scheme. Hence, mainstream cloud storage systems need to refactor their architectures for adopting WebR2sync+, which greatly restricts the industrial applicability of WebR2sync+. Also, the collision probability of SipHash is much higher than that of MD5. Thus, WebR2sync+ runs risks of making more sync errors than other solutions, which greatly impairs the system reliability of WebR2sync+.

[1]In this article, we focus on *pervasive* file editing made by any applications that synchronize files to the cloud storage through web browsers, rather than *specific* web-based file editors such as Google Docs, Microsoft Word Online, Overleaf, and GitHub online editor. Technically, our measurements show that the latter usually leverages specific data structures (rather than delta sync) to avoid full-content transfer and save the network traffic incurred by file editing.

The above-described solutions, including WebRsync, WebRsync-native, and WebR2sync+, seem to be all striving around the tradeoffs among efficiency, applicability, and usability, as comparatively listed in Table 1. Nevertheless, we still wish to explore the possibility of building an efficient web-based delta sync solution without sacrificing applicability or usability.

Recently, an emerging technique (or web language) called *WebAssembly* [54] (or WASM for short) has been attracting increasing attention in the design of web applications. WASM is a portable binary instruction format (for web programming languages) that is efficient in both encoding size and load time. The unique advantages of WASM seem to be likely to enable web-based delta sync applications to enjoy near-native runtime speed without requiring significant cloud-side or client-side changes. Thereby, we explore how to leverage WASM to enable efficient web-based delta sync in a user-friendly and easy-to-apply manner (i.e., to make the best of both worlds).

Observing that the calculation of MD5 checksums incurs the most computation overhead at the client side, we leverage WASM to carry out this operation, resulting in the preliminary solution called WASMrsync. To avoid blocking the main thread that may make the web page unresponsive, WASM APIs are designed to be asynchronous. Therefore, we have to implement WASMrsync with a quasi-asynchronous strategy, i.e., to implement the asynchronous WASM API invocations in a synchronous manner using the await/async mechanism of JavaScript. Unfortunately, there are overheads associated with running the await/async method: The current execution context has to be captured, there is a thread transition, and a state machine is built through which the code runs. Worse still, this await/async property of a WASM API invocation is propagated to the wrapping function of the invocation, which introduces unnecessary overhead for executing even non-WASM code in the wrapping function and thus severely increases the overall sync time. To mitigate this shortcoming, we strategically devise *sync-async code decoupling* that extracts the synchronous non-WASM code from asynchronous modules to avoid unnecessary await/async overhead.

Moreover, to improve the loading efficiency of asynchronous WASM modules, we adopt *streaming compilation* (a novel WASM API) to download and compile WASM modules in parallel. When using WASM, the browser typically needs to download, compile, and instantiate a WASM module successively, and then call functions exported from that module through JavaScript. With streaming compilation, the browser can start compiling WASM modules while the browser is still downloading the module bytes. As soon as the browser downloads all bytes of a single function, the function can be passed to a background thread to compile it, which essentially improves the loading efficiency.

In addition, when handling large files, we notice WASMrsync suffers from the time- and memory-consuming file construction operations at the server side, which stem from the conventional **In-situ Separate Memory Allocation (ISMA)** mechanism. Hence, we devise the **Informed In-place File Construction (IIFC)** mechanism by (1) encoding the actions to every chunk of the updated file (ADD or COPY) with a dependency graph, (2) topologically sorting the graph, and (3) delivering the processed information from the client side to the server side. As a result, we only need to perform a single incremental memory allocation based on the memory space occupied by the old file.

The eventual solution is named WASMrsync+. We evaluate the performance of WebRsync-native, WebR2sync+, and WASMrsync+ by building prototype systems based on a Dropbox-like system architecture. The results show that WebRsync-native can significantly reduce the sync time of WebRsync—in fact, close to the sync time of `rsync`—while sacrificing usability. WebR2sync+ outpaces WebRsync by an order of magnitude, also approaching the performance of `rsync`, but at the cost of limited applicability. In contrast, WASMrsync+ achieves comparable sync time as WebR2sync+ (the former only takes 10%~20% more sync time than the latter) and saves ~50% server-side memory usage, but without impairing the reliability, applicability, and usability.

Table 1. Brief Comparison of the Web- and WASM-based Delta Sync Solutions as Proposed and Implemented in this Article, in Terms of Technique(s), Stagnation (of Web Browser), Efficiency, Applicability, and Usability

System	Technique(s)	Stagnation	Efficiency	Applicability	Usability
WebRsync	HTML File APIs and WebSocket	Yes	Low	Smooth adoption	Easy
WebRsync -native	Native client	No	High	Smooth adoption	Extra plug-in installation
WebRsync -parallel	HTML5 web workers	No	Low	Smooth adoption	Easy
WebRsync+	Edit locality exploiting and lightweight hashing	Yes	Moderate	Smooth adoption	Easy
WebR2sync	Reversing the workflow of `rsync`	No	Moderate	System architecture refactoring	Easy
WebR2sync+	WebR2sync with server-side optimizations	No	High	System architecture refactoring	Easy
WASMrsync	WebAssembly	No	Moderate	Smooth adoption	Easy
WASMrsync+	WASMrsync with both client- and server-side optimizations	No	High	Smooth adoption	Easy

In summary, our work makes the following contributions:

- We elaborate the methodology for deeply and comprehensively studying web- and WASM-based delta sync solutions, involving the key metrics and workloads for quantifying their performance, as well as an automated tool for accurately quantifying the *stagnation* of web browsers (Section 3).
- We propose a variety of web-based delta sync solutions in a step-by-step optimization manner, to reveal the challenges and opportunities of supporting delta sync in traditional frameworks (Section 4).
- We explore the practical feasibility of leveraging WASM to enable efficient web-based delta sync in a user-friendly and easy-to-apply manner by developing the first workable **WASM-based delta sync solution (WASMrsync)** (Section 5).
- We implement an efficient WASM-based delta sync solution (WASMrsync+) without sacrificing applicability or usability by devising both the client- and server-side optimizations (Section 6).

At last, we compare in Table 1 all the web- and WASM-based delta sync solutions as proposed and implemented in this article so readers can easily figure out how they vary. We also make all their source code publicly available at https://WASMDeltaSync.github.io.

2 RELATED WORK AND STATUS QUO

Delta sync, known as delta encoding or delta compression, is a way of storing or transmitting data in the form of differences (deltas) between different versions of a file, rather than the complete content of the file [56]. It is very useful for network applications where file modifications or incremental data updates frequently happen, e.g., storing multiple versions of a file, distributing consecutive user edits to a file, and transmitting video sequences [28]. In the past four decades, a variety of delta sync algorithms or solutions have been put forward, such as UNIX `diff` [27], `Vcdiff` [33], WebExpress [26], Optimistic Deltas [10], `rsync` [49], and **content defined chunking (CDC)** [34].

Due to its efficiency and flexibility, `rsync` has become the de facto delta sync protocol widely used in practice. It was originally proposed by Tridgell and Mackerras in 1996 as an algorithm for efficient remote update of data over a high-latency, low-bandwidth network link [52]. Then, in 1999, Tridgell thoroughly discussed its design, implementation, and performance in Reference [51]. Being a standard Linux utility included in all popular Linux distributions, `rsync` has also been ported to Windows, FreeBSD, NetBSD, OpenBSD, and MacOS [49].

According to a real-world usage dataset [38], the majority (84%) of files are modified by the users at least once, thus confirming the importance of delta sync on the network-level efficiency of cloud storage services. Among all mainstream cloud storage services, Dropbox was the first to adopt delta sync (more specifically, `rsync`) in around 2009 in its PC client-based file sync process [40]. Then, SugarSync, iCloud Drive, OneDrive, and Seafile followed the design choice of Dropbox by utilizing delta sync (`rsync` or CDC) to reduce their PC clients' and cloud servers' sync traffic. After that, two academic cloud storage systems, namely, QuickSync [13] and DeltaCFS [64], further implemented delta sync (`rsync` and CDC, respectively) for mobile apps.

Drago et al. studied the system architecture of Dropbox and conducted large-scale measurements based on ISP-level traces of Dropbox network traffic [18]. They observed that the Dropbox traffic was as much as one-third of the YouTube traffic, which strengthens the necessity of Dropbox's adopting delta sync. Li et al. investigated in detail the delta sync process of Dropbox through various types of controlled benchmark experiments and found it suffers from both traffic and computation overuse problems in the presence of frequent, short data updates [40]. To this end, they designed an efficient batched synchronization algorithm called **UDS (update-batched delayed sync)** to reduce the traffic usage, and further extended UDS with a backwards compatible Linux kernel modification to reduce the CPU usage (recall that delta sync is computation intensive).

Despite the wide adoption of delta sync (particularly `rsync`) in cloud storage services, practical delta sync techniques are currently only available for PC clients and mobile apps rather than web browsers. We conduct a qualitative study of delta sync support in state-of-the-art cloud storage services. The target services are selected for either their popularity (Dropbox, Google Drive, Microsoft OneDrive, iCloud Drive, and Box.com) or representativeness in terms of techniques used (SugarSync, Seafile, QuickSync, and DeltaCFS). For each service, we examined its delta sync support with different access methods, using its latest-version (as of March 2021) Windows PC client, Android app, and Chrome web browser. The only exception occurred to iCloud Drive, for which we used its latest-version MacOS client, iOS app, and Safari web browser.

To examine a specific service with a specific access method, we first uploaded a 1 MB[2] highly compressed new file (f) to the cloud (so the resulting network traffic would be slightly larger than 1 MB). Next, on the user side, we appended a single byte to f to generate an updated file f'. Afterwards, we synchronized f' from the user to the cloud with the specific access method and meanwhile recorded the network traffic consumption. In this way, we can reveal if delta sync is applied by measuring the traffic consumption—if the traffic consumption was larger than 1 MB, the service did not adopt delta sync; otherwise (i.e., the traffic consumption was just tens of KBs), the service had implemented delta sync.

Based on the examination results listed in Table 2, we have the following observations: First, delta sync has been widely adopted in the majority of PC clients of cloud storage services. However, it has never been used by the mobile apps of any popular cloud storage services, though two academic services [13, 64] have implemented delta sync in their mobile apps and proved the efficacy. In fact, as the battery capacity and energy efficiency of mobile apps grow constantly, we expect delta sync to be widely adopted by mobile apps in the near future [37]. Finally, none of the

[2]We also experiment with files much larger than 1 MB in size, i.e., 10 MB and 100 MB, and got the same results.

Table 2. Delta Sync Support in Nine Cloud Storage Services

Service	PC Client	Mobile App	Web Browser
Dropbox	Yes	No	No
Google Drive	No	No	No
OneDrive	Yes	No	No
iCloud Drive	Yes	No	No
Box.com	No	No	No
SugarSync	Yes	No	No
Seafile [50]	Yes	No	No
QuickSync [13]	Yes	Yes	No
DeltaCFS [64]	Yes	Yes	No

studied cloud storage services supports *web-based* delta sync, despite web browsers constituting the most pervasive and OS-independent method for accessing Internet services.

To this end, we first introduce the general idea of web-based delta sync with basic motivation, preliminary design, and early-stage performance evaluation using limited workloads and metrics [61]. Nevertheless, in practice, we notice that leveraging conventional web technologies alone can not enable the efficient web-based delta sync without sacrificing the practicality (i.e., reliability, applicability, and usability). Fortunately, the emerging of WebAssembly [54] sheds light on how to enable web-based applications in both efficient and practical manners. Many useful WebAssembly-based browser applications have been proposed, such as interactive 3D visualization [21], resource accounting [24], audio and video software [36], and games [63]. To this end, we go a step further to introduce the WebAssembly-based delta sync with preliminary design, implementation, optimizations, and performance evaluation using limited workloads and metrics. In this article, our work is conducted based on Reference [62] while going beyond it in terms of techniques, evaluations, and presentations.

3 STUDY METHODOLOGY

In this section, to comprehensively understand the challenges and opportunities of supporting delta sync under current web frameworks, we design a variety of quantitative experiments for benchmarking web-based delta sync systems. We first present several metrics to assess the performance of web-based delta sync systems (Section 3.1) and then propose a variety of workloads to evaluate these metrics (Section 3.2). We also develop a tool named StagMeter to measure the client-side stagnation of these systems (Section 3.4). Last, we set up a Dropbox-like architecture (as an example of the state-of-the-art industrial cloud storage services) with the server, client, and network configurations detailed in Section 3.3. Apart from a new metric (*memory usage*) and a new workload (heavy workload), other parts of the methodology were proposed in our previous work [38, 62].

