Gray Computing: A Framework for Computing with Background JavaScript Tasks

Yao Pan, Jules White, Yu Sun, Jeff Gray, *Senior Member, IEEE*

Abstract—Website visitors are performing increasingly complex computational work on the websites' behalf, such as validating forms, rendering animations, and producing data visualizations. In this article, we explore the possibility of increasing the work offloaded to web visitors' browsers. The idle computing cycles of web visitors can be turned into a large-scale distributed data processing engine, which we term gray computing. Past research has looked primarily at either volunteer computing with specialized clients or browser-based volunteer computing where the visitors keep their browsers open to a single web page for a long period of time. This article provides a comprehensive analysis of the architecture, performance, security, cost effectiveness, user experience, and other issues of gray computing distributed data processing engines with heterogeneous computing power, non-uniform page view times, and high computing pool volatility. Several real-world applications are examined and gray computing is shown to be cost effective for a number of complex tasks ranging from computer vision to bioinformatics to cryptology.

Index Terms—Software economics, JavaScript, Web browser, Cloud computing.

1 Introduction

The Internet is much richer and more sophisticated than it was 20 years ago. Instead of only displaying static text and images, dynamic and interactive content are everywhere. Displaying richer content requires the use of greater computational resources on the client browser's host machine. As a result of the increasing richness of Internet content, visitors are performing increasingly complex computational work on websites' behalf: from validating syntax and requirements of input values of a web form before submission to the server, to animating page elements. Website visitors implicitly perform computational work for the website owner without knowing exactly what type of work their browsers will perform. The line between what computational tasks should or should not be offloaded to the visitor's browser is not clear cut and creates a blurred boundary, which we term *gray computing*.

Previously, browser-based JavaScript applications were single-threaded. Complex computations would directly impact user interaction with the web page, preventing the offloading of heavy computational tasks to website visitors. However, the emergence of new standards, such as Web Workers introduced in HTML5, allow background JavaScript threads on web pages, offering the potential for much more significant background usage of a visitor's computing resources.

Offloading computational tasks to browsers can be useful in many circumstances. For example, large websites routinely do big data processing [1] to process website visitor logs, user photos, social network connections, and other large datasets for meaningful information. The question this article investigates is whether or not organizations could cost-effectively offload these big data processing tasks to client web browsers in background JavaScript threads. How much computational power could be harnessed in this type of model, both for computations directly beneficial to the web visitor, such as product recommendations, as well as for attackers that compromise a website?

There are clearly significant legal and ethical questions around this concept of gray computing, but before deeply exploring them, it is important to ensure that there is actually significant potential computational value in gray computing. For example, if the costs, such as added load to the webserver or reduced website responsiveness, reduce web traffic, then clearly gray computing will not be exploited. Further, if the computational power of browsers is insignificant and can not perform sufficient work to outweigh the outgoing bandwidth costs of the data transferred to clients for processing, then no user incentives or other models to entice users into opting into computationally demanding tasks will be feasible.

Gray computing is similar to the concept of volunteer computing [2], which is a distributed computing paradigm in which computer owners donate their idle computing resources to scientific research projects, to reduce costs and speed up research progress. Both gray computing and volunteer computing rely on heterogeneous and untrustworthy clients' computing resources. The problem with volunteer computing is it appeals to only a narrow scope of users: the volunteer population has remained static around 500,000 for several years [3]. That is less than 0.02% of the current number of devices connected to the Internet [4]. It seems the growth of volunteer computing power has reached a bottleneck. Gray computing does not require users to install any client-side software. Users only need to open a webpage to contribute their computing resources. Therefore, gray

- *Yao Pan and Jules White are with the Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, 37203. E-mail: {yao.pan, julesw}@vanderbilt.edu*
- *Yu Sun is with the Department of Computer Science, California State Polytechnic University, Pomona, CA, 91748. E-mail: yusun@cpp.edu*
- *Jeff Gray is with the Department of Computer Science, University of Alabama, Tuscaloosa, AL, 35487. E-mail: gray@cs.ua.edu*

computing has access to a much larger potential pool of clients: popular websites usually have millions of daily visitors. For example, YouTube has about 990 million daily unique visitors and the average daily time for each visitor on site is about 9 minutes, as of April 2017 [5]. Assume that each client computer has an average processing ability of 30 GFLOPS, which is estimated from the average client computing power of the famous volunteer computing project BOINC [6]. Further, conservatively assume that each client computer is only 25% utilized by gray computing tasks offloaded by YouTube. In this case, the combined processing power of the client computers would be 46.4 PFLOPS. For comparison, SETI@home, a volunteer computing project to search for extraterrestrial intelligence, has an average computing power of 0.78 PFLOPS [7]. Sunway TaihuLight, the fastest super computer on record through April 2017, has a computing speed of 93 PFLOPS [8]. In this example, the combined processing power of the web visitors to YouTube is half of the largest supercomputer in the world.

However, gray computing also faces some challenges that are not presented in traditional volunteer computing:

1) *Performance:* Gray computing uses web browsers as the computing medium. Therefore, JavaScript is the core language for computation. However, JavaScript is generally considered slow and not suitable for high performance computing. Further, most computationally heavy tasks are not implemented in JavaScript and porting existing code from other languages to JavaScript can be time-consuming and error-prone.
2) *User experience:* Unlike traditional volunteer computing where users may leave the computers overnight or do computation when a screensaver is running, the users of gray computing are browsing webpages at the same time. The computational work should not interfere with user experience or slow down webpage rendering or interaction in a perceptible manner.
3) *Cost effectiveness:* Although gray computing utilizes the computational resources of each individual website visitor for free, extra costs are still incurred on the server-side due to extra data transfer and CPU load from distributing computational tasks and their associated data. There has not been deep research into the cost-effectiveness of volunteer computing. With cloud computing providers offering elastic computing resources with prices as low as a few cents per hour [9], it is possible that paying for a set of cloud servers to perform a computationally heavy task is cheaper than the outgoing bandwidth costs of a gray computing system.
4) *Client volatility:* Although volunteer computing also suffers from unpredictable availability of clients, the situation is worse in gray computing. The duration of a website visit can last from a few minutes to several seconds, which is far shorter than the average duration in volunteer computing. It is also more likely for visitors to close a browser tab before a computation is complete than to shut down a volunteer computing client, adding load to the server without any return. The unknown page view duration adds significant complexity for devising an algorithm for efficient task distribution.

Open Question $\Rightarrow$ Is it feasible and cost-effective to build a gray computing data processing infrastructure using website visitors' browsers? Although prior research on volunteer computing has investigated browser-based volunteer computing engines, past research has focused on scenarios where website visitors keep specialized web pages open for long periods of time [10]. There has not been a deep investigation into the cost effectiveness of building browser-based distributed data processing engines with respect to the impact on the user experience while visiting a website. Each of these topics is considered in this article as an evaluation of gray computing. A number of key research challenges exist that stem from the volatility, security, user experience considerations, and cost of performing distributed data processing with gray computing. This paper aims to determine if gray computing is an appealing enough target to warrant further research into these issues.

This article presents an architectural solution that addresses these research challenges and proves the feasibility of performing distributed data processing tasks with gray computing. To prove the feasibility of the gray computing concept, we built, empirically benchmarked, and analyzed a variety of browser-based distributed data processing engines for different types of gray computing tasks, ranging from facial recognition to rainbow table generation. We analyzed the computational power that could be harnessed both for valid user-oriented concerns, as well as by cyber-attackers. The same experiments were performed using Amazon's cloud services for distributed data processing in order to compare the performance and cost of gray computing to cloud computing.

To address potential legal and ethical concerns with gray computing, we discuss deployment strategies for gray computing. Gray computing can be deployed so that computing serves as an exchange medium. Users enjoy free services of a website at the cost of performing background computations while browsing. Gray computing can be applied to general business websites with reward incentives to motivate visitors to do computational tasks. For instance, users visiting online retail sites can be asked if they are willing to participate in the distributed computational tasks and shopping points could be rewarded in return. Gray computing can also be deployed as a form of volunteer computing so that volunteers can contribute to scientific projects by simply opening a website instead of installing any software.

Contributions In our previous work [11], we presented a preliminary analysis of computing with background JavaScript tasks. Various issues with gray computing were examined including architecture, JavaScript performance, computational security, cost effectiveness and user experience. In this article, we present a more comprehensive analysis of gray computing and extend our previous research in the following ways:

- We conducted experiments to assess whether gray computing is feasible on mobile device/mobile apps. We evaluated the software environment of mobile devices in terms of how native/hybrid/web apps benefit from gray computing, as well as the particular impact of the hardware environments of mobile devices, including the processing power, network/-battery issues.

- We extended the user experience section to include a more realistic assessment of the user experience impact of gray computing. We have considered different CPU utilization levels, memory and disk usages. We also includee the frames per second (FPS) rate as a responsiveness metric.
- We added experiments to compare the user experience impact of gray computing and online ads. We observed that gray computing has less impact in terms of page load time and page responsiveness than online ads.
- We investigated the software engineering cost and effort to port existing algorithm implementations to JavaScript. Automated source code transformation tools can be used to reduce the cost and complexity of porting existing code to JavaScript. Also, automatically ported code shows better performance than hand-written JavaScript in our experiments.
- In this article, we discuss possible incentive mechanisms to deploy gray computing. Gray computing can serve as an exchange medium where users contribute their computing resources in exchange for free services from websites.
- We extended gray computing applications to bioinformatics computations, such as pairwise sequence alignment [12] and multiple sequence alignment [13]. We port well-known bioinformatics programs to JavaScript and conduct experiments to assess the cost-effectiveness of deploying these bioinformatics computations with gray computing.

The remainder of this article is organized as follows: Section 2 presents the key research challenges investigated in this article; Section 3 presents our solutions to address these challenges and the corresponding empirical verifications; Section 4 examines some practical applications of distributed data processing with gray computing and analyzes their suitability for this computing model; Section 5 discusses threats to validity; Section 6 describes related work; Section 7 presents concluding remarks and future work.

2 Key Research Questions

In order to determine if it is feasible to adopt gray computing, a number of key research questions need to be addressed:

Question 1: Does JavaScript provide sufficient performance for computationally intensive tasks? Gray computing computations are performed in web browsers using JavaScript. However, JavaScript is an interpreted language that is usually not considered for high-performance computation and is generally believed to be several orders of magnitudes slower than C or C++ [14]. Poor JavaScript computational performance could undermine the cost-effectiveness and competitiveness of distributed data processing with gray computing versus traditional platforms, such as Map Reduce clusters.

Question 2: How costly is it to re-implement or port existing algorithms to JavaScript? For many potential applications that could be distributed through gray computing, such as the computer vision or bioinformatics algorithms that we describe in Section 4, the existing implementations of algorithms are usually in C++ or Java due to their perceived performance advantages. There is a lack of corresponding JavaScript implementations. To distribute computational tasks to web browsers, the algorithms need to be implemented in JavaScript and embedded in web pages. Implementing or porting the existing algorithms and libraries to JavaScript could require substantial time and effort and undermine the feasibility of the browser-based approach.

Question 3: Does gray computing work for mobile devices? With the popularity of mobile devices, an increasing portion of website traffic comes from devices such as smartphones and tablets. Of course, mobile devices have very different hardware and software environments compared to desktop computers. Mobile devices do not necessarily access the Internet through browsers, but often by native apps. It is an interesting question to see how gray computing performs in mobile systems.

Question 4: Does gray computing impact website performance in a user perceptible manner? In traditional specialized volunteer computing clients, such as BOINC [15], users are devoted to the computational task and it is acceptable to assume the system resources are utilized. However, in the context of gray computing, website visitors do not want to be interrupted or have a degraded experience while they are viewing web pages. It is important to determine whether or not the background tasks of gray computing will be intrusive to users and impact the responsiveness of a visited web page.