3.1 Metrics

We propose five metrics to measure the results of our experiment: (1) *sync efficiency,* which measures how quickly a file operation is synchronized to the cloud; (2) *computation overhead,* which explains the difference in sync efficiency of web-based delta sync systems; (3) *sync traffic,* which quantifies how much network traffic is consumed by each web-based delta sync system; (4) *memory usage,* which reveals the amount of memory space required at the server side of web-based delta sync systems; and (5) *service throughput*, which shows the scalability of each web-based delta sync system using standard VM server instances.

3.2 Workloads

To evaluate the performance of each web-based delta sync system under various practical usage scenarios, we generate *simple* (i.e., one-shot), *regular* (i.e., periodical), *intensive*, and *heavy* (i.e., large files) workloads. To generate simple workloads, we make random append, insert, and cut[3] operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, to 100 KB) against real-world files collected from real-world cloud storage services. The dataset is collected in our previous work and is publicly released [38], where the average file size is nearly 1 MB. One file is edited only once (the so-called "one-shot"), and it is then synchronized from the client side to the server side. For an insert or cut operation, when its edit size ≥1 KB, it is first dispersed into 1–20 *continuous sub-edits*[4] and then synchronized to the server like the simple workload.

Regular and intensive workloads are mainly employed to evaluate the service throughput of each system. To generate regular workloads, we still make a certain type of edit to a typical file, whose file size is ~5 MB, but the edit operation is executed every 10 seconds. To generate a practical intensive workload, we use a benchmark of over 8,755 pairs of source files taken from two successive releases (versions 4.5 and 4.6) of the Linux kernel source trees. The average size of the source files is 23 KB, and the file-edit locality is generally stronger than that in Figure 15 (as shown in Figure 1). Specifically, we first upload all the files of the old version to the server side in an FTP-like manner. Then, we synchronize all the files of the new version one-by-one to the server side using the target approaches. There is no time interval between two sequential file synchronizations.

The heavy workload is used to investigate the performance of web-based delta sync systems in large-file edit scenarios. To create heavy workloads, we generate large files of different sizes (ranging from 10 MB, 20 MB, 40 MB, 60 MB, 80 MB, to 100 MB) that mimic real-world files. Then, we also make random append, insert, and cut operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, 100 KB, to 1 MB) against these large files. We edit each file only once and synchronize them from the client side to the server side. For an insert or cut, when its edit size ≥1 KB, it is also first dispersed into 1–20 continuous sub-edits and then synchronized to the server as the simple workload does.

3.3 Experiment Setup

Using the abovementioned measurement metrics and workloads, we conduct a variety of comparative measurement studies of web-based delta sync systems. For each system, we set up a Dropbox-like system architecture by running the web service on a standard VM server instance (with a quad-core Intel Xeon CPU @2.5 GHz and 16 GB memory) rented from Aliyun ECS, and all file content is hosted on object storage rented from Aliyun OSS. The ECS VM server and OSS storage are located at the same data center, so there is no bottleneck between them. The client side of each system was executed in the Google Chrome browser (Windows version 56.0) running on a laptop with a quad-core Intel Core-i5 CPU @2.8 GHz, 16 GB memory, and an SSD disk. The server side and client side lie in different cities (i.e., Shanghai and Beijing) and different ISPs (i.e., China Unicom and CERNET), as depicted in Figure 2. The network RTT is ~30 ms, and the network bandwidth is ~100 Mbps. Therefore, the network bottleneck is kept minimal in our experiments so the major system bottleneck lies at the server and/or client sides. If the network condition becomes much worse, then the major system bottleneck might shift to the network connection.

[3]Here, "cut" means to remove some bytes from a file.
[4]A *continuous sub-edit* means that the sub-edit operation happens to continuous bytes in the file. More details are explained in Section 4.3.2, especially in Figures 14 and 15.

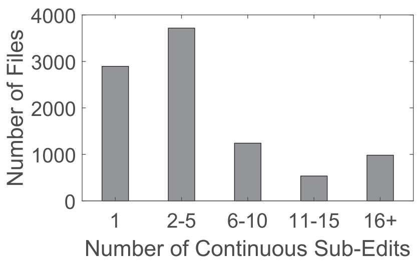


Fig. 1. File-edit locality in the source files of two successive Linux kernel releases (versions 4.5 and 4.6).

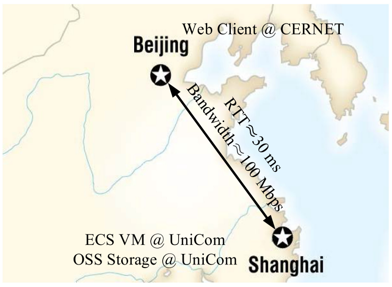


Fig. 2. Experiment setup in China.

3.4 Measuring Stagnation

In practice, the computation-intensive operations (e.g., chunk search and comparison) usually bring considerable computation overhead at the client side for a web-based delta sync system. The heavy CPU consumption causes web browsers to frequently stagnate and even hang. To quantitatively understand the stagnation of web browsers perceived by users, we develop the StagMeter tool to measure the stagnation time by automatically integrating a piece of JavaScript code into the web browser.[5] StagMeter periodically[6] prints the current timestamp on the concerned web page (e.g., the web page that executes delta sync). If the current timestamp (say, t) is successfully printed at the moment, then there is no stagnation; otherwise, there is stagnation and then the printing of the current timestamp will be postponed to $t' > t$. Therefore, the corresponding stagnation time is calculated as $t' - t$.

4 WEB-BASED DELTA SYNC

It is worth mentioning that this section presents contributions of our previous work [62]. In this section, we introduce a variety of web-based delta sync solutions in a step-by-step optimization manner. First, we propose the first workable solution named WebRsync that works as a baseline in Section 4.1. Through performance benchmarking, we find that there exist drawbacks of WebRsync. Then, we investigate several client-side optimizations to partially address the drawback of WebRsync in Section 4.2. Measurement results show that the drawback of WebRsync cannot be fundamentally addressed via only client-side optimizations. Thus, we present WebR2sync+, the preliminary practical solution for web-based delta sync, in Section 4.3. Although they are not practically acceptable solutions that satisfy both high sync efficiency and practicability (i.e., applicability, usability, and reliability), they indicate the challenges and optimization opportunities of supporting delta sync under current web frameworks.

4.1 WebRsync

WebRsync is the first workable implementation of web-based delta sync for cloud storage services. It is implemented in JavaScript based on HTML5 File APIs [45] and WebSocket. It follows the algorithm of `rsync` and thus keeps the same behavior as PC client-based approaches.

4.1.1 *Design and Implementation.* We design WebRsync by adapting the working procedure of `rsync` to the web browser scenario. As demonstrated in Figure 3, in WebRsync when a user edits a

[5]We can also directly use the native profiling tool of the Chrome browser to visualize the stagnation, whose results we found more complicated to interpret than those of StagMeter.
[6]By default, we set the period as 100 ms to simulate the minimum intervals of common web users' operations.

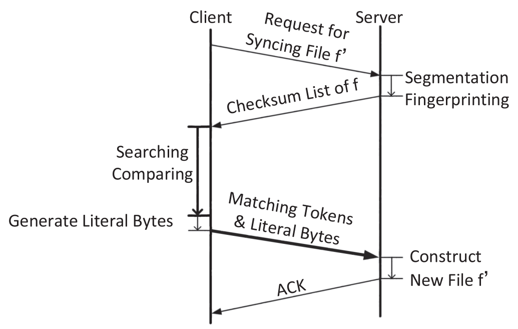


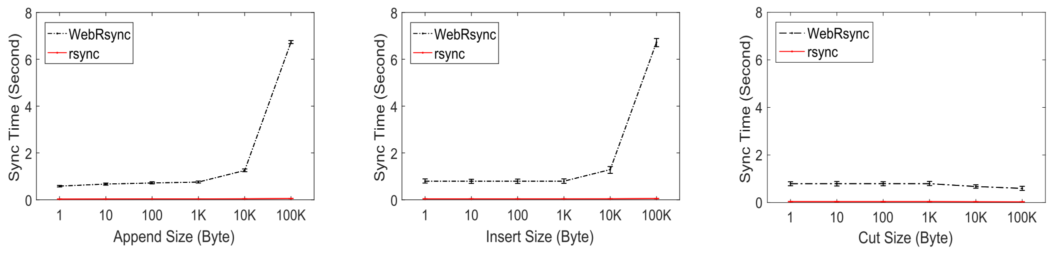
Fig. 3. Design flow chart of WebRsync.

file from f to f', the client instantly sends a request to the server for the file synchronization. On receiving the request, the server first executes fixed-size chunk *segmentation* and *fingerprinting* operations on f (which is available on the cloud side) and then returns a *checksum list* of f to the client. Except for the last chunk, each data chunk is typically 8 KB in size. Thus, when f is 1 MB in size, its checksum list contains 128 weak 32-bit checksums as well as 128 strong 128-bit MD5 checksums [49]. After that, based on the checksum list of f, the client first performs chunk *search* and *comparison* operations on f' and then generates both the *matching tokens* and *literal bytes.* Note that search and comparison operations are both conducted in a byte-by-byte manner on *rolling* checksums; in comparison, segmentation and fingerprinting operations are both conducted in a chunk-by-chunk manner so they incur much lower computation overhead. The matching tokens indicate the overlap between f and f', while the literal bytes represent the novel parts in f' relative to f. Both of them are sent to the server for constructing f'. Finally, the server returns an acknowledgment to the client to conclude the process.

We implement the client side of WebRsync based on the HTML5 File APIs [45] and the WebSocket protocol, using 1,500 lines of JavaScript code. Following the common practice to optimize the performance of JavaScript execution, we adopt the asm.js language [8] to first write the client side of WebRsync in C code and then compile it to JavaScript. The server side of WebRsync is developed based on the node.js framework, with 500 lines of node.js code and 600 lines of C code; its architecture follows the server architecture of Dropbox. Similar to Dropbox, the web service of WebRsync runs on a VM server rented from Aliyun ECS [5], and the file content is hosted on object storage rented from Aliyun OSS [6]. More details on the server, client, and network configurations have been described in Section 3.3 and Figure 2.

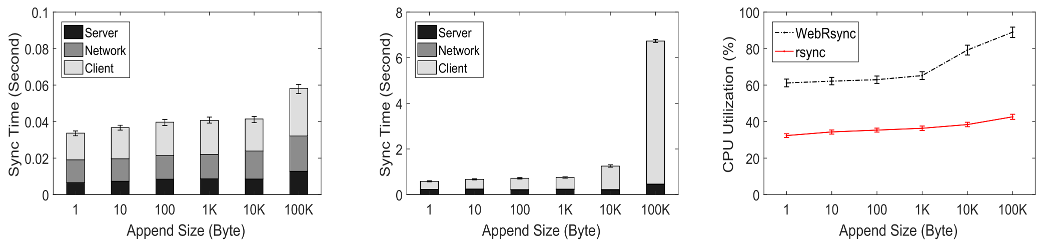
4.1.2 Performance Benchmarking. We first compare the performance of WebRsync and `rsync` under the simple workload to reveal the challenges and optimization opportunities of enabling the web-based delta sync. We perform random append, insert, and cut operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, to 100 KB) upon real-world files [38], as mentioned in Section 3.2. For each of the three different types of edit operations, we first measure its average sync time corresponding to each edit size and then decompose the average sync time into three stages: server, network, and client. Moreover, we measure its average CPU utilization on the client side corresponding to each edit size.

As shown in Figure 4, for each type of file edit operations the sync time of WebRsync is significantly longer than that of `rsync` (by 16–35 times). In other words, WebRsync is much slower than `rsync` on handling the same file edit. Among the three types of file edits, we notice that syncing a cut operation with WebRsync is always faster than syncing an append/insert operation (for the same edit size), especially when the edit size is relatively large (10 KB or 100 KB). This is because



(a) Append. (b) Insert. (c) Cut.

Fig. 4. Average sync time using WebRsync for various sizes of file edits (including append, insert, and cut) under a simple workload. The error bars show the minimum and maximum values at each point.



(a) Breakdown of the sync time of rsync. (b) Breakdown of the sync time of WebRsync. (c) Average client-side CPU utilization.

Fig. 5. Breakdown of the sync time of (a) rsync and (b) WebRsync for append operations, as well as the corresponding average client-side CPU utilizations. The situations for insert and cut operations are similar.

a cut operation reduces the length of a file while an append/insert operation increases the length of a file.

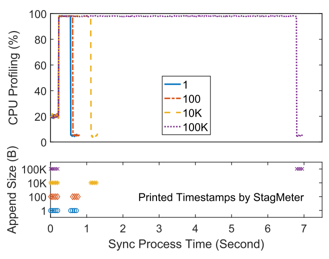
Furthermore, we decompose the sync time of rsync and WebRsync into three stages: at the client side, across the network, and at the server side, as depicted in Figures 5(a) and 5(b). For each type of file edit, around 40% of rsync's sync time is spent at the client side and around 35% is spent at the network side; in comparison, the vast majority (60%–92%) of WebRsync's sync time is spent at the client side, while less than 5% is spent at the network side. This indicates that the sync bottleneck of WebRsync is due to the inefficiency of the web browser's executing JavaScript code. Additionally, Figure 5(c) illustrates that the CPU utilization of each type of file edit in WebRsync is as nearly twice as that of rsync, because JavaScript programs consume more CPU resources.

The heavy CPU consumption causes web browsers to frequently stagnate and even hang. Using the aforementioned measurement tool in Section 3.4, StagMeter, we measure and visualize the stagnation of WebRsync (on handling the three types of file edits) in Figure 6. Note that StagMeter only attempts to print 10 timestamps for the first second. Therefore, spaces between consecutive timestamps represent stagnation, and larger spaces imply longer stagnation. As indicated in all the three subfigures, the stagnation is directly associated with high CPU utilizations.

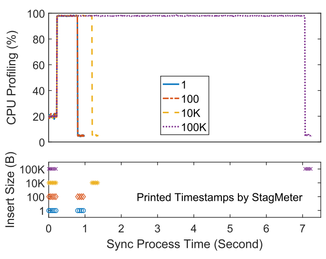
4.2 Native Extension, Parallelism, and Client-side Optimization of WebRsync

We investigate three approaches to partially address the drawback of WebRsync. For each approach, we first describe its working principle and then evaluate its performance using different types of file edits.

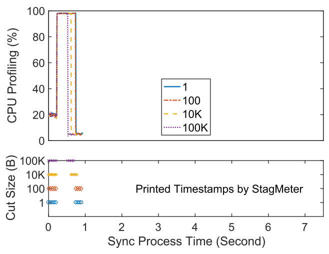
WebRsync-native. Given that the sync speed of WebRsync is much lower than that of the PC client-based delta sync solution (rsync), our first approach to optimizing WebRsync is to leverage



(a) Append.

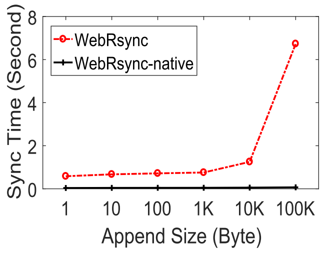


(b) Insert.

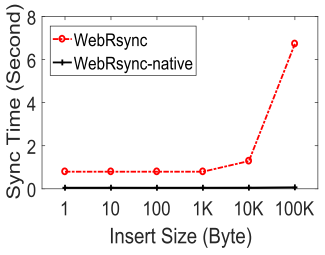


(c) Cut.

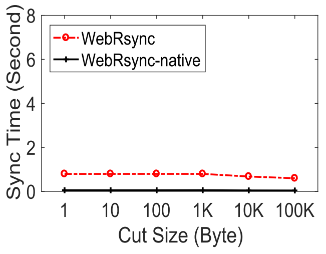
Fig. 6. Stagnation captured by StagMeter for different edit operations and the associated CPU utilizations. The stagnation time is illustrated by the discontinuation of the timestamp on the sync process time.



(a) Append.



(b) Insert.



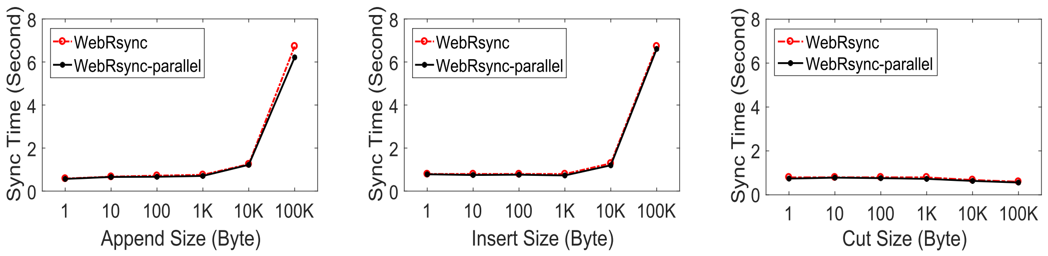
(c) Cut.

Fig. 7. Average sync time using WebRsync-native for various sizes of file edits under a simple workload.

the *native client* [25] for web browsers. The native client is a sandbox for efficiently and securely executing compiled C/C++ code in a web browser and has been supported by all mainstream web browsers. In our implementation, we use the Chrome native client to accelerate the execution of WebRsync on the Chrome browser. We first use HTML5 and JavaScript to compose the webpage interface, through which a user can select a local file to synchronize (to the cloud). Then, the path of the selected local file is sent to our developed native client (written in C++). Afterwards, the native client reads the file content and synchronizes it to the cloud in a similar way as `rsync`. When the sync process finishes, the native client returns an acknowledgment message to the webpage interface, which then shows the user the success of the delta sync operation.

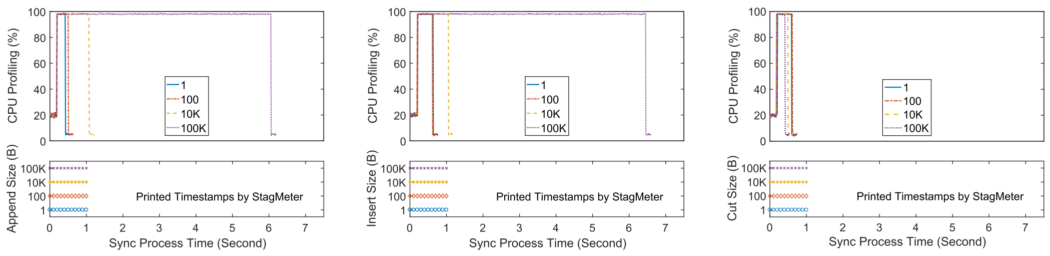
Figure 7 depicts the performance of WebRsync-native, in comparison to the performance of original WebRsync. Obviously, WebRsync-native significantly reduces the sync time of WebRsync, in fact close to the sync time of `rsync`. Accordingly, the CPU utilization is decreased and the stagnation of the Chrome browser is fully avoided. Nevertheless, using the native client requires the user to download and install extra plug-in components for the web browser, which essentially impairs the usability and pervasiveness of WebRsync-native.

WebRsync-parallel. Our second approach is to use HTML5 *web workers* [15] for parallelism or threading. Generally speaking, when executing JavaScript code in a webpage, the webpage becomes unresponsive until the execution is finished—this is why WebRsync would lead to frequent stagnation and even hanging of the web browser. To address this problem, a web worker is a JavaScript program that runs in the background, independently of other JavaScript programs in the same webpage. When we apply it to WebRsync, the original single JavaScript program is divided to multiple JavaScript programs that work in parallel. Although this approach can hardly reduce the total sync time (as indicated in Figure 8) or the CPU utilizations (as shown in Figure 9,



(a) Append. (b) Insert. (c) Cut.

Fig. 8. Average sync time using WebRsync-parallel for various sizes of file edits under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Fig. 9. Although WebRsync-parallel is unable to reduce the CPU utilizations (relative to WebRsync), it can fully avoid stagnation for the Chrome web browser by utilizing HTML5 web workers.

the upper part), it can fully avoid stagnation for the Chrome browser (as shown in Figure 9, the lower part).

WebRsync+. Later, in Section 4.3, we describe in detail how we exploit users' file-edit locality and lightweight hash algorithms to reduce server-side computation overhead. As a matter of fact, the two-fold optimizations can also be applied to the client side. Thereby, we implement the two optimization mechanisms at the client side of WebRsync by translating them from C++ to JavaScript, and the resulting solution is referred to as WebRsync+. As illustrated in Figure 10, WebRsync+ significantly outpaces WebRsync in terms of sync time owing to the client-side optimizations, which is basically within our expectation. Further, we decompose the sync time of WebRsync+ into three stages: at the client side, across the network, and at the server side, as depicted in Figure 11. Comparing Figure 11 with Figure 5(b) (breakdown of the sync time of WebRsync into three stages), we find that the client-side time cost of WebRsync+ is remarkably reduced, thanks to the two optimization mechanisms. However, WebRsync+ cannot fully avoid stagnation for web browsers; instead, it can only alleviate the stagnation compared to WebRsync.

Summary. With the above three-fold efforts, we conclude that the drawback of WebRsync cannot be fundamentally addressed via solely client-side optimizations. That is to say, we may need more comprehensive solutions where the server side is also involved.

4.3 WebR2sync+: The Preliminary Practical Solution

This subsection presents WebR2sync+, the preliminary practical solution for web-based delta sync. The high efficiency of WebR2sync+ is attributed to multi-fold endeavors at both client and server sides. We first present the basic solution, WebR2sync, which improves WebRsync (Section 4.3.1), and then describe the server-side optimizations for mitigating the computation overhead

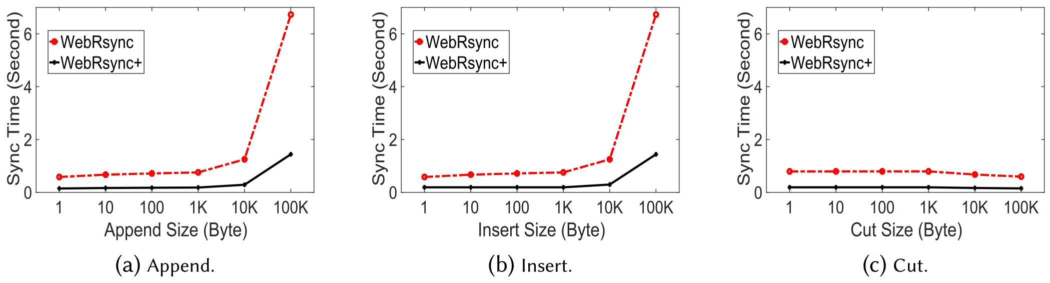


Fig. 10. Average sync time using WebRsync+ for various sizes of file edits under a simple workload (in comparison to WebRsync).

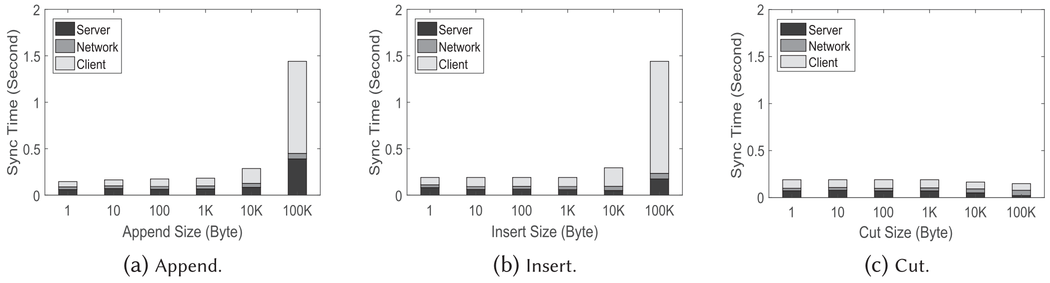


Fig. 11. Breakdown of the sync time of WebRsync+ (shown in Figure 10) for different types of edit operations.

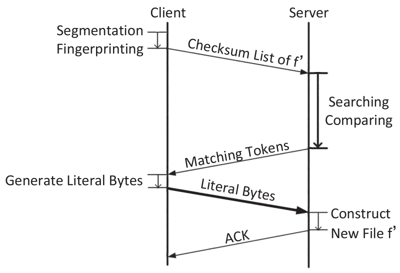


Fig. 12. Design flow chart of WebR2sync.

(Section 4.3.2). The solution that combines both WebR2sync with the server-side optimizations is referred to as WebR2sync+, and we also discuss its limitations in detail in Section 4.3.3.

4.3.1 WebR2sync. As depicted in Figure 12, to address the overload issue, WebR2sync reverses the process of WebRsync (cf. Figure 3) by moving the computation-intensive search and comparison operations to the server side; meanwhile, it shifts the lightweight segmentation and fingerprinting operations to the client side. Compared with the workflow of conventional web-based delta sync, in WebRsync, the checksum list of f' is generated by the client and the matching tokens are generated by the server, while the literal bytes are still generated by the client. Note that this allows us to implement the search and comparison operations in C rather than in JavaScript at the server side. Therefore, WebR2sync can not only avoid stagnation for the web client, but also effectively shorten the duration of the whole delta sync process.

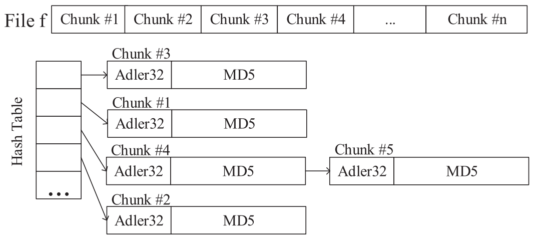


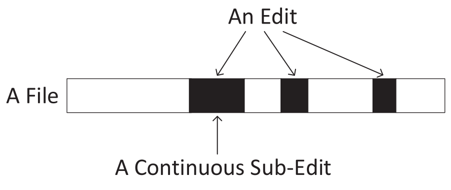
Fig. 13. The three-level chunk searching scheme used by rsync and WebR2sync.