Question 5: What are the mechanisms for handling malicious clients and how will they impact gray computing performance? The nodes of a gray computing engine are arbitrary website visitors and there are no authentication or credentials to guarantee that they are trustworthy. It is possible that attackers can interfere with a gray computing application by sending falsified results to servers. Mechanisms are needed to ensure that the accuracy of gray computing applications will not be negatively impacted by falsified results. Data privacy is another important concern for gray computing. The gray computing application owner generally does not want the clients to access all the data in their storage service, and therefore mechanisms are needed to protect against the leakage of full datasets.

Question 6: How cost-effective is gray computing versus commodity public clouds?

Although gray computing utilizes the computational resources of each individual website visitor, which are free, extra costs are still incurred on the server-side due to extra data transfer and CPU load from distributing computational tasks and their associated data. If the amount of data to be transferred is too large, it is possible that the extra cost added to the server-side is more than the value of the computed result from the client. In other words, it is possible that paying for a set of cloud servers to complete the task is more cost effective than distributing the task using gray computing, particularly when cloud computing providers offer elastic computing resources with prices as low as a few cents per hour [9]. Evaluation is needed to determine whether gray computing is more cost effective than commodity cloud computing resources.

Question 7: How to effectively allocate tasks to clients with unknown page view times? Unlike a centralized data processing engine (e.g., cloud computing servers) where the computation happens on a reliable cluster, a gray computing engine has no control over its clients, who can leave and terminate the computation at will. Page viewing habits differ significantly across users. If a gray computing engine simply assigns tasks to clients randomly, the client may leave before the computation is complete, adding load to the server without any return on investment. The unknown page view duration adds significant complexity to devising an algorithm for efficient task distribution.

Question 8: How to address the potential legal/ethical issues? The offloading of computational tasks to website visitors in gray computing is invisible and does not necessarily need to get a user's approval. This immediately raises the legal and ethical issues of violating a user's right to know and misusing their computation resources. Some key questions to be considered are: What are the potential legitimate ways of deploying gray computing? What are the possible incentive mechanisms to attract visitors and to prevent misuse?

The remainder of this article presents the results of experiments and analyses that we performed to answer each of these key questions. As will be shown, there are practical solutions to the challenges of gray computing and it can be a cost-effective approach to perform a number of complex tasks, such as image processing.

3 Answering the Research Questions

To answer the key research questions, we built an implementation of a distributed data processing engine that could tap gray computing power, benchmarked it, and performed cost/performance analysis on a variety of different data processing tasks. We set out to build the most optimized gray computing engine that we could design to ensure that our cost/benefit analyses were realistic. Initially, we used past research on volunteer computing to guide our gray computing architecture [16], [17], [18], but found a number of architectural assumptions in volunteer computing that did not hold in gray computing hosted in cloud-based websites. In particular, we found architectural issues discussed in prior work, such as mismatches between past architectures and cloud computing pricing, that we sought to address.

Our design focused on optimizing the gray computing distributed processing engine architecture for websites served out of cloud computing environments, such as Amazon EC2. Based on our analysis of the architectures used in past research [16], [17], [19], the key architectural limitations of past research when applied to a cloud computing environment are as follows:

- **Prior approaches assume a fixed sunk cost for the computing time of the task distribution server.** In a cloud computing environment, the cost of the computing time for the task distribution server is not fixed and scales with load. Therefore, past architectures need to be reassessed to minimize the distribution server load and decrease cost. Otherwise, gray computing is less cost-effective.
- **Pricing asymmetry of cloud computing resources can be exploited to reduce gray computing costs.** Amazon S3 and Microsoft Azure only charge for data transfer out of their storage services and not data transfers into their storage services. Prior work did not optimize the data distribution and results reporting architectures to take advantage of pricing asymmetry.
- **Task distribution servers were reported as the bottleneck in some prior work [19].** The load on task distribution servers can be reduced substantially by offloading data distribution to content delivery networks, which provide better cost/performance ratios than serving the same data out of a cloud-based server.
- **All results were reported directly back to the task distribution server in prior work.** Server load can be reduced by allowing browser-based clients to bypass the task distribution server and directly write results to cloud-based storage using a temporary URL authorization model, such as the one provided by Amazon S3.

To address the architectural issues described above, we developed a novel architecture for browser-based data processing engines hosted out of cloud computing environments, such as Amazon EC2. Our architecture was focused on providing websites a MapReduce interface to gray computing, which is commonly used for data processing tasks [1].

To reduce the workload of the task distribution server, we utilized the cloud provider's storage service to serve data for computing directly out of the storage service. The task distribution server is only responsible for determining what tasks should be given to a client. The task distribution server handles HTTP GET requests and responds with a JSON-based string containing the task ID and data location URL for each task. Each client calls the API once before working on an individual task. Our empirical analysis shows a relatively small workload is added to the task distribution server and the cost is negligible.

Besides reducing the workload of the task distribution server, serving data directly out of the cloud-based storage server also exploits the pricing asymmetry in cloud storage services. In Amazon S3 and Microsoft Azure, only outbound data transfer is charged. Data transferred into the storage service is free. This means clients can report the results of computations directly to the storage service for free. This setup allows the data processing engine to reduce bandwidth costs.

Because the clients are highly volatile and typically only have short page view times, improving data transfer speeds is critical to the overall performance. It is essential to maximize the time that clients spend executing computational tasks and minimize the time spent waiting for input data. One optimization that has been applied in our architecture is the use of a Content Delivery Network (CDN) [20]. Instead of serving the input data through a single storage server, a CDN works by serving the content with a large distributed system of servers deployed in multiple data centers in multiple geographic locations. Requests for content are directed to the nodes that are closest and have the lowest latency connection to the client. Consider Amazon's CDN service CloudFront as an example. The original version of the data is stored in an origin server, such as S3. Amazon copies the data and produces a CDN domain name for the data. The clients request the data using the CloudFront domain name and CloudFront will determine the best edge location to serve the contents. An overview of this proposed architecture is shown in Figure 1.

In order to manage and coordinate the computational tasks, we built a cloud-based task distribution server using Node.js and the Express framework [21]. The task distribution server partitions the data into small chunks and assigns them to clients for processing. We use Amazon S3 as our storage server to store input data for clients and receive results from clients. An EC2 server is used to subdivide the tasks and maintain a task queue to distribute tasks to clients. In our architecture, tasks are distributed to clients as follows:

1) The client requests an HTML web page from the server.
2) The server injects an additional JavaScript file into the web page that includes the data processing task.
3) A JavaScript Web Worker, which executes in a background processing thread after the page is fully loaded, is used to perform the heavy data processing computation without impacting the foreground JavaScript rendering.
4) The client sends AJAX HTTP requests to retrieve the input data from the CDN. Once it receives the input data, it runs a map or/and reduce function on the data.
5) The client issues an HTTP PUT or POST of the results directly back to the cloud storage server. After submitting the results, the Web Worker messages the main thread to indicate completion. Upon receipt of this message, the main thread sends a new HTTP request to fetch another data processing task from the server.

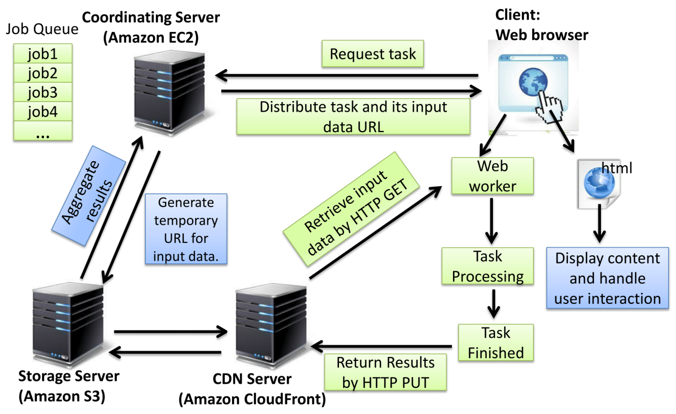

Job Queue
job1
job2
job3
job4
...
Coordinating Server (Amazon EC2)
Client: Web browser
Request task
Distribute task and its input data URL
Aggregate results
Generate temporary URL for input data.
Retrieve input data by HTTP GET
Web worker
html
Task Processing
Display content and handle user interaction
Task Finished
Storage Server (Amazon S3)
CDN Server (Amazon CloudFront)
Return Results by HTTP PUT

Fig. 1. Architecture of our distributed data processing engine for gray computing [11].

3.1 Benchmarking JavaScript Performance on Computationally Intensive Tasks

Moving the computation from a general OS to web browsers, JavaScript is the key language to carry out the computational tasks, so we first set out to assess the performance of JavaScript.

JavaScript was previously not a high-performance language, as early experiments and benchmarks showed [14]. However, as the Internet evolved and web pages became increasingly complex and interactive, there has been a growing demand for high-performance JavaScript engines to run tasks, such as 3D rendering, machine learning, or image processing. For example, the entire Unreal gaming engine has been ported to JavaScript and runs at acceptable frame rates [22].

Because the performance of JavaScript is critical to the user experience on complex sites and directly affects page load and response time, popular browsers such as Chrome, Firefox, and Safari have made great efforts to optimize their JavaScript computing engines and have produced significant increases in JavaScript computational speed.

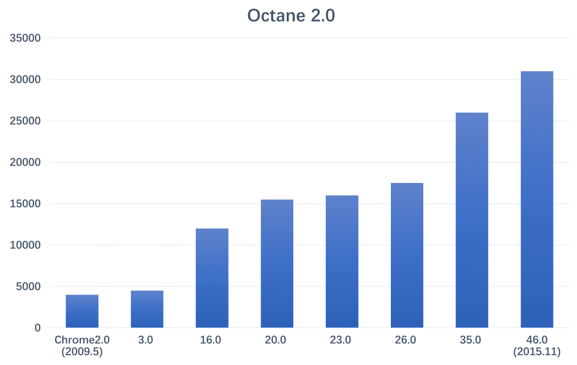


Fig. 2. JavaScript benchmark scores for different versions of browsers over time.

Figure 2 shows the scores of Octane [23], which is a JavaScript benchmark, on different versions of Chrome browsers. These benchmarks include various representative real-world problems. The earliest version of Google Chrome (2.0) dates back to 2009 and the latest version is from 2015. As we can see, the JavaScript Engine performance has improved dramatically in recent years.

TABLE 1
Computing Time Comparison of Programming Languages For Representative Computing Tasks [24].

Benchmark	JavaScript	Java	PHP	C
Regex-dna	**3.55s**	20.80s	28.52s	5.46s
NBody	27.90s	21.54s	5min	**9.96s**
k-nucleotide	120.54s	46.38s	45.69s	**34.26s**
fasta	16.5s	5.66s	59.37s	**5.28s**
Binary-Trees	34.3s	**7.50s**	9min	9.64s
SpectralNorm	15.72s	16.37s	7min	**7.85s**
Smith-Waterman	18.3s	12.9s	N/A	**3.17s**
RONN	34.9s	22.5s	N/A	**8.1s**

Table 1 shows performance results obtained from several benchmarks implemented in different languages [24].

The computer language benchmarks game is a site that includes implementations of the same algorithm in different programming languages. We selected JavaScript, Java, PHP, and C for comparison. The performance of JavaScript was tested on the Google V8 JavaScript engine. The following computational tasks were selected for the benchmarking. Different types of computations that we expected to be similar to useful gray computing tasks were covered.

- **Regex-dna** is designed to test the performance of a regex pattern match on a DNA sequence. Regex pattern match is a common operation for sequence alignment.
- **k-nucleotide** repeatedly updates hashtables and k-nucleotide strings. It is used to test integer operations.
- **SpectralNorm** calculates an eigenvalue using the power method. It tests floating-point operations.
- **Smith-Waterman** is a sequence alignment algorithm implementation that we will discuss in Section 4.
- **RONN** is a protein disorder prediction program that we will discuss in Section 4.