4.3.2 Server-side Optimizations. While WebR2sync significantly cuts the computation burden on the web client, it brings considerable computation overhead to the server side. To this end, we make additional two-fold efforts to optimize the server-side computation overhead.

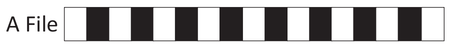
Exploiting the locality of file edits in chunk search. When the server receives a checksum list from the client, WebR2sync uses a three-level chunk searching scheme to figure out matched chunks between f and f', as shown in Figure 13 (which follows the three-level chunk searching scheme of `rsync` [49]). Specifically, in the checksum list of f' there is a 32-bit weak rolling checksum (calculated by the Adler32 algorithm [14]) and a 128-bit strong MD5 checksum for each data chunk in f'. When this checksum list is sent to the server, the server leverages an additional *(rolling checksum) hash table* whose every entry is a 16-bit hash code of the 32-bit rolling checksum [49]. The checksum list is then sorted according to the 16-bit hash code of the 32-bit rolling checksums. Note that a 16-bit hash code can point to multiple rolling and MD5 checksums. Thereby, to find each matched chunk between f and f', the three-level chunk searching scheme always goes from the 16-bit hash code to the 32-bit rolling checksum and further to the 128-bit MD5 checksum.

The three-level chunk searching scheme can effectively minimize the computation overhead for general file-edit patterns, particularly random edits to a file. However, it has been observed that real-world file edits typically follow a local pattern rather than a general (random) pattern, which has been exploited to accelerate file compression and deduplication [42, 58–60]. To exemplify this observation in a quantitative manner, we analyze two real-world fine-grained file-editing traces with respect to Microsoft Word and Tencent WeChat collected by Zhang et al. [64]. The traces are fine-grained, since they leveraged a loopback user-space file system (Dokan [16] for Windows) to record not only the detailed information (e.g., edit type, edit offset, and edit length) of users' file operations but also the content of the updated data. In each trace, a user made several *continuous sub-edits* to a file and then did a save operation, and this process was repeated many times. Here, a continuous sub-edit means that the sub-edit operation happens to continuous bytes in the file, as demonstrated in Figure 14. Our analysis results of the above-mentioned traces (Microsoft Word and Tencent WeChat), in Figure 15, show that in nearly a half (46%) of cases a user saved 1–5 continuous sub-edits, thus indicating fine locality. Besides, in over one-third (35%) of cases a user saved 6–10 continuous sub-edits, which still implies sound locality. However, in only a minority (5%) of cases a user saved more than 16 continuous sub-edits, which means undesirable locality.

In addition, to explore the universality of the file-edit locality characteristics, we have also conducted experiments on multimedia files (i.e., images and videos). However, our experimental results indicate that for images and videos, even minor file edits can lead to full file synchronization, thus losing the advantages of delta sync. After further analysis of this phenomenon, we find that it is due to the encoding and decoding characteristics of images and videos. Thus, we adopt text files of different sizes as our study's objectives, rather than multimedia files (i.e., images and files).



(a) An edit consists of several continuous sub-edits.



(b) The worst case of a file edit in terms of locality.

Fig. 14. An example of continuous sub-edits due to the locality of file edits: (a) the relationship between a file edit and its constituent continuous sub-edits; (b) the worst-case scenario in terms of locality.

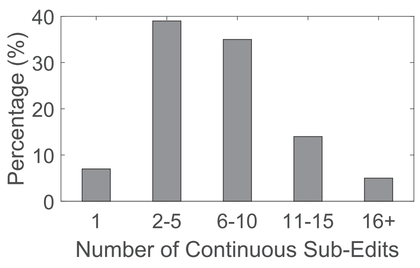


Fig. 15. A real-world example of file-edit locality. The number of continuous sub-edits is highly clustered.

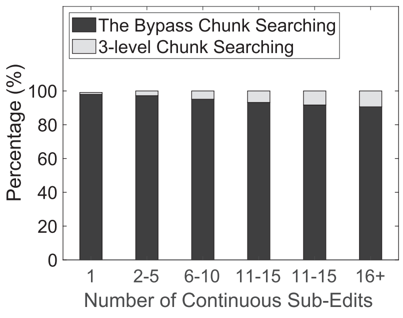


Fig. 16. The proportion of the three-level chunk searching and the bypass chunk searching used in synchronizing files with different sub-edits.

The locality of real-world file edits offers us an opportunity to bypass a considerable portion of (unnecessary) chunk search operations. In essence, given that edits to a file are typically local, when we find that the ith chunk of f' matches the jth chunk of f, the $(i + 1)$-th chunk of f' is highly likely to match the $(j + 1)$-th chunk of f. Therefore, we "simplify" the three-level chunk searching scheme by directly comparing the MD5 checksums of the $(i + 1)$-th chunk of f' and the $(j + 1)$-th chunk of f. If the two chunks are identical, then we simply move forward to the next chunk and call this process a bypass chunk searching; otherwise, we return to the regular three-level chunk searching scheme. In other words, file edits with fine locality (i.e., those having a small number of sub-edits) can leave more continuous blocks unmodified, which enables us to perform more lightweight bypass chunk searching instead of the heavy three-level chunk searching. To explore how often the three-level chunk searching scheme versus the bypass chunk searching scheme is used, we conduct a measurement study based on the aforementioned Microsoft Word and Tencent WeChat traces [64]. Figure 16 illustrates that for files with different sub-edits, the use of the bypass searching scheme accounts for more than 90%, while the three-level chunk searching scheme accounts for less than 10%. In other words, we can bypass most (up to ~90%) chunk search operations in real usage scenarios by exploiting the locality of users' file edits. In particular, the frequency of using the bypass searching scheme increases with the decrease of the number of sub-edit operations, which is within our expectation, since more sub-edits usually mean undesirable file-edit locality.

Table 3. A Comparison of Candidate Pseudorandom Hash Functions in Terms of Collision Probability (on 64-bit Hash Values) and Computation Overhead (Cycles per Byte)

Hash Function	Collision Probability	Cyclesper Byte
MD5	Very Low ($<10^{-9}$)	5.58
Murmur3	High ($\approx 1.05 \times 10^{-4}$)	0.33
CityHash	High ($\approx 1.03 \times 10^{-4}$)	0.23
FNV	High ($\approx 1.09 \times 10^{-4}$)	1.75
Spooky	High ($\approx 9.92 \times 10^{-5}$)	0.14
SipHash	Low ($<10^{-6}$)	1.13

Replacing MD5 with SipHash in chunk comparison. By exploiting the locality of users' file edits as above, we manage to bypass most chunk search operations. After that, we notice that the majority of server-side computation overhead is attributed to the calculations of MD5 checksums. MD5 was initially designed as a cryptographic hash function for generating secure and low-collision hash codes [48], which makes it computationally expensive. Thus, we wonder whether MD5 can be replaced with a lightweight pseudorandom hash function in chunk comparison with no system performance degradation.

Quite a few pseudorandom hash functions can satisfy our goal, such as Spooky [11], FNV [35], CityHash [23], SipHash [9], and Murmur3 [3]. Among them, some are very lightweight but vulnerable to collisions. For example, the computation overhead of MD5 is around 5 to 6 cycles per byte [53], while the computation overhead of CityHash is merely 0.23 cycle per byte [4], but the collision probability of CityHash is quite high. However, some pseudorandom hash functions have extremely low collision probability but are a bit slow. As listed in Table 3, SipHash seems to be a sweet spot—its computation overhead is about 1.13 cycles per byte, and its collision probability is acceptably low. By replacing MD5 with SipHash in our web-based delta sync solution, we manage to reduce the computation complexity of chunk comparison by nearly 5 times.

Although the collision probability of SipHash is relatively low, it is still three orders of magnitude higher than that of MD5 (as shown in Table 3). Thus, the adoption of Siphash also brings with some service reliability problems, which we will analyze in detail in Section 4.3.3. As a fail-safe mechanism, we make a lightweight full-content hash checking (using the Spooky algorithm) in the end of a file synchronization to deal with possible collisions in SipHash chunk fingerprinting. We select the Spooky algorithm because it works the fastest among all the candidate pseudorandom hash algorithms (as listed in Table 3). If the full-content hash checking fails for the synchronization of a file, then we will roll back and re-sync the file with the original MD5 chunk fingerprinting.

4.3.3 WebR2sync+. The integration of WebR2sync and the server-side optimization produces WebR2sync+. The client side of WebR2sync+ is implemented based on the HTML5 File APIs, the WebSocket protocol, an open-source implementation of SipHash-2-4 [22], and an open-source implementation of SpookyHash [29]. In total, it is written in 1,700 lines of JavaScript code. The server side of WebR2sync+ is developed based on the node.js framework and a series of C processing modules. The former (written in 500 lines of node.js code) handles the user requests, and the latter (written in 1,000 lines of C code) embodies the reverse delta sync process together with the server-side optimizations.

As illustrated in Figure 17, WebR2sync+ achieves the best performance in terms of sync time among all web-based delta sync solutions owing to our proposed optimizations. Nevertheless, although WebR2sync+ has achieved satisfactory results close to `rsync`, it still has two significant drawbacks. First, although using Siphash can effectively reduce the overhead, the collision probability of Siphash is three orders of magnitude higher than that of MD5, according to our

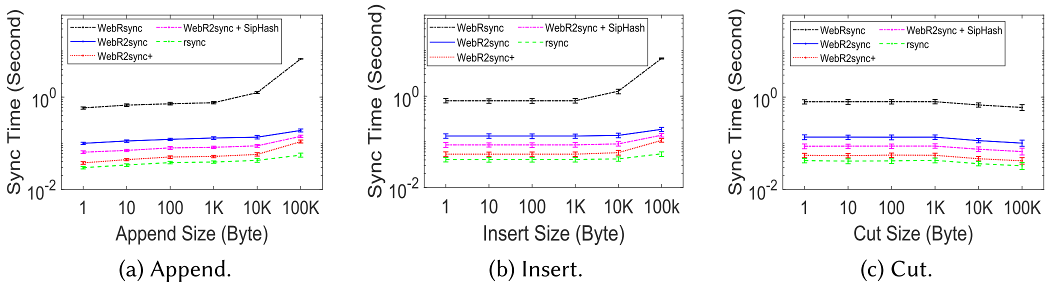


Fig. 17. Average sync time of different web-based delta sync approaches for various sizes of file edits under a simple workload.

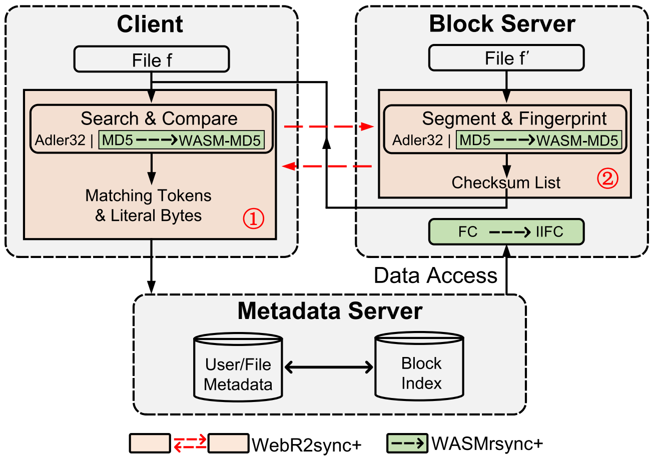


Fig. 18. Architectural overview of a typical cloud storage system and the efforts needed to apply WebR2sync+ and WASMrsync+. WebR2sync+ requires reversing the critical workflow between the cilent and the server, while WASMrsync+ only involves minor in-place component replacements (e.g., replacing MD5 with WASM-MD5).

measurement results (as listed in Table 3). Once a hash collision occurs during the synchronization process, two file chunks with different content may be recognized as the same file chunks, thus a synchronization error occurs. And the efficiency of WebR2sync+ is partly due to replacing the heavyweight MD5 with the lightweight Siphash. Thus, WebR2sync+ runs the risk of making more synchronization errors than other web-based delta sync solutions using MD5, which greatly damages the system reliability of WebR2sync+. To solve this issue, we have to adopt the aforementioned full-content hash checking and rollback strategy to ensure correct synchronization of a file. Second, to address the overload issue of the client side, WebR2sync+ reverses the process of `rsync` by moving the computation-intensive search and comparison operations to the server side; meanwhile, it shifts the segmentation and fingerprinting operations to the client side. That is to say, WebR2sync+ breaks the original workflow of rsync due to the reverse scheme.

Figure 18 depicts the architectural overview of mainstream cloud storage systems (e.g., Dropbox), which follows the original workflow of `rsync` (Figure 3). Each client maintains a connection to a metadata server. The metadata server authenticates each user and stores metadata about the user's files, including the list of the user's files, their sizes and attributes, and pointers to where the files can be found on the block server. Usually, block servers are built on top of commercial storage services such as Amazon S3. If mainstream cloud storage systems want to adopt WebR2sync+, then they need to spend considerable development cost to refactor their architectures by reversing the corresponding computing operations between the client side (component ①) and the server side (component ②) and modifying interfaces between them, which significantly impairs the industrial applicability of WebR2sync+. In a nutshell, WebR2sync+ is an efficient but not practical web-based delta sync solution due to the sacrifice of reliability and applicability. Thus, we next explore feasible web-based delta sync schemes that can preserve both reliability and industrial applicability (WASMrsync and WASMrsync+). For clear comparisons, we also present the corresponding efforts required for existing cloud storage systems to adopt WASMrsync+ in Figure 18 and will detail them in Section 6.

5 WEBASSEMBLY-BASED DELTA SYNC: THE FIRST ENDEAVOR

Considering the above two drawbacks of WebR2sync+, we need to rethink how to reduce the client-side overhead caused by MD5 checksums computation without replacing the safer hashing algorithm MD5 and breaking the original workflow of `rsync` (cf. Figure 3). Fortunately, we notice an emerging technique (or web language) called ***WebAssembly* (WASM)** [54] has been attracting increasing attention in the design of web applications.

5.1 WebAssembly

WASM [54] is a portable binary instruction format (or web programming language) that is efficient in both encoding size and load time, enabling deployment on the web for client and server applications. In general, the most important characteristics of WASM include the following four aspects: (1) Efficient and fast: WASM is designed to be encoded in a size- and load-time-efficient binary format and aims to be executed at native speed by taking advantage of common hardware capabilities available on a wide range of platforms; (2) Safe: WASM describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines. When embedded in the web, WASM will enforce the same-origin and permissions security policies of the browser. (3) Open and debuggable: WASM is designed to be pretty printed in a textual format that will be used when viewing the source of WASM modules on the web. (4) Easy-to-apply: WASM modules can call into and out of the JavaScript context and access browser functionality through the same Web APIs accessible from JavaScript. Benefiting from the above-mentioned characteristics, WASM can enable web-based applications to enjoy near-native runtime speed while without requiring significant cloud-size or client-side changes like the *native client* [25] in a user-friendly and easy-to-apply manner.

5.2 WASMrsync

This subsection introduces the first workable implementation of WebAssembly-based delta sync for cloud storage services. Considering the near-native runtime speed and minimal client-side modification requirements of WASM, we directly adopt WASM in WebRsync with the solution referred to as WASMrsync.

To enable cloud storage systems to adopt WASMrsync with the minor refactoring of their architectures, WASMrsync retains the original workflow of `rsync` (cf. Figure 3). Observing the majority of client-side computation overhead is attributed to the calculation of MD5 checksums in WebRsync (cf. Section 4), WASMrsync improves on WebRsync by directly leveraging WASM to carry out this operation.

Specifically, following the guidelines of official documents for leveraging WASM [43], we first reimplement the MD5 checksum calculation in C code. Then, we compile the C source code to a WASM module by using Emscripten [7], a complete open-source compiler toolchain for

```
// The regular expressions follow the JavaScript style
(async () => {
  // Download the wasm module
  const response = await fetch('md5.wasm');
  const buffer = await response.arrayBuffer();
  // Compile the wasm module
  const module = await WebAssembly.compile(buffer);
  // Instantiate the wasm module
  const instance = await WebAssembly.instantiate(module);
  const result = instance.export.md5(str);
})();
```

Fig. 19. A simplified example of running WASM in a download-compile-instantiate manner in JavaScript.

compiling C/C++ source code to a WASM module, and end up with a binary module file—*md5.wasm*. According to practical advice [55], to support the future plans to allow WASM modules to be loaded, WASM must currently be loaded and compiled by JavaScript. To this end, we follow these three steps: (1) first, we convert the bytes of *md5.wasm* into a JavaScript datatype *ArrayBuffer*; (2) then, we compile the bytes using the asynchronous function WebAssembly.compile, which returns a *Promise Object* that resolves to a WebAssembly.Module; (3) at last, we instantiate the module by running a new WASM API WebAssembly.instantiate passing in a module and any imports requested by the module. Figure 19 shows a simplified code snippet that does the complete download-compile-instantiate dance in WASMrsync.

Note that the component of leveraging WASM to calculate MD5 is packaged as a dynamic library for mainstream cloud storage systems to apply directly. Therefore, mainstream cloud storage systems do not need to refactor their architectures, but only need to replace the original functional modules with our encapsulated dynamic libraries, which essentially improve the applicability of WASMrsync. With the above effort, we end up using the HTML5 APIs and JavaScript "glue" code to integrate the compiled and loaded *md5.wasm* into our original system. To find out how much computation time of MD5 checksums can be reduced through WASM, we perform the calculation of MD5 checksums in different data chunk sizes (ranging from 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, to 512 KB) by using WASM and JavaScript upon real-world files collected from real-world cloud storage services [38]. As illustrated in Figure 20, WASM performs the calculation of MD5 checksums almost 1.8 to 5.5 times faster than JavaScript, and the acceleration ratio increases as the data chunk size increases. Nevertheless, considering that increasing the data chunk size makes delta sync become coarse-grained and thus requires more sync traffic, we empirically set the data chunk size to 64 KB to achieve a tradeoff between sync time and sync traffic. We next compare the performance of WASMrsync, WebRsync, and WebR2sync+ under the simple workload to validate the effectiveness of this optimization mechanism. As epitomized in Figure 21, WASMrsync stays between WebRsync and WebR2sync+ in terms of sync time, which is basically within our expectations. More specifically, WASMrsync can outpace WebRsync by 3 to 5 times for every different type of operation.

Although WASMrsync can significantly mitigate the computation burden at the web client by leveraging WASM to conduct the most expensive operation—the calculation of MD5 checksums—there still exists a performance gap between WASMrsync and WebR2sync+, which is shown in Figure 21. To explore the performance improvement opportunities for WASMrsync, we decompose

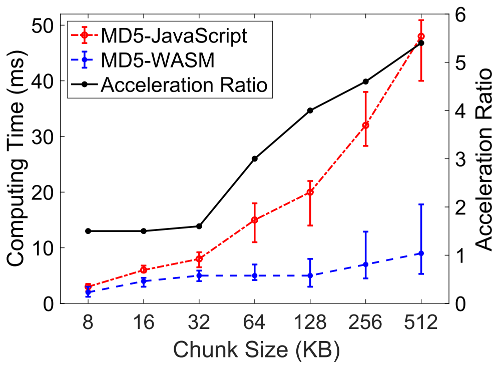
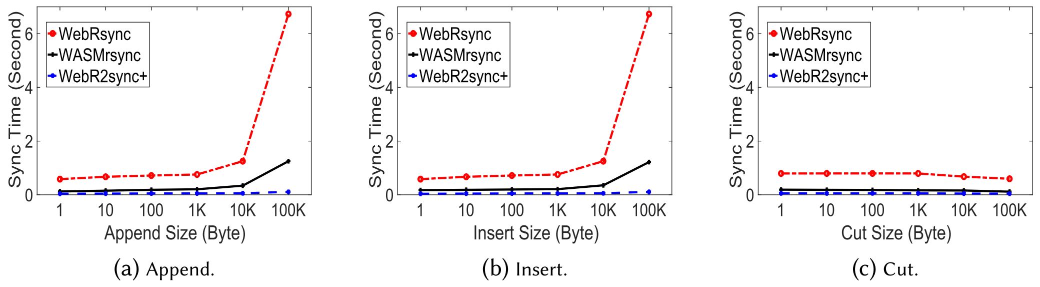
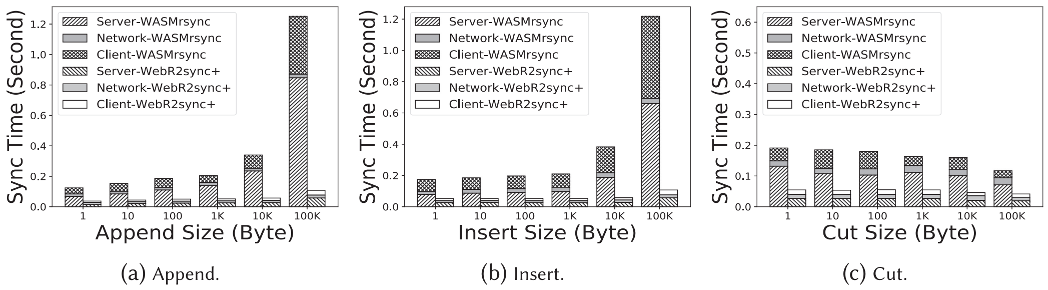


Fig. 20. Average computing time of MD5 by using WASM and JavaScript and the corresponding acceleration ratio for various sizes of data chunks.



(a) Append. (b) Insert. (c) Cut.

Fig. 21. Average sync time using WASMrsync for various sizes of file edits under a simple workload (in comparison to WebRsync and WebR2sync+).



(a) Append. (b) Insert. (c) Cut.

Fig. 22. Breakdown of the sync time of WASMrsync and WebR2sync+ for different types of edit operations.

the sync time of WASMrsync and WebR2sync+ into three stages: at the client side, across the network, and at the server side. Our analysis results, in Figure 22, show that for each type of file edit operation, the client-side and server-side sync time of WASMrsync is significantly longer than that of WebR2sync+ (by 4 to 6 and 5 to 8 times, respectively).

6 WASMRSYNC+: WEBASSEMBLY-BASED DELTA SYNC MADE EFFICIENT AND PRACTICAL

In this section, we present WASMrsync+, the resulting WebAssembly-based practical solution of supporting delta sync under current web frameworks. As discussed in Section 5, the performance gap between WASMrsync and WebR2sync+ reveals the need for a more comprehensive solution that requires optimization on both the client and server sides. Thus, to bridge this gap, we make additional two-fold efforts to improve the client-side sync efficiency and devise a mechanism to mitigate the performance bottleneck at the server side.

6.1 Client-side Optimizations

Devising strategic sync-async code decoupling at the client. By inspecting the execution flow of WASMrsync, we find that the quasi-asynchronous nature of WASM in rsync has led to considerable performance degradation. As demonstrated in Figure 3, when the client receives the checksum list of f from the server, the client first performs chunk *search* and *comparison* operations on f' in a byte-by-byte manner on checksums (including rolling and MD5 checksums) and then generates both the *matching tokens* and *literal bytes*. Inevitably, this manner requires us to implement the chunk *search* and *comparison* operations at the client side in a synchronous way.

However, to avoid blocking the main thread that may make the web page unresponsive, WASM APIs are designed to be asynchronous [55], e.g., WebAssembly.compile introduced in Section 5. Thus, we have to implement WASMrsync with a quasi-asynchronous strategy, i.e., to implement the asynchronous WASM API invocations in a synchronous manner using the await/async mechanism of JavaScript. Unfortunately, there are overheads associated with running the await/async method: (1) the current execution context has to be captured; (2) there is a thread transition; (3) and a state machine is built through which the code runs. Worse still, this await/async property of a WASM API invocation is propagated to the wrapping function of the invocation, which introduces unnecessary overhead for executing even non-WASM code in the wrapping function and thus severely increases the overall sync time.

We practically overcome this shortcoming through strategic sync-async code decoupling that extracts the synchronous non-WASM code from asynchronous modules to avoid unnecessary await-/async overhead. Specifically, we re-examine and modify the system implementation to avoid using async/await for very short methods and non-WASM exported functions, or having await statements in tight loops (run the whole loop asynchronously instead). And we change the multi-layer nested callbacks to the single-layer nested callbacks to minimize the workload caused by propagating the aforementioned await/async nature.

Adopting stream compilation in loading WASM modules at the client. As shown in Figure 19, when leveraging WASM, developers usually need to download a WASM module, compile it, instantiate it, and then use the exported module in JavaScript. For consistency and for the sake of keeping the main thread free, every operation in the code snippet is designed to be asynchronous. The asynchronous property of WASM APIs offers us an opportunity to process these operations in parallel for improving the overall loading efficiency. Hence, we manage to adopt *streaming compilation* [44] in loading WASM modules to meet this parallelization. *Streaming compilation* is an emerging web-accelerating technology that allows code to be downloaded while the browser compiles it.

We embody this adoption by replacing WebAssembly.compile and WebAssembly.instantiate with the state-of-the-art WASM API—WebAssembly.instantiateStreaming. This WASM API can compile and instantiate a WASM module directly from a streamed underlying source in one go. Specifically, by adopting stream compilation, we can compile the *md5.wasm* module already while we are still downloading the module bytes. Note that this API has been supported by V8 (JavaScript engine) [2] and Chrome 61 [1].