As can be seen in Table 1, C performs the best, as expected. However, the speed difference between JavaScript and C on the benchmarked computing tasks is at most 6X. Moreover, JavaScript shows absolute advantages over popular server scripting languages, such as PHP. To further examine JavaScript efficiency, we also benchmarked a number of complex computational tasks in the evaluations performed in Section 4 with real-world use cases. The results confirm the conclusion above that JavaScript has comparable performance with native C/C++ code.

Although we did not test their capabilities, newer proposed JavaScript libraries and frameworks, such as WebCL [25], enable client JavaScript programs to access client GPUs to further improve computational speed. WebCL is a standard for JavaScript implementation of the OpenCL API, which enables web applications to harness GPU and multi-core CPU parallel processing from within a Web browser. Currently, WebCL is still in draft version and no native support is provided by common web browsers. Therefore, we did not include WebCL in our experiments. However, there is research showing that WebCL can bring up to a 50% reduction in execution time by utilizing the GPU [26], which could be a superior technique for gray computing in the future.

3.2 Porting code to JavaScript

Besides the efforts from popular web browsers on improving the JavaScript interpreter, there is another direction of efforts that aims to port existing code from other programming languages to JavaScript. A variety of highly optimized tools have been developed to automatically port C/C++ programs to JavaScript.

Emscripten [27] is a compiler developed by the Mozilla team that compiles LLVM (Low Level Virtual Machine) assembly code into highly-optimizable JavaScript in the asm.js [28] format. It works by first compiling C/C++ code through Clang into an intermediary representation in LLVM. The ported JavaScript code is machine generated and thus different from the native hand-written version. A key question is the amount of performance overhead added by the machine-generated implementation compared to the original implementation. Prior research results [29] showed that optimized JavaScript code can achieve performance of roughly 2x to 8x slower than C++ code. We conducted some benchmark experiments of our own to validate this claim. The results were quite startling: the ported JavaScript code not only outperformed the hand-written JavaScript code, but *achieved performance near the native hand-written C++ code*. Table 2 shows the results from the comparison. The first column is the hand-written native C++ code. The second column is the equivalent hand-written implementation of the JavaScript code. The third column is the ported JavaScript code produced by translating the hand-written native C++ code using Emscripten 1.2.0 with default parameters. The fourth column is the ported JavaScript code using Emscripten with the "-O2" optimization flag. The C++ code was compiled using LLVM gcc 5.0 with the "-O3" optimization parameter. The exact reason for the dramatic performance improvement of the Emscripten generated code is unclear. However, we hypothesize, and will investigate in future work, that the reason for the performance boost is that Emscripten targets a subset of JavaScript called asm.js that is designed to be highly optimizable by JavaScript interpreters. The reduced feature set of asm.js translates into much better performance. Asm.js eliminates JavaScript's object-oriented constructs and thus eliminates many hard-to-optimize code constructs. A more detailed description of asm.js can be found in [29].

Emscripten not only works on single C/C++ files but also complex projects containing hundreds or thousands of source files. The authors of Emscripten have successfully ported hundreds of C/C++ projects into JavaScript and a list of the ported projects is available on Emscripten's homepage [27]. Some of the ported projects include Unreal Engine 4 (3D gaming), Qt (UI), and SQLite (Database), and ffmpeg (video processing). In our own evaluation, we collected five widely used bioinformatics algorithms and successfully ported three of them into JavaScript with Emscripten. Emscripten does not work on all C/C++ projects. In our experiment, porting complex codebases often results in various compilation errors that must be resolved manually. One problem we encountered with Emscripten is that it will not work if the source files contain any file system API calls because JavaScript cannot access files in a Browser. Emscripten provides a workaround by preloading and embedding the files before hand. Emscripten also will not work if the source files contain any multi-threaded code. We will need to manually change the code to a single-threaded structure. Another issue is when there are inline functions declared in a header, which Emscripten cannot recognize and will report as an undefined function error. We will need to make the function a static declaration. Also, the source file cannot contain any platform-dependent statement. The source code will need to be modified in order to be platform-independent. These types of errors have straightforward fixes, but do require manual intervention. The Emscripten FAQ page [30] describes additional situations where manual interventions are needed for porting code to JavaScript. However, Emscripten is a project is actively developed and maintained, so many of these limitations may be fixed in the future.

TABLE 2
Execution Time Comparison of Native C, Native JS, and Ported JS (from C).

Benchmark	C	native JS	ported JS
Nbody	6.6s	16.9s	6.2s
fasta	4.6s	15.9s	8.8s
SpectralNorm	6.7s	8.6s	7.8s

The Google Web Toolkit (GWT) [31] is an open source tool developed by Google that allows web developers to create JavaScript front-end applications in Java. The Java to JavaScript compiler of GWT, which originally was used to facilitate web development, can be used to port Java-based algorithms. We used GWT to port BioJava functions to JavaScript as part of our evaluation. There are also many successful examples using GWT to port complex Java projects to JavaScript such as Quake 2 [32] and the Eclipse Graphical Editing Framework [33]. The down-side of the GWT approach is that most high performance algorithms are implemented in C/C++. Thus, the application scope of GWT is limited. Another disadvantage is that GWT does not have as many optimizations as Emscripten, so the ported JavaScript code tends to be slower than JavaScript code ported by Emscripten [34].

There are other automated translation approaches that target JavaScript besides Emscripten and GWT: pyjs [35] which compiles Python into JavaScript, and Opal [36] which compiles Ruby into JavaScript. However, we focused on GWT and Emscripten in our experiments. A more extensive evaluation of the various approaches will be one direction of our future work.

The process of porting code to JavaScript using GWT and Emscripten is straightforward. For GWT, we add the target Java source code to the GWT project and write a wrapper class in Java to call the function in the target source code. The JavaScript front-end is generated automatically. To port a project with Emscripten, the ”emmake” command is called instead of ”make” to build the project first and the ”emcc” command is called on the generated LLVM code to produce a JavaScript file. The situation can be complex when the project has external library references and the porting may require some extra effort. However, the advantage is the porting process can be done by someone with experience only once and the resulting program can serve the general community.

Another issue with the ported code is the code size. We observed that the Emscripten-generated JavaScript code is much larger than equivalent hand-written JavaScript code. This is understandable since the ported JavaScript is more low-level. But it can be an issue since the JavaScript file needs to be transferred to clients for computation and these files would take extra network bandwidth. Two methods could be applied to optimize the JavaScript code size. The first is the Closure compiler [37], which compiles JavaScript to optimized JavaScript by removing dead code and renaming variables. The second way is to compress the file with gzip. This makes sense because gzip is supported by HTTP request headers and the compressed JavaScript files can be easily decompressed on the client side. Table 3 shows different implementations of rainbow table generation described in Section 4. The file size of JavaScript can be significantly reduced after applying the Closure compiler and gzip.

TABLE 3
File Size Comparison for Different Versions of Rainbow Table Generation.

Version	File size
C++	6kb
hand-written JS	4kb
ported JS	574kb
ported JS with closure compiler	464kb
further gzipped	113kb

3.3 Assessing the Feasibility of Gray Computing on Mobile Devices

With the popularity of mobile devices, mobile visitors are consuming a majority of a websites overall traffic. Gray computing can theoretically be applied to mobile visitors as well. However, the hardware and software environments of mobile devices are different from a typical desktop configuration. For example, mobile devices not only use browsers, but also apps to access web content. Also, mobile devices usually have limited processing power, battery resources and network bandwidth compared to desktops. These differences raise questions on whether gray computing can still work in a mobile environment. To assess the feasibility of applying gray computing to mobile visitors, we conducted experiments to evaluate the performance of running gray computing tasks on mobile devices.

Currently, mobile apps can be categorized into three types: native apps, web apps and hybrid apps. Native apps are apps created with native Android or iOS SDKs. Under the hood, they use Java or Objective C/Swift. Native apps have access to low-level hardware APIs and are distributed in app stores. Web apps are actually web pages but with the option to be installed on your home screen by creating a bookmark. Web apps rely on HTML/CSS/JavaScript. Web apps are interpreted and sandboxed, and hence have less access to the camera, GPS, accelerometer, etc.

Hybrid apps are web apps in native shells. There are two sub-types of hybrid apps. The first kind of hybrid apps are built with hybrid app frameworks such as Apache Cordova, Ionic, React Native, Appcelerator, etc. These frameworks allow developers to focus on one codebase (usually in JavaScript), and automatically generate apps that can be run on different platforms such as Android, iOS, etc. These frameworks are becoming increasingly popular among developers because of their cross-platform capabilities. Only one codebase is needed instead of multiple codebases for different platforms. This can significantly speed up the development process and reduce development and maintenance costs. The second kind of hybrid apps are apps built by Native SDK with WebView components. A WebView is like an in-app browser for developers to embed web content inside the apps. It is sometimes difficult to tell them from the native apps. The WebView approach is very popular especially when a lot of apps mainly show static data and

TABLE 4
Performance comparison of computing 200000 SHA1 hashes on various iOS platforms. Experiments conducted on an iPhone 6 Plus with iOS 10.3.

Platforms	Computation time
Native iOS	3.9s
Safari browser	5.7s
UIWebView	32s
WKWebView	6.0s
React-Native	51s

TABLE 5
Performance comparison of computing 200000 SHA1 hashes on various Android platforms. Experiments conducted on an Android Tango tablet with Android 4.4.

Platforms	Computation time
Native Android (4.4)	8.5s
Chrome browser	6.3s
WebView Android (4.4)	6.7s
WebView Android (4.0)	19.1s
React-native	26.1s

are generally mobile versions of the websites. Although we do not have an exact number of what percentage of apps are hybrid apps, many top apps are embracing hybrid design [38].

How does gray computing work in different mobile-app settings? For pure native apps, gray computing might not be directly applicable because JavaScript is not used in native SDKs. However, the architecture of gray computing, the adaptive scheduling algorithm, the cost analysis, and security issues we described are still useful and could be used to manage native tasks distributed to mobile clients. Because web apps use HTML/JavaScript and run in browsers, there is no difference with the desktop browser setting. For hybrid apps, WebViews are very similar to browsers and can be used to render HTML/JavaScript content inside apps. However, WebViews have different implementations under different platforms (UIWebView/WKWebView for iOS and WebView for Android). It is unclear if WebViews can achieve similar performance to desktop browsers and if WebViews enforce any additional limitations on JavaScript.

In order to understand JavaScript performance in hybrid apps, we conducted experiments to test the performance of computationally intensive tasks in different WebView implementations. The task we chose was a computation of 200,000 SHA1 hashes of sequentially generated strings of length 8. As we can see from Table 4 and Table 5, newer versions of WebView (WKWebView for iOS and WebView in Android 4.4+) have similar JavaScript performance with the mobile browsers of the corresponding platforms. Legacy WebView versions (e.g., UIWebView for iOS and WebView in earlier versions of Android) can be significantly slower than their mobile browser couterparts primarily because of the lack of support for JIT and other optimizations.

Besides the software environment, gray computing also faces different hardware environments on mobile devices. The processing power of mobile devices are usually considered slower than desktops. However, the performance gap is being narrowed as the processing power of mobile devices has improved rapidly in recent years. Table 6 shows the JavaScript performance comparison of the desktop and mobile devices. A mid-range desktop and several representative mobile devices are chosen to run MD5/SHA1 hashing in JavaScript. Chrome is used for desktop and Android devices and Safari is used for iOS devices. The experiment shows iPhone 7 and iPad Pro have already achieved similar performance of a mid-range desktop in terms of MD5/SHA1 hashing. Of course, the iPhone 7 is a very high-end smartphone and is not very representative of all mobile devices in use. But the trend is that the smartphones are becoming faster while the processing power of desktops has made limited progress.

TABLE 6
Performance comparison of various mobile devices with desktop computers. The table shows the time needed to compute same number of MD5/SHA1 hashes for different platforms.