To find out the client-side optimizations' real impacts on the performance of the final product, we conduct extensive experimental evaluations in Section 6.3 in terms of our proposed metrics. The experimental results (Section 6.3) show that the client-side optimizations have the most significant effect on reducing sync time of the final solution.

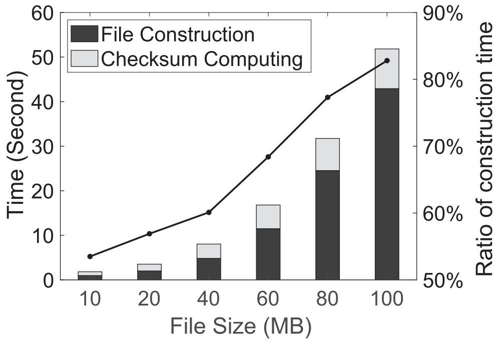


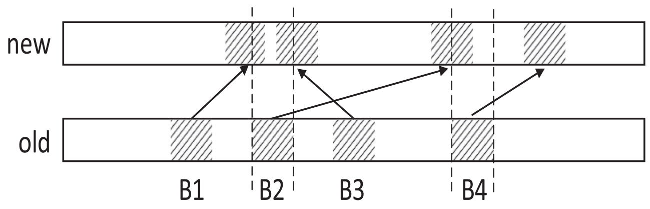
Fig. 23. Breakdown of the server-side sync time of WASMrsync and the ratio of construction time to the total server-side sync time for different large files.

6.2 Server-side Optimizations

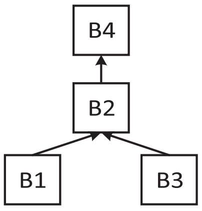
Designing Informed In-place File Construction mechanism at the server. By adopting the client-side optimizations as above, we manage to make WASM more compatible with web-based delta sync to improve the sync efficiency. After that, when handling large files (≥10 MB), we additionally notice that WASMrsync+ suffers from the time- and memory-consuming file construction at the server side. To quantify the proportion of file construction time in the entire server-side sync time, we perform random append, insert, and cut operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, 100 KB, 1 MB) into different large files (ranging from 10 MB, 20 MB, 40 MB, 60 MB, 80 MB, to 100 MB). These large files are generated to simulate real-world files. For each large file, we take the average sync time of appending/inserting/cutting different edit sizes as the final results. According to our measurement results (as exemplified in Figure 23), for such files, the construction time usually accounts for about 54%–83% of the entire sync time, and the proportion increases with the file size.

In particular, WASMrsync still follows the algorithm of `rsync` and may inherit some of the inherent defects of `rsync`. Thus, to demystify this phenomenon, we carefully investigate the internals of rsync and find the root cause is that the server side adopts the conventional **In-situ Separate Memory Allocation (ISMA)** mechanism to construct the updated file f'. In detail, when parsing the "patch" of every newly submitted file chunk, the server needs to (1) reallocate a memory region that is able to contain both the previous parsed chunk (stored in another memory region) and the new chunk, (2) copy the parsed chunk data from its memory region to the reallocated one, and (3) append the new chunk data. This means that the server side needs to perform excessive memory allocation/copy operations and maintain two memory spaces (M_{new} and M_{old}) when constructing an updated file. For example, when the chunk size is set to 64 KB, inserting 1 MB of data into a 100 MB file requires the server side to maintain a total memory space of 201 MB (100 MB for old, 101 MB for new) and perform about 3,156 ($2 \times \frac{101MB}{64KB}$) memory allocation/copy operations. For space-constrained servers, maintaining two memory spaces also significantly limits the number of concurrent clients supported by them. Further, we note that ISMA is largely associated with a lack of information about the updated file (i.e., the size of the updated file and the actions to chunks).

Based on the above-mentioned observations and investigations, to reduce the overhead introduced by ISMA, we devise the **Informed In-place File Construction (IIFC)** mechanism and deploy this strategy in WASMrsync. The objectives of the IIFC mechanism include: (1) performing a single incremental memory allocation to prevent the excessive time-consuming memory allocation/copy operations; and (2) reusing the memory space already occupied by the old file to eliminate the need for additional memory space when constructing the updated file. To meet the



(a) Example.



(b) Dependency graph.

Fig. 24. An example of the dependency relationship of chunks and the associated dependency graph.

two optimization objectives, we first modify the client to send the size of the updated file to inform the server in advance. Thus, the server can conduct a single incremental memory allocation based on the memory space occupied by the old file according to the informed size of the updated file. Then, we encode the actions to matched and unmatched file chunks as *Copy* and *Add* according to the chunk searching and comparing results, respectively, and send the encoding information to the server along with the literal bytes. Based on the encoding information and the literal bytes, the server copies the unmatched data chunks from their original memory regions to the new ones and adds the changed data chunks to the new memory regions.

It is extremely challenging to perform the In-place file construction, because we must account for hazards that arise when conducting a *Copy* action. *Copy* not only reads and copies a chunk of a file, but also overwrites some bytes that may exist in other data chunks. The overwritten chunks cannot be used in future *Copy* actions, because they no longer contain the original data. Figure 24(a) plots a simple example of file synchronization. The matched chunks will be moved from their original location (bottom) to their new location (top). To ensure the correctness of sync results, $B2$ must be completed before $B1$ and $B3$, because the destination of these chunks will corrupt the source for $B2$. Since the new $B2$ occupies the original region of $B4$, $B4$ needs to be copied before $B2$. In the abstract, we can get the associated dependency graph shown in Figure 24(b).

To undo the hazards and inspired by Figure 24, we modify the workflow of WASMrsync and embody the IIFC mechanism by taking the following steps: First, after encoding the actions to chunks of the updated file (*Copy* or *Add*) according to the chunk searching and comparing results, the client buffers *Copy* and *Add* actions in memory. For matched chunks, we need to buffer the source offset and the target offset of them. Unmatched Chunks require an extra field for the length of chunk data. Thus, we can get the original chunk dependency graph, but there may be some dependency cycles with regard to *Copy* actions.

Second, to ensure the correctness of sync results, we need to topologically sort the original dependency graph and break cycles as they are detected. To this end, we devise a DFS-based algorithm. The algorithm constructs a directed graph (denoted as $G = (N, E)$) in which edges $E = (e_1, e_2, e_3, \ldots)$ represent ordering constraints among *Copy* actions and nodes $N = (n_1, n_2, n_3, \ldots)$ indicate matched chunks. The unmatched chunks are self-describing and require no data from the old region. Therefore, *Add* actions need no reordering. A topological sort of the graph determines an execution order for processing *Copy* actions at the server. When a total topological ordering proves impossible because of cycles in the graph, we randomly convert the action to one node in the cycle from *Copy* to *Add* and add the corresponding chunk data explicitly. In the end, we can get a new chunk dependency tree with no cycles. After the above steps, the client side delivers the new chunk dependency graph, followed by literal bytes to the server side. Finally, the server applies *Copy* or *Add* actions when receiving the data. The server seeks the given offset to either copy a matched chunk from its original memory region or insert the unmatched data chunk.

It is worth mentioning that when handling a file larger than the server's memory with the IIFC, we devise a sequential processing strategy. Specifically, once one chunk dependency tree (smaller than memory) is formed, we write the content of these involved chunks to the new file and free up memory space in real time. When all the chunk dependency trees are sequentially generated and written to the new file in the same way, the entire file synchronization is finished.

Here, the side effects of IIFC mainly lie in two aspects. On the one hand, buffering *Copy* and *Add* actions consumes additional memory space (usually ranging from several to tens of KB), but it is much less than M_{new} required in traditional `rsync` for large files (≥10 MB). On the other hand, the process of detecting and resolving the dependency costs additional time to conduct an analysis of all *Copy* actions before transmitting data to the server. Traditional `rsync` overlaps detecting and transmitting matching tokens and literal bytes by sending data to the server immediately, while we have to wait until all *Copy* actions are found and analyzed before transmitting data. Nevertheless, this process is usually completed in tens of milliseconds, so the trivial additional sync time is acceptable, considering the significant reductions in memory usage and sync time.

To figure out the impact expected on the performance of the final product for the server-side optimization (IIFC) in terms of our proposed metrics, we conduct extensive experimental evaluations in Section 6.3. Our experimental results show that the reduction of memory usage and the increase of service throughput are mainly due to the server-side optimization (i.e., the IIFC mechanism). The detailed experimental results are analyzed in Section 6.3.

6.3 WASMrsync+: The Final Efficient and Practical Product

The integration of WASMrsync, the client-side optimization, and the server-side optimization creates WASMrsync+. The client side of WASMrsync+ is implemented based on the HTML5 File APIs, the WebSocket protocol, and WASM. In total, it is written in 2,200 lines of JavaScript code and 500 lines of C code. The former deals with the client-side workflow of rsync, and the latter implements the calculation of MD5 checksums to prepare for adopting WASM. The server side of WASMrsync+ is developed based on the node.js framework with 1,500 lines of node.js code and a series of C processing modules with 600 lines of C code for handling the user requests and embodying the server-side optimization (IIFC); its architecture also follows the server architecture of Dropbox. Similar to other solutions, the web service of WAMrsync+ runs on a VM server rented from Aliyun ECS, and the file content is hosted on object storage rented from Aliyun OSS [6]. More details about the server, client, and network configurations have been described in Section 3.3 and Figure 2.

It is worth recalling that even though mainstream cloud storage systems (i.e., Dropbox) do not support delta sync in web browsers, they still have web clients. Thus, although WASMrsync+ also requires deployment at the web client side and the server side, the optimizations of WASMrsync+ including leveraging WASM to calculate MD5 and the implementation of IIFC can all be packaged as dynamic libraries for mainstream cloud storage systems to directly apply to their web clients and servers, respectively. As illustrated in Figure 18, mainstream cloud storage systems do not need to refactor their architectures (i.e., reversing the components ① and ② and modifying interfaces between them like WebR2sync+), but only need to replace the original functional modules (MD5 and **File Construction (FC)**) with our encapsulated dynamic libraries (WASM-MD5 and IIFC), which essentially improve the applicability of WASMrsync+. Note that, since WASM-MD5 and IIFC are highly encapsulated JavaScript and C dynamic libraries, they can be easily integrated into these systems' web clients and servers by following the conventional way of loading JavaScript modules and C libraries. To validate the effectiveness of WASMrsync+, we evaluate the performance of WASMrsync+, in comparison to WebRsync, WebR2sync, WebR2sync+, WASMrsync, and (PC client-based) `rsync` under the aforementioned workloads and metrics in Section 3.

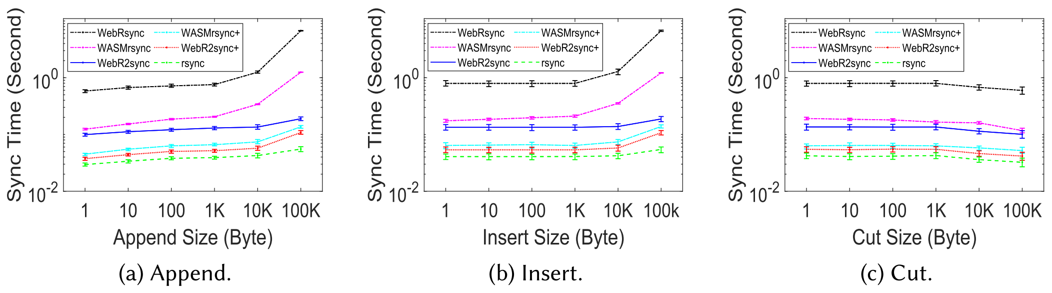


Fig. 25. Average sync time of different delta sync approaches for various sizes of file edits under a simple workload.

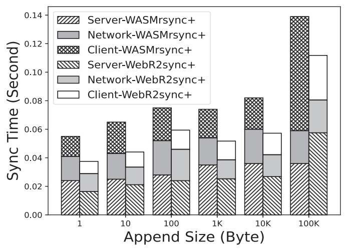
Sync efficiency. We measure the efficiency of WASMrsync+ in terms of the time for completing the sync. Figure 25 shows the time for syncing against different types of file operations under a simple workload. We can see that the sync time of WASMrsync+ is substantially shorter than that of WASMrsync (by 2 to 5 times) and WebRsync (by 13 to 19 times) for every different type of operation. Note that Figure 25 is plotted with a log scale. In other words, WASMrsync+ outpaces WebRsync by around an order of magnitude, approaching the speed of WebR2sync+.

Similar to Figure 22, we further break down the sync time of WASMrsync+ into three stages compared with WebR2sync+ as shown in Figure 26. Comparing Figures 22 and 26, we notice that the majority of client-side and server-side sync time has been largely decreased. Particularly, the file construction time at the server side is saved by nearly 3–15 times, which demonstrates the effectiveness of the IIFC mechanism. This indicates that the computation overheads of the client and server sides in WASMrsync are substantially reduced in WASMrsync+.

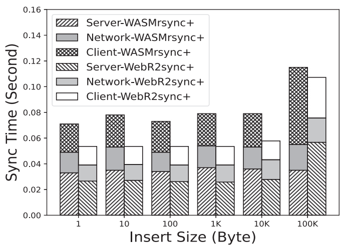
To further explore the respective effects of the client- and server-side optimizations on improving the sync efficiency of WASMrsync+, we additionally measure the sync time of WASMrsync, **WASMrsync with the client-side optimization (WASMrsync-CO)**, **WASMrsync with the server-side optimization (WASMrsync-SO)**, and WASMrsync+. As demonstrated in Figure 27, the sync time of WASMrsync-CO and WASMrsync-SO is between that of WASMrsync+ and WASMrsync. Note that Figure 27 is plotted with a log scale. In other words, both the client- and server-side optimizations can effectively reduce the sync time of WASMrsync. In particular, the client-side optimization has a more significant effect on reducing the sync time of WASMrsync compared with the server-side optimization.

Although the time of calculating MD5 can be significantly reduced by using WASM, it is still about 16%–33% more than that of Siphash according to our measurement results. Meanwhile, even though the IIFC mechanism can effectively reduce the file construction time of WASMrsyc+, it still introduces the additional time of detecting and resolving the chunk dependency. Thus, the sync time of WASMrsync+ is a bit higher (only 10%~20%) than that of WebRs2sync+. Even though WASMrsync+ does not outperform WebR2sync+, WASMrsync+ achieves comparable sync time as WebR2sync+ under the premise of preserving the system reliability and industrial applicability.

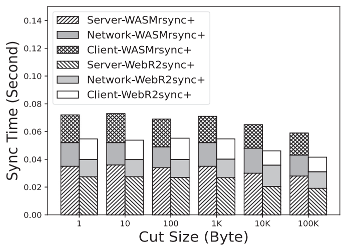
Computation overhead. Moreover, we record the client-side and server-side CPU utilizations in Figures 28 and 29, respectively. At the client side, WebRsync consumes the most CPU resources, while WebR2sync+ consumes the least for shifting the most computation-intensive chunk search and comparison operations from the client to the server. PC client-based `rsync` consumes nearly a half CPU resources as compared to WebRsync. The CPU utilizations of WASMrsync and WASMrsync+ lie between `rsync` and WebR2sync+. This is within our expectation, because WASMrsync and WASMrsync+ perform the most computation-intensive chunk search and



(a) Append.

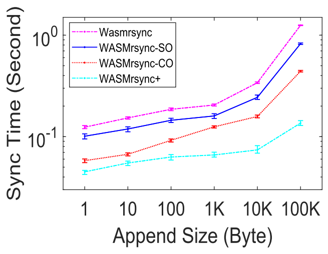


(b) Insert.

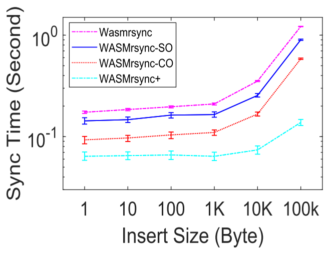


(c) Cut.

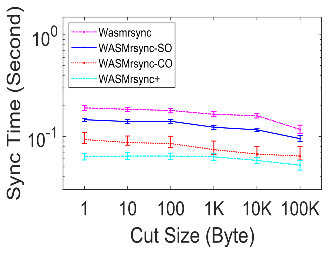
Fig. 26. Breakdown of the sync time of WASMrsync+ and WebR2sync+ for different types of edit operations.



(a) Append.

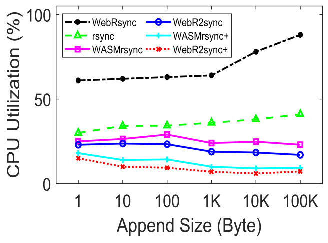


(b) Insert.

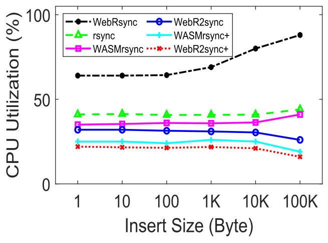


(c) Cut.

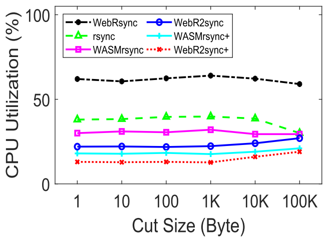
Fig. 27. Average sync time of different WASM-based delta sync approaches for various sizes of file edits under a simple workload.



(a) Append.



(b) Insert.

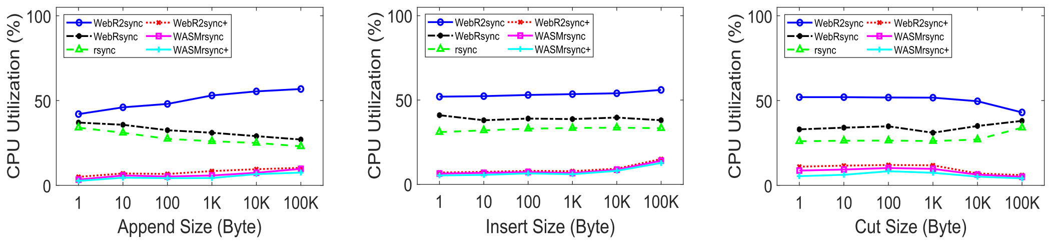


(c) Cut.

Fig. 28. Average **client**-side CPU utilization of different delta sync approaches under a simple workload.

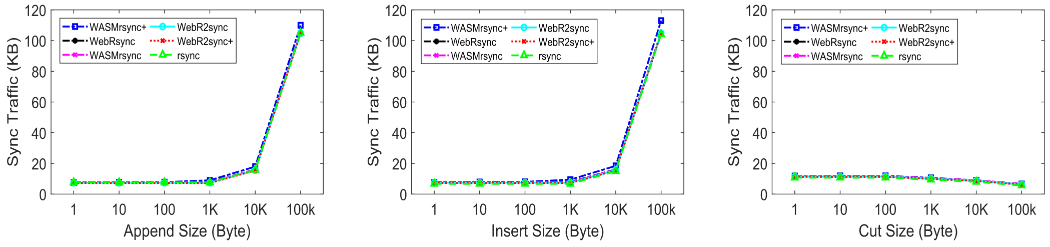
comparison operations at the client side for preserving the original workflow of `rsync` and using the heavyweight but more reliable checksum function–MD5. Nevertheless, the client-side CPU utilization of WASMrsync is significantly reduced compared to WebRsync, demonstrating the effectiveness of adopting WASM at the client side. The client-side CPU utilization of WASMrsync+ is less than that of WASMrsync (nearly by 10%–20%), which proves the efficacy of our two-fold client-side optimizations. Owing to the moderate (<40%) CPU utilizations, both the clients of WASMrsync and WASMrsync+ do not exhibit stagnation, so do WebR2sync and WebR2sync+.

On the server side, WebR2sync consumes the most CPU resources, because the most computation-intensive chunk search and comparison operations are shifted from the client to the server. On the contrary, the server-side CPU utilizations of WASMrsync and WASMrsync+ are both less than that of WebR2sync+, because they do not move the expensive chunk search and comparison operations from the client side to the server side. In particular, WASMrsync+ consumes the least CPU resources, which also validates the efficacy of our server-side optimizations.



(a) Append. (b) Insert. (c) Cut.

Fig. 29. Average **server**-side CPU utilization of different delta sync approaches under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Fig. 30. Sync traffic of different sync approaches for various sizes of file edits under a simple workload.

Sync traffic. Figure 30 illustrates the sync traffic consumed by the different approaches. We can see that for any type of edit, the sync traffic (between 1 KB and 120 KB) is significantly less than the average file size (~1 MB), confirming the power of delta sync in improving the network-level efficiency of cloud storage services. For the same edit size, the sync traffic of an append operation is usually less than that of an insert operation, because the former would bring more matching tokens while fewer literal bytes (refer to Figure 3). Besides, when the edit size is relatively large (10 KB or 100 KB), a cut operation consumes much less sync traffic than an append/insert operation, because a cut operation brings only matching tokens but not literal bytes. It is worth mentioning that although WASMrsync+ spends a small amount of additional traffic to transfer the information of the dependency graph of matched chunks due to the IIFC mechanism, this is trivial compared to the improved overall performance of WASMrsync+.

Memory usage. Furthermore, to validate the efficacy of IIFC, we record the memory usages of all the proposed solutions under heavy workloads (refer to Section 3.2). Considering that the edit sizes (10 KB–1 MB) are relatively small compared to the file sizes of large files (10 MB–100 MB), for every large file, we take the average amount of memory space required by various edit sizes as the final experimental result. As Figure 31 depicted, for any size of large files, with the exception of WASMrsync+, the memory space required by the server side of these solutions is basically twice the size of these large files. On the contrary, WASMsync+ only needs to allocate nearly the same amount of memory as the size of these large files to correctly complete the overall sync process, which demonstrates the power of the IIFC mechanism. In other words, WASMrsync+ can save ~50% server-side memory usage compared to other solutions.

Service throughput. Finally, we measure the service throughput of WASMrsync+ in terms of the number of concurrent clients it can support. In general, as the number of concurrent clients increases, the main burden imposed on the server comes from the high CPU utilizations in all cores. When the CPU utilizations on all cores approach 100%, we record the number of concurrent

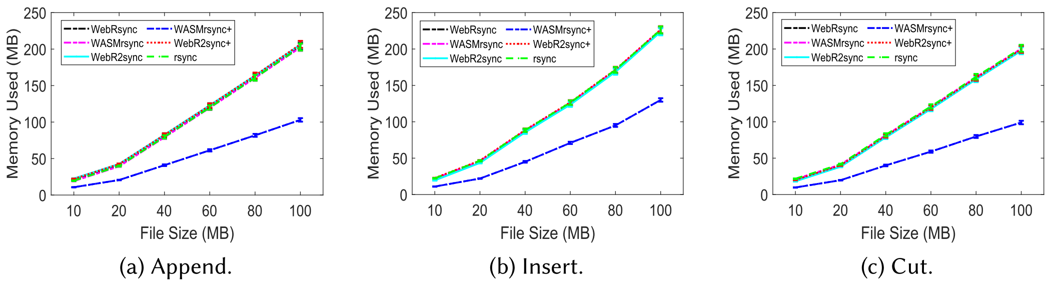


Fig. 31. Memory usage of different sync approaches for various sizes of files under a heavy workload.

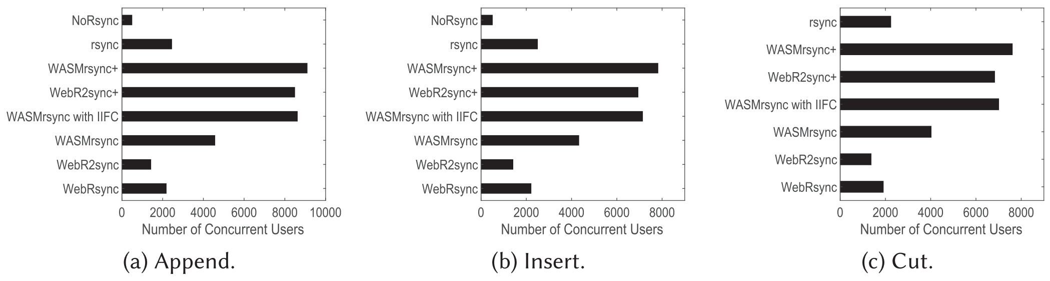


Fig. 32. Number of concurrent clients supported by a single VM server instance (as a measure of service throughput) under regular workloads (periodically syncing various sizes of file edits).

clients at that time as the service throughput. As shown in Figure 32, WASMrsync+ can simultaneously support 7,600–9,100 web clients' delta sync using a standard VM server instance under regular workloads, which is more than WebR2sync+ can support (6,800–8,500). The throughput of WASMrsync+ is 3–4 times that of WebR2sync/rsync and ~17 times that of NoRsync. NoRsync means that no delta sync is used for synchronizing file edits, i.e., directly uploading the entire content of the edited file to the cloud. Also, we measure the service throughput of each solution under intensive workloads (which are mixed by the three types of edits; refer to Section 3.2).

The service throughput is affected not only by the file size but also by the number of sub-edits in the web-based delta sync scenario. Although the average file size in intensive workloads is 23 KB, there are many sub-edits in these involving files, so the CPU utilization of the server can reach 100%. As illustrated in Figure 33, under the intensive workloads, WebR2sync supports fewer concurrent users than the rsync scheme. This is reasonable, because WebR2sync has moved more computational tasks from the client side to the server side. Meanwhile, WebR2sync+ can support more concurrent users than rsync due to our two-fold server-side optimizations, including exploiting the locality of file edits in chunk search and replacing MD5 with SipHash in chunk comparison. The experimental result also indicates the effectiveness of our proposed server-side optimizations.