Devices	MD5	SHA1
Desktop(i5 3570)	12.8s	16.3s
iPhone 7	11.8s	16.1s
iPhone 6	23.2s	31.0s
Samsung Galaxy S7	19.7s	27.4s
iPad Pro (2015 model)	13.2s	18.1s

Another obstacle with running gray computing tasks on mobile devices is the limited battery life and network data usage. Gray computing tasks may drain the batteries and increase the network data usage cost for devices using a cellular data plan. In general, we do not encourage users to use battery and cellular data plan to perform computation. The computational tasks can be configured to work only when the mobile devices are charged and when WiFi is available by checking the JavaScript Battery Status API [39] and Network Information API [40]. The Battery Status API provides a way for JavaScript to know whether a device is being charged or not and what is the battery charge level. The Network Information API [40] provides information about a mobile device' connection status (e.g., 'Wifi', 'cellular', etc.). The network information API is currently experimental and supported in Chrome and Firefox only.

Users should have the right to turn on or turn off the computation at any time, even when they are using battery and cellular network. Therefore, the gray computing tasks may incorporate a UI element, such as a button, which allows users to start or terminate a gray computing computation at anytime.

Moreover, gray computing is focused on short duration computations instead of long-running tasks. According to statistics from Alexa [5], the average time users spend on most websites is 3-10 minutes. A few minutes of computation will consume some battery life on a mobile platform, but it is similar in computational demand to viewing a short video. The power consumption for the short duration computations that gray computing is designed for is unlikely to be significant. We are also looking forward to advances in wireless charging. After it becomes practical, the limitations of power consumption will be less impacted.

There are some existing projects that suggest mobile users are passionate with respect to contributing their processing power. Power to Give is an app developed by HTC for smartphone users to contribute their mobile devices' processing power to scientific research. The app has over a million installs so far. Therefore, we believe mobile users have the potential to participate in gray computing tasks regardless of battery and bandwidth consumption.

In summary, we believe gray computing still works for mobile device settings, although with some additional restrictions. Considering the portability and increasingly long time people spend on their smartphones and tablets, mobile devices can definitely contribute significant computing power to gray computing.

3.4 Benchmarking Background Web Worker Computational Task Impact on User Experience

One concern with deploying gray computing to websites is whether the background computing tasks will affect the website's foreground user interaction tasks. Gray computing will be of little interest to website owners and visitors if the computing tasks affect the user experience heavily.

Web Workers is a new feature introduced in HTML5 [41], which can create a separate JavaScript thread in the background. Using Web Workers, background computing tasks and foreground rendering / user interaction tasks can run in separate threads, allowing the main JavaScript needed to render the page and validate forms to run unimpeded without affecting user experience. Some examples on the usage of Web Workers can be found here [42].

To test whether doing computationally intensive tasks in the background will affect the user experience, we conducted experiments with Tampermonkey [43] to inject data processing tasks into popular websites. Tampermonkey is a browser extension that allows users to install scripts that make on-the-fly changes to web page content before or after the web page is loaded in the browser.

The first concern regarding user experience is page load time. However, Web Workers spawn threads in the *onload()* function of a web page, which is executed only after the HTML page finishes loading. Therefore, background Web Worker threads can be run without affecting page load time.

The second concern is the page responsiveness, which is an important metric to ensure users do not experience any obvious lag while interacting with the page. Many metrics exist for page responsiveness. In our experiments, we mainly consider two metrics that simulate real user scenarios. The first metric is the frames per second (FPS) of the web pages while scrolling or playing animations. If the frame rate drops below a certain threshold, users may feel the web pages are not smooth while interacting. The second metric is the feedback lag after some interaction. We chose the search box item suggestion pop up time for evaluation because it is a pervasive and time-critical feature of many websites.

For the first metric, we used FPS to measure the impact of Web Worker tasks on responsiveness. We measured the FPS under two scenarios. The first scenario is while playing animations. We used an animation speed test [44] to record the FPS rate while the web page is playing various animations. We adjusted the complexity of the animation to make the frame rate around 30 FPS when no background tasks were running. The second scenario is while scrolling up and down the pages. The rendering burden increases dramatically while users scroll the web pages. FPS often drops while scrolling and therefore harms the user experience. We used Jank Meter [45] to record the FPS rate while scrolling up and down the web pages. We measured the FPS differences with a Web Worker tasks and without a Web Worker tasks under different background system loads.

TABLE 7
FPS Rate Under Different System Loads.

System Loads	Average FPS while playing animation
CPU 25%	32
CPU 50%	30
CPU 75%	16
Memory 30%	32
Memory 40%	33
Memory 60%	31
Memory 80%	31
Anti-virus (20% CPU)	30
Anti-virus (70% CPU)	18

We simulated different background system loads of CPU utilization, memory usage and disk access while running Web Worker tasks. For CPU utilization, we used CPUSTRES to simulate a background CPU utilization rate of 25%, 50%, 75%. For the memory usage, we used HeavyLoad to simulate memory usage of 30%, 40%, 60%, 80% on an 8GB RAM computer. For the disk access load, we simulated heavy disk access by running an anti-virus software scan. Notice the anti-virus scanning also contributes 20%-70% CPU utilization while running.

The experiment results in Table 7 show a Web Worker task only results in very small reduction in FPS rate when the background CPU utilization level is below 50%. However, the difference becomes larger as the CPU utilization level goes up and when reaches over 70%, there will be a significant drop of FPS rate. For the memory usage, there is no discernible difference after running a Web Worker task under different memory usage percentages. This is probably due to the relatively low memory footprint of most gray computing tasks as shown in Table 9. For the disk usage, pure disk access does not result in much difference in the FPS rate. However, the CPU utilization level while performing anti-virus scanning can reach more than 60% in peak, which results in temporary reduction of the FPS rate.

In summary, the CPU utilization level is the key factor affecting the web page responsiveness. An additional Web Worker task will not have user perceptive impact on the responsiveness of the web page when the background CPU utilization is below 50%. However, there can be significant slow down when the background CPU utilization level is more than 75%. The assumption we have is that users computers tend to stay idle for most of time while users are surfing the Internet. We conducted baseline CPU utilization monitoring on a desktop (with an Intel i5 3570 CPU) running Windows 10 using a performance monitoring tool [46] while the browser was open. The average CPU utilization level is only 10% - 20%. Of course, the situation varies from user to user. Therefore, the Web Worker tasks will have a visible user interaction element for users to terminate the computation at any time. After all, we are only trying to utilize the "idle" computing resources of users.

For the second metric, our goal was to test if the generation of search suggestions would be slowed down by a computationally intensive background task. Two scripts were written in Tampermonkey [43]. Script 1 was setup to inject a search term into the search bar and record how long it took for the web page's foreground client JavaScript to generate the search result HTML elements and produce a baseline time for comparison. Script 2 spawned a Web Worker to perform a computationally intensive task required by the applications we propose in Section 4, such as face recognition and image scaling, and then injected the search terms. The time to generate auto-completion suggestions while running the computationally intensive background tasks was measured. We compared the time to generate the search suggestions with and without the Web Worker task to see if the background computation would impact the user experience on a number of popular websites.

We used 50 different keywords as search inputs and for each keyword we ran 100 trials and averaged the load times. As the results shown in Table 8 illustrate, there was no discernible difference in the search suggestion load time with and without a Web Worker computational task running when the background CPU utilization is below 50%. This is consistent with the results in the previous FPS experiments. The results show that background gray computing tasks would not be easily discernible by the user – causing both alarm due to the potential for misuse and potential from a distributed data processing point of view. The experiment was conducted on Chrome 41.0. We also conducted experiments in Firefox with the similar browser extension Greasemonkey [47]. However, Firefox is currently a single-threaded architecture. Therefore, computation in a Web Worker of one tab would completely freeze the browser. The Firefox team is working on a project called Electrolysis [48] which adds multiple process ability to Firefox. Thus, the user experience should not be a problem for Firefox users in the near future. For IE, Web Workers are supported since 10.0. Computational tasks cannot be run in the background in earlier versions of IE.

TABLE 8
Comparison of Average Search Box Hint Results Generation Time with and without Web Worker Tasks.

System Load	Search Box Generation Time		
	Twitter	Wikipedia	Gmail
Without Web Worker	0.59s	0.46s	0.73s
With Web Worker (25% CPU util)	0.58s	0.48s	0.75s
With Web Worker (50% CPU util)	0.62s	0.50s	0.75s
With Web Worker (75% CPU util)	0.75s	0.71s	1.13s

To measure the added CPU load on the system, we stopped all other non-essential processes on the system that could consume CPU time. When we loaded the page with a Web Worker's computational task and injected the search term, the utilization rate of Core 1 was increased to 10-15%. The CPU utilization rate of Core 2 increased to 100%. After the search results were loaded, utilization of Core 1 remained around 3% and utilization of Core 2 remained at 100%. When we loaded the page without a Web Worker, only the utilization rate of Core 1 changed and Core 2 remained 0. The results show that the browser could efficiently leverage multiple cores and isolate the CPU load of the background task to prevent it from impacting foreground JavaScript performance.

TABLE 9
Memory Consumption of Some Gray Computing Tasks.

Tasks	Peak Memory Usage
MD5 hashing	12MB
Face detection	90MB
Image scaling	67MB
Rainbow table	20MB

We noticed that our results are different from some results observed in previous work [49] where a serious performance degradation can be seen due to the volunteer computing clients. When users are working on their computers and take a break to leave it to run some volunteer computing applications, the applications can pollute the system's memory and lead to cache contention and paging. When users come back to work, the performance of the application they use is impacted. The reason gray computing does not suffer from the same issue is that the memory demands of gray computing applications are typically low because the task distribution system pessimistically divides work into small units that can be processed fully and have their results submitted within the possibly short page view durations, which generally produce very small memory footprint.

3.4.1 User Experience Impact Comparison with Online Ads

In our previous experiments, we have shown that gray computing tasks have little impact on user experience when the background system load is not heavy. However, there is an impact when the system load is high. Is it worthwhile for websites to deploy gray computing, and will website visitors accept gray computing?

We can consider this question from another perspective. Gray computing provides a way for website owners to reduce cost or make money. There are some other ways for website owners to monetize their websites. For example, online ads are a very popular and mature way for website owners to make money by displaying advertisement on the websites. How is gray computing compared to online ads in terms of user experience impact and revenue?

Online ads can be visually annoying. Users often do not have a choice but to view what is displayed. Many ads exist in the main UI thread and the ads are becoming increasingly dynamic and interactive in order to draw attention. As a result, web pages may take longer to load because the need to fetch various ad related resources. Web pages may become less smooth due to the extra system resources consumed by ads. In comparison, gray computing runs silently on a background thread and is less likely to affect user experience. In order to better understand the user experience impact of ads versus gray computing, we conducted experiments to compare the user experience impact of gray computing and ads in terms of two metrics.

First, we compared the impact of ads and gray computing on a website's page load time. We used Adblock to

TABLE 10
User Experience Impact of Web Worker and Ads

	Avg Page Load Time	Frame Rate while Scrolling	
		Average FPS	Minimum FPS
Original Website (with Ads)	8.5s	46	17
With Web Worker	8.5s	45	17
After Ads Block	3.2s	56	31

temporarily block all the ads and recorded the page load time before and after blocking. We tested on 100 different URLs on 20 different domains (the URLs are chosen to include a moderate number of ads). As shown in Table 10, the average page load time without ads blocking is 8.5s and only 3.2s when the ads are blocked. We can see ads significantly slow down the page load time of web pages. While for gray computing, Web Workers are executed in the *onload()* function, so the computational tasks are started only when the page load finishes. Our experiments confirmed that the page load time is the same with and without Web Worker tasks. Therefore, gray computing has an advantage in page load time compared to ads.