The results in Figure 33 also show that even under the intensive workloads, WASMrsync+ can simultaneously support the most concurrent users (about 810 web clients' delta sync) compared with other solutions using a single VM server instance, which proves the effectiveness of leveraging WASM and the corresponding client-side and server-side optimizations. To further clarify the respective effects of the client- and server-side optimizations on increasing the service throughput of WASMrsync+, we also measure the throughput of WASMrsync with IIFC. As shown in Figures 32 and 33, the service throughput of WASMrsync with IIFC is very close to that of WASMrsync+. Thus, we can reasonably believe the increase of service throughput of WASMrsync+ is mainly due to the adoption of the IIFC mechanism.

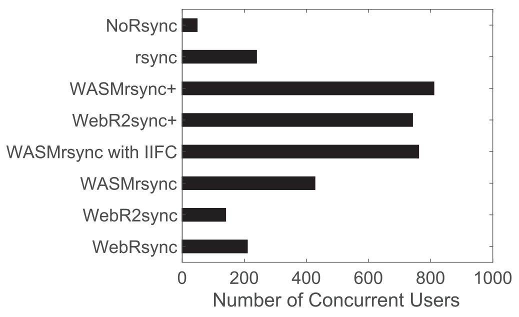


Fig. 33. Number of concurrent users supported by a single VM server instance under intensive workloads (syncing two versions of Linux source trees).

7 CONCLUSION

This article presents our efforts towards supporting efficient and practical delta sync under current web frameworks. Despite a rich body of delta sync techniques in cloud storage services, practical delta sync techniques are only available for PC clients and mobile apps, but not web browsers. To fill in this critical field blank, we first propose a variety of web-based delta sync solutions in a step-by-step optimization manner. Despite not being acceptable in terms of practicability (i.e., reliability, applicability, and usability), they indicate the challenges and optimization opportunities of supporting efficient and practical web-based delta sync solutions for cloud storage services.

Fortunately, WASM sheds light on how to overcome these challenges. Thereby, we leverage WASM to develop a preliminary practical WASM-based delta sync solution named WASMrsync. In practice, we find the inherent natures of WASMrsync lead to great performance degradations. To this end, we further present WASMrsync+ by adopting the client-side and server-side optimizations. WASMrsync+ achieves comparable sync efficiency as WebR2sync+ with nearly only a half of memory usage, but without impairing the practicability. Moreover, WASMrsync+ can simultaneously support more web clients' delta sync than WebR2sync+ using a standard VM server instance.

ACKNOWLEDGMENTS

We would like to acknowledge Ennan Zhai, Tianyin Xu, Quanlu Zhang, Yao Liu, and Mingliang Li for their critical contributions and constructive suggestions to this work. We also thank the editors and anonymous reviewers of TOS for their insightful comments and suggestions.

REFERENCES

[1] Google. 2021. An introduction to the Chrome 61. Retrieved from https://developer.chrome.com/blog/new-in-chrome-61/.
[2] Wikipedia. 2021. An introduction to the V8 (JavaScript) engine. Retrieved from https://en.wikipedia.org/wiki/V8_(JavaScript_engine.
[3] Aappleby. 2021. Murmur3 Hash Function). Retrieved from https://github.com/aappleby/smhashe.
[4] J. Alakuijala, B. Cox, and J. Wassenberg. 2016. Fast keyed hash/pseudo-random function using SIMD multiply and permute. *arXiv preprint arXiv:1612.06257* (2016).
[5] Aliyun.com. 2021. Aliyun ECS (Elastic Compute Service). Retrieved from https://www.aliyun.com/product/EC.
[6] Aliyun.com. 2021. Aliyun OSS (Object Storage Service). Retrieved from https://www.aliyun.com/product/os.
[7] Alon Zakai. 2021. Emscripten, a complete open source compiler toolchain to WebAssembly. Retrieved from https://emscripten.org/index.htm.
[8] asmjs.org. 2021. asm.js, a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. Retrieved from http://asmjs.or.

[9] J.-P. Aumasson and D. J. Bernstein. 2012. SipHash: A fast short-input PRF. In *Proceedings of the Indocrypt*. Springer, 489–508.

[10] G. Banga, F. Douglis, M. Rabinovich, et al. 1997. Optimistic deltas for WWW latency reduction. In *Proceedings of the ATC*. USENIX, 289–303.

[11] Bob Jenkins. 2021. SpookyHash: A 128-Bit Noncryptographic Hash. Retrieved from http://burtleburtle.net/bob/hash/spooky.htm.

[12] E. Bocchi, I. Drago, and M. Mellia. 2015. Personal cloud storage: Usage, performance and impact of terminals. In *Proceedings of the CloudNet*. IEEE, 106–111.

[13] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. 2015. QuickSync: Improving synchronization efficiency for mobile cloud storage services. In *Proceedings of the MobiCom*. ACM, 592–603.

[14] P. Deutsch and J.-L. Gailly. 1996. *Zlib Compressed Data Format Specification Version 3.3*. Technical Report. RFC Network Working Group.

[15] Dokan. 2021. Web Workers. Retrieved from https://www.w3schools.com/html/html5_webworkers.as.

[16] Dokan. 2021. Dokan: A user mode file system for Windows. Retrieved from https://dokan-dev.github.i.

[17] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. 2013. Benchmarking personal cloud storage. In *Proceedings of the IMC*. ACM, 205–212.

[18] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras. 2012. Inside Dropbox: Understanding personal cloud storage services. In *Proceedings of the IMC*. ACM, 481–494.

[19] Jinlong E., Yong Cui, Peng Wang, Zhenhua Li, and Chaokun Zhang. 2018. CoCloud: Enabling efficient cross-cloud file collaboration based on inefficient web APIs. *IEEE Trans. Parallel Distrib. Syst.* 29, 1 (2018), 56–69.

[20] Mehmet Fatih Erkoc and Serhat Bahadir Kert. 2011. Cloud computing for distributed university campus: A prototype suggestion. In *Proceedings of the FOE*.

[21] Richard Finney and Daoud Meerzaman. 2018. Chromatic: WebAssembly-based cancer genome viewer. *Cancer Inform.* 17 (2018), 1176935118771972.

[22] Frank Denis. 2021. A Javascript Implementation of SipHash-2-4. Retrieved from https://github.com/jedisct1/siphash-j.

[23] Geoff Pike and Jyrki Alakuijala. 2021. CityHash. Retrieved from https://opensource.googleblog.com/2011/04/introducing-cityhash.htm.

[24] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the ACM/IFIP/USENIX Middleware*. Springer, 123–135.

[25] Google. 2021. Native Client for Google Chrome. Retrieved from https://developer.chrome.com/native-clien.

[26] B. C. Housel and D. B. Lindquist. 1996. WebExpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the MobiCom*. ACM, 108–116.

[27] J. W. Hunt and M. D. MacIlroy. 1976. *An Algorithm for Differential File Comparison*. Bell Laboratories New Jersey.

[28] J. Hunt, K. Vo, and W. Tichy. 1996. An Empirical Study of Delta Algorithms. In *Software Configuration Management*. Springer, 49–66.

[29] jamesruan. 2021. Javascript version of SpookyHash. Retrieved from https://github.com/jamesruan/spookyhash-j.

[30] Danish Jamil and Hassan Zaki. 2011. Security issues in cloud computing and countermeasures. *Int. J. Eng. Sci. Technol.* 3, 4 (2011), 2672–2676.

[31] K. Basques and P. LePage. 2021. Reading Files in JavaScript using the HTML5 File APIs. Retrieved from https://www.html5rocks.com/en/tutorials/file/dndfiles.

[32] Won Kim, Soo Dong Kim, Eunseok Lee, and Sungyoung Lee. 2009. Adoption issues for cloud computing. In *Proceedings of the MoMM*. 2–5.

[33] D. G. Korn and K.-P. Vo. 2002. Engineering a differencing and compression data format. In *Proceedings of the ATC*. USENIX, 219–228.

[34] E. Kruus, C. Ungureanu, and C. Dubnicki. 2010. Bimodal content defined chunking for backup streams. In *Proceedings of the FAST*. USENIX, 239–252.

[35] Landon Curt Noll. 2021. FNV Hash). Retrieved from http://www.isthe.com/chongo/tech/comp/fnv.

[36] Stéphane Letz, Yann Orlarey, and Dominique Fober. 2017. Compiling Faust Audio DSP code to WebAssembly. Web Audio Conference.

[37] Z. Li, Y. Dai, G. Chen, and Y. Liu. 2016. *Content Distribution for Mobile Internet: A Cloud-based Approach*. Springer.

[38] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. 2014. Towards network-level efficiency for cloud storage services. In *Proceedings of the IMC*. ACM, 115–128.

[39] Z. Li, X. Wang, N. Huang, M. A. Kaafar, Z. Li, J. Zhou, G. Xie, and P. Steenkiste. 2016. An empirical analysis of a large-scale mobile cloud storage service. In *Proceedings of the IMC*. ACM, 287–301.

[40] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. 2013. Efficient batched synchronization in Dropbox-like cloud storage services. In *Proceedings of the ACM/IFIP/USENIX Middleware*. Springer, 307–327.

[41] Z. Li, Z. Zhang, and Y. Dai. 2013. Coarse-grained cloud synchronization mechanism design may lead to severe traffic overuse. *Tsinghua Sci. Technol.* 18, 3 (2013), 286–297.
[42] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the FAST*. USENIX, 111–123.
[43] mozilla.org. 2021. How to use WebAssembly? Retrieved from https://developer.mozilla.org/en-US/docs/WebAssembly/Concept.
[44] mozilla.org. 2021. Making WebAssembly even faster: Firefox's new streaming and tiering compiler. Retrieved from https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler.
[45] mozilla.org. 2021. Using files from web applications. Retrieved from https://developer.mozilla.org/en-US/docs/Using_files_from_web_application.
[46] Pour Rezaei, Parisa. 2015. *User Experience Studies of Personal Cloud Storage Services*. Master's thesis. Tampere University of Technology.
[47] Raluca Budiu. 2019. Mental Models for Cloud-Storage Systems. Retrieved from https://www.nngroup.com/articles/cloud-storage.
[48] R. L. Rivest et al. 1992. RFC 1321: The MD5 message-digest algorithm. *Internet Activ. Board* 143 (1992).
[49] samba.org. 2021. rsync Web Site. Retrieved from http://www.samba.org/rsyn.
[50] seafile.com. 2021. Seafile: Enterprise file sync and share platform with high reliability and performance). Retrieved from https://www.seafile.com/en/hom.
[51] A. Tridgell. 1999. *Efficient Algorithms for Sorting and Synchronization*. Ph.D.'s thesis at the Australian National University.
[52] A. Tridgell and P. Mackerras. 1996. The rsync algorithm. Technical Report. Australian National University.
[53] VAMPIRE. 2021. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Retrieved from https://bench.cr.yp.to/results-hash.htm.
[54] WebAssembly. 2021. WebAssembly, a new portable, size- and load-time-efficient format suitable for compilation to the web. Retrieved from http://webassembly.or.
[55] WebAssembly. 2021. WebAssembly.compile, understanding the async JavaScript APIs about WebAssembly. Retrieved from https://webassembly.org/getting-started/js-api.
[56] Wikipedia. 2021. Delta encoding, the Wikipedia page. Retrieved from https://en.wikipedia.org/wiki/Delta_encodin.
[57] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. 2016. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710.
[58] W. Xia, H. Jiang, D. Feng, and Y. Hua. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the ATC*. USENIX, 26–30.
[59] W. Xia, H. Jiang, D. Feng, and Y. Hua. 2015. Similarity and locality based indexing for high performance data deduplication. *IEEE Trans. Comput.* 64, 4 (2015), 1162–1176.
[60] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. 2016. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the ATC*. USENIX, 101–114.
[61] H. Xiao, Z. Li, E. Zhai, and T. Xu. 2017. Practical web-based delta synchronization for cloud storage services. In *Proceedings of the HotStorage*. USENIX.
[62] H. Xiao, Z. Li, E. Zhai, T. Xu, Y. Li, Y. Liu, Q. Zhang, and Y. Liu. 2018. Towards web-based delta synchronization for cloud storage services. In *Proceedings of the FAST*. USENIX, 155–168.
[63] Alon Zakai. 2018. Fast physics on the web using C++, JavaScript, and Emscripten. *Comput. Sci. Eng.* 20, 1 (2018), 11–19.
[64] Q. Zhang, Z. Li, Z. Yang, S. Li, Y. Guo, and Y. Dai. 2017. DeltaCFS: Boosting delta sync for cloud storage services by learning from NFS. In *Proceedings of the ICDCS*. IEEE, 264–275.

Received 15 June 2021; revised 21 November 2021; accepted 26 November 2021