Second, we measured the FPS rate while page scrolling. Browser cache is disabled to avoid affecting rendering speed. Depending on the number and types of ads displayed on the web pages, there are different degrees of slow down to the FPS rate while scrolling. While for gray computing, the frame rate is consistent while the background CPU utilization level is below 50%.

Gray computing has less user experience impact than ads. But can gray computing help website owners make as much money as ads? We use Google Adwords as an example, the cost per click advertisers pay is around $0.5, the average click through rate is around 0.3% [50]. Since Google as the platform will take some money, the website owners will get less than $0.0015 for one visitor. For gray computing, suppose the average visiting time is 5 minutes and average processing power of visiting clients is equivalent to an EC2 m3.medium instance, which is priced at $0.06/hour. Suppose the utilization rate is 50%. Website owners can get $0.06/2*5/60 = $0.0025 computation value from one visitor. Because website owners need to give part of the revenue to users as incentives, the websites will make less than $0.0025. Depending on the computational task, length of stay of visitors, and power of visitors computers, gray computing can generate comparable revenue value to ads and with less impact on the user experience of activities, such as scrolling and page load speed.

Another perspective is ads only bring revenue to the website owners but gray computing can bring income for both the website owners and users. Further, some tasks that are performed by the users machine can be beneficial to the user (e.g., automatic tagging of friends faces in photos, automatic summarization of reviews to help the user evaluate a product, etc.).

In summary, we are not arguing gray computing is better than ads, because ads are still a useful way for business marketing and for customers to get information. But gray computing can be a viable supplement or alternative for website owners to make money. Website owners may consider reducing the number of ads to improve user experience and deploy some gray computing tasks instead.

3.5 Mechanisms for Handling Malicious Clients and their Associated Overhead

Because clients in a gray computing distributed data processing engine are not guaranteed to be trustworthy, it is possible that the results returned from them can be fraudulent.

Malicious client attacks can be classified as follows:

- **Independent malicious attacks:** In this case, malicious clients act independently. Given the same input, they will return arbitrary incorrect results. Unintentional errors such as I/O failure or CPU miscalculations can also be modeled by this category [51].
- **Collaborative malicious attacks:** A stronger type of attack is when attackers have control over a group of clients. When a client from the coordinated group of malicious clients is assigned a task from the server, it checks to see whether another malicious client has received a task with the same input. If so, the malicious client will return the same falsified result in order to fool the server. It is possible that multiple collaborative malicious parties exist simultaneously. However, in the following analysis, we will consider the worst case in which all malicious clients collaborate with each other.

If the server does not verify results, falsified results will be accepted by the server and taint the overall accuracy of data processing tasks. Therefore, verification mechanisms need to be implemented in order to guarantee that the overall result will not be rendered useless by a small subset of malicious clients. One simple and basic approach to handle malicious results is to duplicate tasks, where identical task units are assigned to different clients to compute and clients' results are compared for consistency. Duplication is widely used in a variety of distributed computing system designs, such as volunteer computing [52] and MapReduce [53].

There are many more complex techniques for verification of volunteer computing results. For example, sampling approaches [54], [55], [56] verify each client by sending a task unit with a known correct result. If the result from a client doesn't match the correct result, the client will be blacklisted. Credibility approaches [52], [57] maintain reputation scores for each client by observing the behavior of the clients. These approaches overcome the drawbacks of the extra computation costs of task duplication, but are vulnerable to some specific attacks: a client can behave well at first to gain credibility and send falsified results after that. Distributed agreement and consensus algorithms, such as Byzantine agreement [58], are also possible but not ideally suited to an environment with a centralized server and no communication between clients.

A simple but effective approach to handle malicious clients is to use a task duplication and majority voting scheme. More complex strategies exist, such as credibility-based approaches, but are not necessarily a good fit for the highly-volatile browser-based data processing environment. Many visitors may access a website only once and the website may have no history statistics to derive their visitor reputation scores. Suppose the overall application consists of n divisible task units. There are C concurrent clients and among them, M are malicious clients, which will send falsified results. f is the fraction of malicious clients among the total clients. Our duplication approach works by assigning identical task units to k different clients and verifying the results returned from different clients. If the computation results from different clients are the same, then the result will be accepted. If not, a majority voting strategy is applied and the result from the majority of clients is accepted. The server randomly distributes tasks to clients, while ensuring no client receives both a task and its duplicate. Thus, clients have no control over which task units they will receive. The duplication approach fails only when a majority of a task's duplications are sent to a collaborative group of malicious clients.

Error Rate: The error rate ϵ is defined as the fraction of the final accepted results that are malicious. As described by Sarmenta et al. [52], the error rate can be given by:

$$\epsilon = \sum_{j=d}^{2d-1} \binom{2d-1}{j} f^j (1-f)^{(2d-1-j)} \qquad (1)$$

where d is the minimum number of matching results needed for the majority voting scheme. For example, for $d = 2$, the server accepts a result when it gets two identical results. When the server gets two different results, it keeps reassigning the task until 2 identical results are received. Redundancy, R, is defined as the ratio of the number of assigned task units to the number of total task units. It can be proved that $R = d/(1-f)$.

Although simple, the duplication approach is robust even with small d. For large websites with millions of visitors, it is difficult for attackers to gain control over a large portion of the clients. Suppose a hacker has a botnet of a thousand machines, $f = 0.001$ for a website with a million total clients. As shown in Figure 3, even with $d = 2$, we can ensure 99.9999% of all the accepted results are correct. The choice of d is also dependent on the type of application. Some applications are generally less sensitive and even no duplication is acceptable since the falsified results do little harm.

3.5.1 Data Privacy

One potential security threat comes from data privacy in the storage service. In terms of gray computing, data privacy refers to who has access and can decipher the meaning of data sent out by the server and written by the clients. The topic of data privacy has been explored in various areas of software engineering [59], [60], [61], [62].

If the data is owned by the website, (e.g., if Amazon distributes the product images or reviews for image scaling

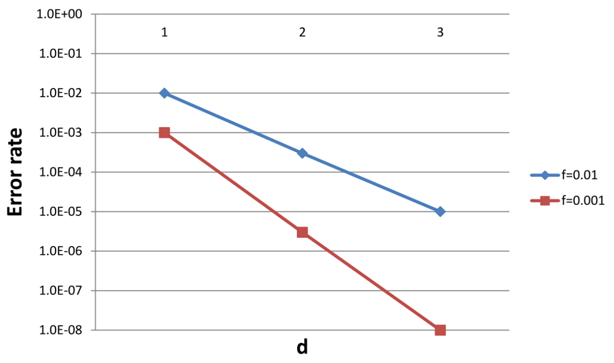


Fig. 3. Error rates for different parameter values of d and f.

or sentiment analysis, as we described in Section 4), there are no privacy concerns. Since these data is publicly available online, the use and sharing of this data should not raise any data privacy concerns. If the data is collected from users, then either an explicit permission should be requested from users, or some data anonymization approaches [63], [64] should be applied. However, as some work has shown [59], data anonymization alone is not enough. General guidelines regarding purpose limitation, data minimization, data security and transparency should be followed [59].

Sometimes, even if the data is not very sensitive, the data owner may still want to disclose only individual records instead of the whole dataset. If all clients can read all the data in the storage service, the website cannot keep its dataset private. Most web-based storage providers, however, offer a mechanism for generating temporary URLs for both reading and writing data to specific files or buckets within the storage service. For example, on Amazon S3, the task distribution server can generate a temporary URL to allow a client to read only the subset of the data items needed for its computational task. The temporary URL can be time-bounded to allow only a short period of access to prevent later sharing of the URL and access of the data by another client. Moreover, the URLs can encode permissions, allowing a client to write data to the storage service without other clients being able to read the result. Our approach uses these temporary URLs to distribute data and allow clients to report results directly back to the storage service. However, even with temporary URLs, it is still possible that malicious clients could coordinate to obtain a large chunk of the dataset.

3.6 A Cost Model for Gray Computing

Another challenge that must be dealt with is the cost-effectiveness of gray computing. To assign tasks to a large number of browser clients, a coordinating server is needed to send the input data to clients and collect the computing results from them. The extra data transferred, in addition to the original web page, will consume additional bandwidth and processing power of the server. It is possible that this extra cost can be more than the value of the results computed by the clients for some types of applications. Therefore, we need a cost model to help developers to decide whether it is cost-effective to deploy an application through gray computing and estimate the cost savings.

In our cost model, we use the Amazon EC2 cloud computing service as our benchmark. We chose Amazon's EC2 because it is a cost-effective and highly scalable cloud platform to process big data. Additionally, cloud computing has a quantifiable computing cost that we can compare against. For example, Amazon provides Map/Reduce services whose cost is based on the type and time of the computing resources that are consumed. Note that our cost model is generic and could be parameterized with costs from any cloud provider, but currently we focus on Amazon EC2.

For our gray computing distributed data processing engine, the cost can be broken down as:

$$\begin{aligned}C_{browser} =& C_{transfer} + C_{request} + C_{distribute}\\ =& C_{toclient} + C_{toserver} + C_{request} + C_{distribute}\\ \approx& C_{toclient} + C_{toserver} + C_{request}\end{aligned}$$

$C_{transfer}$ is the data transfer cost calculated on volume of data; $C_{request}$ is the HTTP request cost incurred based on the number of GET or POST requests; $C_{distribute}$ is the virtual compute units consumed by task distribution. We have shown in our empirical analysis that $C_{distribute}$ is nearly zero for our architecture because the data is served directly out of the cloud storage service and the task distribution server only needs to do a small amount of work.

3.6.1 $C_{distribute}$ Cost Analysis

As illustrated in Figure 1, a task distribution server is needed to dispatch tasks to all the web clients. This server accepts requests from clients and returns the brief task information (e.g., the task ID and task input data location in S3), so that clients can then fetch the correct data in S3 separately. Obviously, the more concurrent clients there are, the higher the throughput the server needs to support.

Our reference implementation architecture, described at the start of Section 3, reduces the load on the task distribution server by serving data for processing directly out of the cloud-based storage system. The task distribution server is not involved in serving the large volumes of data that need to be processed or storing the results. To assess the load produced by this architecture, we conducted an experiment to estimate the cost of hosting a task distribution server.

The task distribution server that we implemented exposes an HTTP API to handle HTTP GET requests and responds with a JSON-based string containing the task ID and data location URL for each task. Each client calls the API once before working on an individual task. In order to estimate the number of servers needed to support a given task distribution throughput, we performed a number of load stress tests against different types of general-purpose EC2 servers and calculated the peak throughput each server can support.

As shown in Table 11, m1.large has the best performance/cost ratio. Facebook received about 100 billion hits everyday [65], which is about 4.167 billion hits per hour. Every page hit triggers a task request, so the task dispatching server fleet to handle this traffic load needs to support a throughput of 4.167 billion requests / hour, or 1.16 million requests / second. Using the m1.large server type, a total of 144 servers are required to support such a high throughput, which costs about \$34.56 / hour.

TABLE 11
Peak task distribution throughput and cost of different EC2 instance types.

EC2 Inst. Type	Throughput	Cost	Price/Throughput
t1.micro	450 tasks/sec	\$0.02/hr	\$1.23E-8
m1.small	1560 tasks/sec	\$0.06/hr	\$1.06E-8
m1.medium	3300 tasks/sec	\$0.12/hr	\$1.01E-8
m1.large	8100 tasks/sec	\$0.24/hr	\$8.23E-9

The key question is the relative expense of the task distribution compared to the cost of distributing the task data. With such a large volume of page visits, we assume that each client requests 100KB input data from S3 (100KB is a small and conservative estimate considering a lot of web page resources such as images are over 100KB). The larger the amount of data per client, the more insignificant the task distribution cost is relative to the data transfer costs. The total amount of data to be transferred out every hour will be 4.167 billion * 100KB = 388 TB. Based on current S3 pricing [66], this will cost about \$37,752 / hour. It can be seen that the cost of hosting the dispatching server only accounts for about 0.09% of the total cost to handle the clients. Because of the small task distribution cost, we ignore it in the use case analyses presented later.

$C_{transfer}$ can be further broken down into $C_{toclient}$, $C_{toserver}$ and C_{code}.

$$\begin{aligned}C_{toclient} =& k * I * P_{transferout}\\ =& k * I * (P_{OriginToCDN} + P_{CDNout})\\ C_{toserver} =& O * P_{transferin}\\ =& O * (P_{toCDN} + P_{toOrigin})\\ C_{code} =& n{*}I_{code} * P_{transferout}\end{aligned} \tag{2}$$

$C_{toclient}$ is the cost to fetch input data for computing from the S3 storage server; I is the original size of input data to be transferred to clients for processing; the actual volume of data transferred will be k times of I as described in Equation 3; $P_{OriginToCDN}$ is the unit price to transfer data from the origin server (S3 in our case) to CDN servers (CloudFront in our case); P_{CDNout} is the unit price to transfer data from CDN servers to the clients. $C_{toserver}$ is the cost to return computation results from clients to the storage server; O is the size of data to be returned; P_{toCDN} is the unit price to transfer data from clients to CDN servers and $P_{toOrigin}$ is the unit price to transfer data from CDN servers to the origin server. C_{code} is the bandwidth cost to distribute the processing logic code, such as hashing algorithms in rainbow table generation, to the browser clients. n is the total number of user visits to the website. During each visit, the client may request and complete multiple tasks, but only one piece of processing code needs to be transferred. I_{code} is the size of the processing code in JavaScript. $P_{transferout}$ is the unit price of bandwidth cost to transfer data from server to client.

The actual data transferred from server to client will be more than the original size of input data I. One reason is described in Section 3.5, the same task needs to be sent to d different clients to be robust to malicious clients. Another reason is the clients may leave before the computation finishes. The data transferred to these clients is wasted. The actual volume of data transferred out will be k times the data needed for one copy of the task, where

$$\begin{aligned} k =& d + \sum_{n=1}^{\infty} d(1-\mu)^n \\ =& d + d*(1-\mu)/\mu = d/\mu \end{aligned} \tag{3}$$

d is the duplication factor. The variable μ is the successful task completion ratio of browser clients (*i.e.*, the percentage of distributed tasks that are processed successfully before a browser client leaves the site).

In Section 3.7, we discuss the estimation of value μ and its relationship with average page view duration, task granularity, and task distribution algorithms.

$$C_{request} = (k+d)*n*P_{request} \tag{4}$$

n is the number of tasks that need to be processed. $P_{request}$ is the unit price of HTTP requests to the CDN server. For each task distribution session, the client needs one HTTP request to fetch the data and one HTTP request to return the results. Since each task needs to be duplicated d times and not all the tasks are completed successfully, more fetch requests are needed than return requests. That is, kn requests to fetch input data and dn requests to return the results.

The cost to run the tasks in the cloud is given by C_{cloud}:

$$C_{cloud} = T_{cloud} * I * P_{unit} \tag{5}$$

where P_{unit} is the cost per hour of a virtual compute unit. For example, Amazon provides an "ECU" virtual computing unit measure that is used to rate the processing power of each virtual machine type in Amazon EC2. Virtual compute units are an abstraction of computing ability. T_{cloud} is the computing time to process 1 unit of data with one virtual compute unit in the cloud.

The distributed data processing engine built on gray computing is only cost effective when:

$$C_{cloud} > C_{browser} \tag{6}$$

That is, the cost to distribute and process the data in the browsers must be cheaper than the cost to process the data in the cloud.

$$\begin{aligned} T_{cloud} * I * P_{unit} >& k * I * (P_{OriginToCDN} + P_{CDNout}) \\ &+ O * (P_{toCDN} + P_{toOrigin}) \\ &+ n * I_{code} * P_{transferout} \\ &+ (k+d) * n * P_{request} \end{aligned} \tag{7}$$

Because prices are all constant for a given cloud service provider, the key parameters here are T_{cloud}, which can be computed by a benchmarking process, and k, which can be computed based on the voting scheme for accepting results and average page view time of clients.

The cost saving U is defined as:

$$U = C_{cloud} - C_{browser} \tag{8}$$

A positive U indicates an application is suitable for gray computing.

The benchmarking process to examine whether an application is cost-effective is as follows:

1) Start a cloud computing instance.
2) Put I GB of input data on the node.
3) Launch the data processing task.
4) Measure the time, t, to process the I input data.
5) If $t/I > k * P_{transferout}/P_{unit}$, the computation is worth distributing to web clients.

For some computational tasks, there may be a time requirement. For example, a daily computational task will need to be completed within 24hrs. If the computation needs to be finished within a time bound B, the process to determine the cost-effectiveness would be:

1) Run the process above.
2) Let $C = \lceil t/B \rceil$.
3) If $C/I > k * P_{transferout}/P_{unit}$, assuming there are at least C simultaneous clients, then the computation is worth distributing to web clients.

3.7 An Adaptive Scheduling Algorithm for Gray Computing

The clients in gray computing are website visitors' browsers, which are not static and reliable. The clients may join and leave the available computational resource pool frequently. The result is that a client may leave before its assigned computational task is finished, adding extra data transfer and producing no computational value. We define μ as the successful task completion ratio. The value of μ is important and directly influences the cost of gray computing. The higher value of μ, the less data transferred is wasted, and thus more cost-effective gray computing will be.

There are two factors affecting the successful task completion ratio μ: the page view duration of the client and the computing time of the assigned task. The relationship between μ, average page view duration and task sizes is depicted in Figure 4. Assume assigned task size is proportional to the computing time needed. For a fixed task chunk size, the longer the page view duration, the fewer chances the client will leave before the computation completed. The distribution of the page view durations of a website's clients is determined by the website's own characteristic. However, the computing time of the assigned tasks is what we can control. Assume the whole application can be divided into smaller chunks of arbitrary size. Reducing the single task size assigned to the clients will increase the task completion ratio, but result in more task units. More task units means more cost on the requests to fetch data and return results. Thus, there is a tradeoff between the size of tasks and the number of tasks.

Instead of treating all the clients as the same and assigning them tasks of the same size, we developed an adaptive scheduling algorithm. We utilize the fact that website visitors' dwell time follows a Weibull distribution [67], [68]. The probability density function of the Weibull distribution is given by:

$$f(t|k,\lambda) = \frac{k}{\lambda}(\frac{t}{\lambda})^{k-1}e^{-(\frac{t}{\lambda})^k} \quad t \geq 0 \tag{9}$$

In the probability density function, if $k > 1$, the Weibull distribution has the property of positive aging, which means the longer the object has survived, the more likely it is to fail. If $0 < k < 1$, the Weibull distribution has the property of negative aging, which means the longer the object has

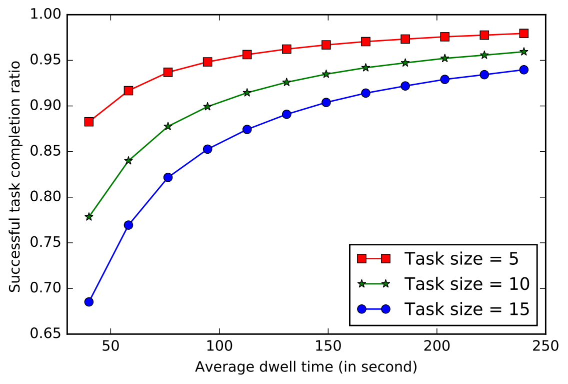


Fig. 4. The relationship between average page view duration and μ for different task sizes.

survived, the less likely it is to fail. Researchers have discovered from real-world data that 99% of web pages have a negative aging effect [67]. This implies the probability that a visitor leaves a web page decreases as time passes. This can be explained by the fact that visitors usually have a quick glance of a web page to see whether they are interested in the content or not. During this stage, the probability of leaving the page is high, but after the visitor passes this stage, they may find the content interesting and are willing to spend time dwelling on the page. Thus, the probability of leaving becomes lower.

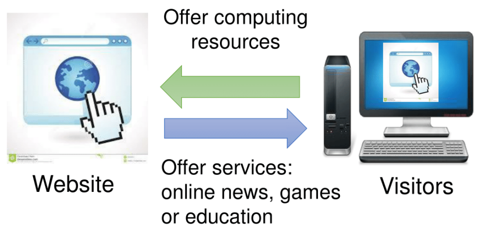
Inspired by the Weibull distribution of page dwell time, we propose an adaptive scheduling algorithm which dynamically adjusts the subsequent task size assigned to the same client according to the dwell time of the client. Instead of partitioning the task into equally sized chunks, we assign task size of m_i to the i-th request from the same client where $m_i < m_{i+1}$ when $1 \leq i < n$. Our adaptive scheduling algorithm works by assigning tasks of smaller size to clients first and increasing the subsequent task sizes until a threshold is reached. In this way, we achieve higher successful time completion ratio than fixed task size while also reducing the number of task requests.

For comparison, we also implemented a Hadoop [69] scheduler, which works as follows: The entire job is partitioned into task units of the same size. The server maintains a queue of tasks to assign. When a client connects to the server, the server randomly selects a task from the queue and assigns it to the client. If a task fails, the failed task will be pushed onto the back of the queue and wait to be reassigned to other clients.

To derive μ values for the schedulers, we ran simulations in Matlab with varying average page view times and task durations. We assumed the page view duration follows a Weibull distribution with average page view duration equal to T_{apv}. We used a duplication factor of $d = 2$ for a voting scheme that required 2 matching results. The results are shown in Table 12. We can see that 1) μ increases as the average page view time increases (*i.e.*, the successful task completion rate goes up), and 2) The adaptive scheduler achieves higher μ and faster task completion times compared to the Hadoop scheduler.

TABLE 12
Comparison of different schedulers in terms of task completion success ratio μ.

Scheduler	μ			
Average page view duration	30s	1min	2min	3min
Hadoop	0.80	0.85	0.88	0.90
Adaptive	0.84	0.88	0.92	0.93



Offer computing resources
Website
Offer services: online news, games or education
Visitors

Fig. 5. Computing can serve as an exchange medium.

3.8 Deployment Strategies

There are clearly significant legal and ethical questions around the concept of gray computing if computations happen without users knowing. To avoid legal and ethical issues, deployers need to obtain explicit consent of client users. This can be achieved easily by a pop-up agreement when users first visit the website or register for account. The difficult part is how to make users willing to offer their computing resources. In the following paragraphs, we provide several potential deployment strategies.

First, gray computing can be used to support scientific research, such as computational needs in biology, mathematics, and astronomy, as well as computational needs in the life sciences and medicine. Because such interests may be related closely to a visitor's interest, a visitor may be more willing to contribute to computing tasks of these research areas. The distributed computational tasks can be deployed on the websites of non-profit organizations such as environmental, public welfare, academic/educational organizations and government websites.

In addition to supporting scientific and medical research, gray computing can also be applied to general commercial websites that may offer reward incentives to motivate visitors to help with computational tasks. For instance, users visiting online retail sites can be asked if they are willing to participate in the computation process in order to earn shopping points in return. The rewards can also be in other forms besides monetary incentives. For example, for online web games, websites can reward players with virtual goods and money. Such rewards are attractive to players while at almost no cost to the websites. In fact, online games could be provided freely if those playing the games agree to participate in gray computing tasks. This could allow an online game vendor to be a broker for harnessing compute cycles for computational needs of clients who may pay for the service.

There are some other evidences supporting the application of gray computing. For example, Karame et al. [70] proposed the concept of microcomputing in which computing can serve as an exchange medium. Many online content providers, such as online newspapers, videos and games, offer services of small unit value but of large quantity.

Surveys have shown that users are reluctant to authorize credit card payments with small transactions due to complex procedures. Content providers are not willing to charge for small transactions as well because the cost of processing a small value transaction can be more than the profits earned [71]. However, content providers cannot always offer free services. They need to find a business model to generate revenue to sustain their online presence. Gray computing can serve as an exchange medium where the website visitors agree to perform computational tasks assigned by website owners while viewing the content for free, and the websites gain profits from the computational tasks accomplished by visitors while not disturbing the users, as illustrated in Figure 5.

Another attractive perspective of gray computing is it can serve as an alternative to online advertisements. Even though most websites seem to be free to view, websites commonly exploit visitors' computational resources to deliver advertisements, collect user statistics, and perform other tasks. Many websites have banners, pop-up windows for advertisements, or other content that utilize bandwidth or computing power of visitors' computers. Online advertisements can be annoying to users and may increase page loading time. If a website must exploit something from its visitors, background computing tasks can be a choice more acceptable than online advertisements because they are less visually intrusive and do not impact page loading time as we have shown in Section 3.4.1.

4 Examples of Gray Computing

In this evaluation section, we examine some practical applications of distributed data processing with gray computing. We compute the cost saving U for each application to find out which type of example applications are cost-effective in a gray computing model.

4.1 Experimental Environment

Browser client: We used a desktop computer for our experiments with an Intel Core i5-3570 CPU clocked at 3.5GHz and 8GB of memory. The computational tasks were implemented in JavaScript and executed on Mozilla Firefox 31.0.

Cloud instance: We used an Amazon EC2 m3.medium instance (ECU=3) and Ubuntu 14.04 64bit operating system for benchmarking cloud data processing performance.

Price for cloud computing: See Table 13 and 14. $P_{unit} = \$0.067/hour$. We chose a duplication factor $d = 2$ for most use cases, because it guarantees a relatively high accuracy as we have shown in Section 3.5. We also chose a conservative successful task completion ratio of $\mu = 0.8$, which is representative of the Hadoop scheduler efficiency with an average page view time of roughly 30s.

We compute the cost saving: $U = T_{cloud} * I * P_{unit} - C_{toclient} - C_{toserver} - C_{request}$ for each task, as shown in Table 15. A data processing task is cost-effective for distributed processing with browsers if U is a positive number.

TABLE 13
A Subset of Amazon Cloud Service Type and Price
(As of April 2017)

Service	Subtype	Price	ECU
EC2	t2.micro	\$0.013 /Hr	varied
	m3.medium	\$0.067 /Hr	3
	m3.large	\$0.133 /Hr	6.5
	c3.large	\$0.105 /Hr	7

TABLE 14
A Subset of Amazon Cloud Service Type and Price
(As of April 2017)

Service	Subtype	Price
S3	Transfer IN To S3	Free
	S3 to CloudFront	Free
CloudFront	To Origin	\$0.02/GB
	To Internet (First 10TB)	\$0.085/GB
	To Internet (Next 40TB)	\$0.08/GB
	To Internet (Next 100TB)	\$0.06/GB
	To Internet (Next 350TB)	\$0.04/GB
	HTTP requests	\$0.0075/per 10000

4.2 Use Cases

4.2.1 *Face detection*

Overview. For the first gray computing case study, we chose face detection, which is a data processing task that Facebook runs at scale on all of its photo uploads to facilitate user tagging in photos. Face detection is a task that most Facebook users would probably consider to be beneficial. Face detection also has an interesting characteristic in that it can be run on the data that is already being sent to clients as part of a web page (e.g., the photos being viewed on Facebook). Notice in our cost model, a large portion of the cost comes from sending input data from the server to clients. This cost becomes prohibitively expensive for data intensive tasks. Data transfer costs appear inevitable, since it is impossible for the clients to compute without input data. However, there are some circumstances when the input data needs to be sent to the client anyway, such as when the data is an integral part of the web page being served to the clients. In these cases, no extra data transfer costs are incurred. Facebook has 350 million photos uploaded every day [72]. Face detection in photos is a relatively computationally intensive task. Our hypothesis was that significant cost savings could be obtained through gray computing.

Experimental Setup. To test whether offloading face detection tasks to gray computing is cost-effective, we conducted experiments for face detection tasks on both JavaScript in the browser and OpenCV, which is a highly optimized C/C++ computer vision library, in the cloud. For

TABLE 15
Comparison of cost saving per GB for different tasks.

Task	Cost savings \$ per GB
Face detection	0.09
Image scaling	0.008
Sentiment analysis	0.065
Word count	-0.29
Rainbow table	∞
MSA	7.95
Protein disorder	194

the JavaScript facial detection algorithm, we used an open-source implementation of the Viola-Jones algorithm [73]. For the Amazon cloud computing implementation, we use the same algorithm but a high performance C implementation from OpenCV 2.4.8. There are more advanced face detection algorithms that achieve better accuracy, but need more computing time, which would favor browser-based data processing.

Empirical Results. The results of the experiment are shown in Table 16. We can see that the computation time is approximately linear to the total number of pixels or image size. This result is expected because Viola-Jones face detection uses a multi-scale detector to go over the entire image, which makes the search space proportional to the image dimensions.

Analysis of Results. Suppose the average resolution of the 350 million photos being uploaded is 2 million pixels, which is less than the average resolution of most smartphone cameras, such as the 12 million pixel (12 megapixel) iPhone 6S camera. It takes $1.7s*3.5*10^8/3600h = 165,278h$ of computing time for an EC2 m3.medium instance to process these photos. With our utility function $U = T_{cloud} * I * P_{unit} - 1/\mu * d * I * P_{transferout} - O * P_{transferin} - C_{code} - C_{request}$, where $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring photos to the clients. The only extra cost is transferring the face detection code. In our experiment, $I_{code} = 50KB$. The extra bandwidth consumed in a month is $350*10^6/5*50*10^{-6}*30 = 105TB$. $C_{code} = 0.085*10*1000+0.08*40*1000+0.06*55*1000 = \$7350$. For each photo, the returned computing result should be an array of coordinations of rectangles, which is rather small, so O can be ignored. The client does not need to fetch the photos but one HTTP request to return the results. We assume the client processes 5 photos at a time (which takes around 7s on a browser with average hardware). We chose a duplication factor $d = 1$ since this application is not very sensitive to security. Thus, $C_{request} = 350*10^6/5*0.0075/10^4 = 52.5$, and $U = 165278*0.067 - 7350 - 52.5 = \$3671$ could be saved each day by distributing this task to browser clients rather than running it in Amazon EC2. That is a yearly cost savings of roughly $1.34 million dollars. The actual algorithm Facebook uses is likely to be more complex than the algorithm we used and a larger T_{cloud} is expected. Therefore, the cost savings could be even larger than we calculated.

4.2.2 *Image Scaling*

Overview. Another example of a useful gray computation that can be applied to data already being delivered to clients is image scaling. Websites often scale image resources, such as product images, to a variety of sizes in order to adapt them to various devices. For instance, desktop browsers may load the image at its original size and quality, while mobile clients with smaller screens may load the compressed image with lower quality. The Amazon Kindle Browser talks directly to EC2 rather than target websites and receives cached mobile-optimized versions of the site and images that are produced from large-scale image processing jobs.

We can offload image compression tasks to gray computing. Similar to face detection, the photos are already being delivered to the clients, so there is no added cost for transferring data from the server to clients. After loading the images, each client will do the scaling and compression work and then send the scaled images for mobile devices back to the server. When the number of images to process is large, the saved computation cost could be substantial.

Experimental Setup. There are many image scaling algorithms and they achieve different compression qualities with different time consumptions. We wanted to measure JavaScript's computation speed on this task and did not want to focus on the differences in efficiency of the algorithms. We chose bicubic interpolation [74] for image scaling on both the browser and server. Bicubic interpolation is implemented in JavaScript and C++ for both cloud and browser platforms.

Empirical Results. The experiment results in terms of computation speed are shown in Table 17.

Analysis of Results. To obtain a quantitative understanding of the cost savings by offloading the image scaling tasks to gray computing, we again use Facebook as an example and make several assumptions: suppose the average resolution of the photos being uploaded is 2 million pixels and the average scaling ratio is 50%. 350 million uploaded photos daily take $0.55s*3.5*10^8/3600h = 53,472h$ for one EC2 m3.medium instance to scale and compress. With our utility function $U = T_{cloud}*I*P_{unit} - k*I*P_{transferout} - O*P_{transferin} - C_{code} - C_{request}$. Again $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring images to the clients. The only extra cost is transferring the image scaling algorithm code. In our experiment, $I_{code} = 3KB$. The extra bandwidth consumed in a month is $350*10^6/5*3*10^-6*30 = 6.3TB$. $C_{code} = 0.085*6.3*1000 = \$535.5$. We assume the client processes 10 photos at a time (which takes around 10s on a browser with average hardware). We chose a duplication factor $d = 1$ since this application is not very sensitive to security. $O = 3.5*10^8*80KB/10^6 = 2.8*10^4 GB$. $C_{request} = 3.5*10^8/10*0.0075/10^4 = 26.25$. Therefore, $U = 53472*0.067 - 2.8*10^4*0.02 - 535.5 - 26.25 = \$2461$ is saved each day by distributing this task to browsers, which aggregates to $898,219 a year.

There are many techniques that can be used to further improve the quality of the resized images, such as sharpening, filtering, etc. These techniques require additional computation time (larger T_{cloud}), so the amount of money saved could be further increased if these techniques are applied to uploaded images.

4.2.3 *Sentiment Analysis*

Overview. Sentiment analysis [75] refers to using techniques from natural language processing and text analysis to identify the attitude of a writer with respect to some source materials. For instance, Amazon allows customers to review the products they purchased. It would be useful to automatically identify the positive/negative comments and rank the products based on the customers attitude. Since the comment context is sent to the websites' visitors anyway, there will be no extra cost incurred due to the data transferred from servers to clients.

Experimental Setup. There are many machine learning algorithms proposed for sentiment analysis in the liter-

TABLE 16
Computing Time Comparison for Face Detection Tasks.

Image Dimension	Number of pixels	Size	JS Computing Time(s)	EC2 Computing Time(s)
960*720	0.69 million	82KB	0.67	0.56
1000*1500	1.5 million	156KB	1.05	1.30
1960*1300	2.55 million	277KB	1.70	2.15

TABLE 17
Computing Time Comparison for Image Resize Tasks.

Image Dimension	Size	JavaScript Computing Time(s)			EC2 Computing time(s)		
		Scaling ratio 30%	50%	70%	30%	50%	70%
960*720	82KB	0.14	0.35	0.66	0.18	0.22	0.25
1000*1500	156KB	0.29	0.69	1.34	0.43	0.45	0.58
1960*1300	277KB	0.48	1.2	2.08	0.66	0.73	0.86

ature [75]. We implemented a Naive Bayes Classifier, a simple but quite effective approach, for sentiment analysis. The classifier takes as input a user review and predicts whether the review is positive or negative. The classifier is implemented with JavaScript for the browsers and Python for EC2. We trained our classifier with a movie review dataset [76] containing 1000 positive and 1000 negative reviews. We collected movie reviews from the Internet as test data and partitioned them into 10 files each of size 200KB. Then we used the trained classifier to predict the attitude of each review item and recorded the time needed.

Empirical Results. For an input size of 200KB, the prediction time of browsers with JavaScript is 0.3s and for the same task running on an EC2 m3.medium instance, the prediction time is 0.7s.

Analysis of Results. For 1GB of input data, the cost saving is $1000/0.2 * 0.7/3600 * 0.067 = \$0.065$. The size of sentiment analysis code I_{code} is 5KB. To obtain a quantitative understanding of how much money can be saved by offloading the sentiment analysis tasks to gray computing, we use Facebook as an example and make several assumptions: suppose each user uploaded photo has an average of 1Kb comments. Since Facebook has 350 million photos uploaded daily, there is $350 * 10^6 * 1/10^6 = 350GB$ comments in total. If Facebook wants to analyze the attitude of every comment of the photos, the cost is $350 * 0.065 * 365 = \$8304$ a year.

4.2.4 *Word count*

Overview. Word counting is the classic use case that is used to demonstrate Big Data processing with MapReduce. Word counting requires determining the total occurrence of each word in a group of web pages. It is a task that requires a relatively large amount of textual input with a very small amount of computational work.

Experimental Setup. We compared the cost-effectiveness of running a word count task in JavaScript versus the Amazon Elastic Map Reduce (EMR) service (using a script written in Python). The Amazon EMR experiment was configured with 1 master node (m1.medium) and 2 slave nodes (m1.medium).

Empirical Results. The experiment results in terms of computing speed are shown in Table 18. For 1GB of input, $C_{request}$ is very small, the cost saving is $1/0.38*6/60*0.109*2 - 0.14 * 2.5 = \$ - 0.29$.

Analysis of Results. As can be seen, the cost saving is a negative number which means the value of the computed results is less than the cost of transferring the text data to the clients. Word counting is not an ideal application for distributed data processing with gray computing because it is a data intensive but computationally simple task. However, if the word count was being run on web pages delivered to the clients, the data transfer cost would not be incurred and it would be as cost effective as in Section 4.2.1 and 4.2.2.

TABLE 18
Computing Time Comparison for Wordcount.

Input size	Browser(JavaScript)	Amazon EMR
18MB	13.7s	94s
380MB	3min	6min

4.2.5 *Rainbow Table Generation*

Overview. Another use case was designed to analyse the gray computing power that could be wielded by an attacker if they compromised a large number of websites and began distributing malicious data processing tasks in their web pages. For the malicious use case, we focused on password cracking with rainbow tables, which has become a common type of work performed by cyber-attackers on data stolen from websites. A rainbow table [77] is a precomputed table for reversing cryptographic hash functions, usually for cracking password hashes. These tables are used to recover the plaintext passwords that are stored as hashes in databases.

To distribute the tasks, the central server only needs to send a start string and end string of a range of the password to the clients. The size of input is extremely small and can almost be ignored. Therefore, $C_{toclient} = 0$.

Experimental Setup. The browser environment is the same as described in Section 4.1. For the cloud instance, we implemented the rainbow table generation algorithm in C++ and compiled it with g++ 4.8.2.

Empirical Results. The results of the experiment are shown in Table 19. Generating 900,000 hashes on a desktop's browser using ported JavaScript took 4s. Generating the same hashes on an Amazon EC2 m3.medium using C++ took 3s.

Analysis of Results. Popular websites have been found to be deliberately hijacked to embed malicious JavaScript. For example, Forbes.com was compromised in November 2014 [78]. Consider the traffic of Forbes.com, which is about

TABLE 19
Computing Time Comparison for Generating a Rainbow Table.

Input size	Browser(native JS)	EC2(c++)	Browser(ported JS)
9E5	90s	3s	4s

4.4 million unique IP visits per day and 3 minutes average dwell time on the site [5]. Suppose a task duplication factor of 2 and task successful ratio of 0.8, the attacker would be able to wield the equivalent of $4.4 * 10^6 * 0.05hour * 0.8/2 * 365day = 3.212 * 10^7$ computing hours a year with browsers. To accomplish the same task with an Amazon EC2 m3.medium instance for a year, the yearly cost would be $3.212*10^7/4*3*0.067\$/hour = \$1,614,030$. We assume the average time for a client to process one task is 10 seconds. The client processes 18 tasks during its visit on average. $C_{request} = (2.5 + 2) * 4.4 * 10^6 * 18 * 365 * 0.0075/10^4 = \$97,564$. $C_{code} = n * I_{code} * Ptransferout = 4.4 * 10^6 * 113Kb * 10^-6 * \$0.085/GB = \$42.26$. With our utility function $U = C_{cloud} - C_{toserver} - C_{code} - C_{request}$, the yearly cost saving U is around \$1.5 million. The near-native computing speed of JavaScript in the browser and the low data transferred per unit of computational work makes this an effective processing task to distribute.

4.2.6 *Pairwise Sequence Alignment with the Smith-Waterman Algorithm*

Overview. In bioinformatics, sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Designing high-performance algorithms for aligning biological sequences is a challenging problem. Biological sequences are made up of residues. In DNA sequences, these residues are nucleic acids (e.g.,ATCG), while in protein sequences, these residues are amino acids.

DNA replication errors cause substitutions, insertions, and deletions of nucleotides, leading to edited DNA texts. Since DNA sequences are subject to insertions and deletions, we cannot directly compare the i-th symbol in one DNA sequence to the i-th symbol in the other. Instead, Edit distance is used. Edit distance is defined as the distance between two strings as the minimum number of editing operations needed to transform one string into another, where the edit operations are insertion of a symbol, deletion of a symbol, and substitution of one symbol for another.

The Smith-Waterman algorithm is a well-known dynamic programming algorithm for performing local sequence alignment for determining similar regions between two DNA or protein sequences. The algorithm was first proposed by T. Smith and M. Waterman in 1981 [12]. One main drawback of the SmithWaterman algorithm is its high time complexity: To align two sequences with lengths m and n, $O(mn)$ time is required. Many algorithms (*e.g.*, BLAST [79] which uses a heuristic algorithm) tried to improve the speed, but at the cost of accuracy.

While BLAST is faster than Smith-Waterman, it cannot guarantee the optimal alignments of the query and database sequences as Smith-Waterman, so some matches between query sequence and database sequence may be missed. Thus, the Smith-Waterman algorithm still performs significantly better than BLAST in terms of accuracy [80], [81].

Experimental Setup. We used SSEARCH36.0, which is the Smith-Waterman algorithm implementation from the FASTA sequence analysis package [82] for the cloud computing test. The JavaScript implementation was obtained by porting the BioJava 3.0 implementation to JavaScript using the Google Web Toolkit 2.6.

Empirical Results. The experiment results can be found in Table 20.

TABLE 20
Computation Time of Pairwise Sequence Alignment (S1:query sequence, S2:reference sequence).

S1 Length	S2 Length	ssearch36 (cloud)	JavaScript
2096	3583	1.84s	5.79s
993	9904	2.8s	9.24s
2096	9904	5.3s	16.8s

Analysis of Results. It can be seen that the computation time is approximately linear to the length of the query sequence and reference sequence. To get a quantitative understanding of the cost savings, we assume that a research lab wants to align 10^7bp (10MB) of newly produced sequences. They need to align these sequences to a database of 10^{10}bp (10GB) to identify the function of the sequence. Suppose the average length of a produced sequence is 1000 residues (about 1KB), and we partition the database into chunks of roughly 50KB. The cost to run this task using cloud servers is: $14s/3600 * 0.044 * 10^5 * 2 * 10^4 = \$342,000$. The cost to run this task with browsers given the duplication factor $d = 2$ is: $0.12 * 2.5 * 60 * 2 * 10^5 * 10^3/10^6 = \$3,600$. Because the search is done on a large dataset and the best match can happen in any place, even when there are small amounts of malicious clients, the probability they affect the final result is very small. Therefore, the duplication is actually not necessary for this task and the cost can be further reduced to $0.12 * 60 * 2 * 10^5 * 10^3/10^6 = \$1,440$. We can see that the cost to run a Smith-Waterman search on a large dataset is prohibitive for normal labs, while the cost is acceptable with a distributed browser-based approach.

4.2.7 *Multiple Sequence Alignment*

Overview. Multiple Sequence Alignment (MSA) is a sequence alignment of three or more biological sequences. MSA is important in many applications such as phylogenetic tree construction and structure prediction. Finding the global optimum of MSA for n sequences has been proven to be an NP-complete problem [13]. Heuristic approaches try to find the multiple alignment that optimizes the sum of the pairwise alignment score. Progressive methods first estimate a tree, then construct a pairwise alignment of the subtrees found at each internal node.

MUSCLE (MUltiple Sequence Comparison by Log-Expectation) [13] is one of the most widely-used and best-performing multiple alignment programs according to published benchmark tests, with accuracy and speed being consistently better than CLUSTALW [83].

Experimental Setup. We used the PREFAB3.0 database for our experiment. The database contains 1,932 alignment cases averaging 49 sequences of length 240. All the MSA test cases are below 45KB in file size, which is feasible for transferring through a web page. We used latest implementation of MUSCLE (3.8.31) [84] for the cloud computing test. The JavaScript version was obtained by compiling the source code of MUSCLE with Emscripten 1.1.6 64bit.

Empirical Results. We ran the muscle program on 100 alignments randomly chosen from the PREFAB dataset. The processing time on a m1.small instance on average was 0.675s/KB. For the JavaScript version, the processing time on the desktop on average was 2.13s/KB.

Analysis of Results. We apply the cost model to the experiment result we obtained. The cost savings per GB is: $10^6 * 0.675/3600 * 0.044 - 0.12 * 2/0.8 = \$7.95$. This shows the browser-based approach is more cost-effective than a cloud computing solution.

4.2.8 *Protein Disorder Prediction*

Overview. Many proteins contain regions that do not form well-defined 3-dimensional structures in their native states. The accurate recognition of these disordered regions is important in molecular biology for enzyme specificity studies, function recognition, and drug design. The detection of a disordered region is also essential in structural biology since disordered regions may affect solubility or crystallizability. RONN (Regional Order Neural Network) [85] is a software approach, using bio-basic function neural network pattern recognition, to detect natively disordered regions in proteins.

Experimental Setup. We acquired a local version of RONN in C++ from the RONN authors. The JavaScript version is obtained by compiling the source file in C++ with Emscripten 1.1.16. The input of the RONN algorithm is a sequence of proteins in FASTA format. We used 100 protein sequences with an average length of 5,124 residues from the PREFAB datasets.

Empirical Results. The processing time on a m1.small instance is an average of 15.9s/KB. For the JavaScript version, the processing time on a desktop is an average of 17.2s/KB.

Analysis of Results. We apply the cost model to the experiment results we obtained. The cost savings per GB is: $10^6 * 15.9/3600 * 0.044 - 0.12 * 2.5 = \$194$. This shows that a browser-based approach is more cost-effective than cloud computing for RONN.

4.3 Summary

Table 15 shows the comparison of cost saving per GB for different tasks. As we can see, computationally intensive tasks such as rainbow table generation and bioinformatics algorithms have very high cost saving per GB and thus are more suitable for gray computing. One characteristic of these tasks is they often involve repetitive calculation of multiple iterations. Some other promising problems include Monte-Carlo simulations, simulated annealing, evolutionary/genetic algorithms, etc. Face detection, image scaling and sentimental analysis tasks have moderate cost saving per GB, but considering the massive volume of the corresponding image/text data, the potential is still huge. For tasks such as word count, they are more data-intensive instead of computationally intensive. Only very simple processing has been done on the data. Therefore, it is not worth to distribute the data to some remote clients for processing.

5 Other Potential Issues & Challenges

In this section, we discuss some potential threats and challenges to the gray computing framework and the experimental results.

5.1 Mobile Devices

Gray computing was originally designed to work with web browsers on desktops. However, more and more website traffic is now coming from mobile devices. Mobile devices have different hardware and software environments that might render some of our assumptions of gray computing invalid. Although the gap has narrowed in recent years, mobile devices tend to have less processing power than desktops due to power consumption constraints and heat dissipation concerns. Also, mobile devices have limited battery capacities and data transfer allowances. So users may not be willing to contribute their processing power, even with incentives, because of battery life concerns and data plan charges.

Some browsing is now done through dedicated apps, such as the Facebook app, that access web content but through a potentially non-browser interface. Native apps in Android and iOS do not necessarily rely on JavaScript. Therefore we cannot offload any Web Worker tasks to these users. Although many apps now are hybrid and allow JavaScript to be run within an embedded WebView, the performance of WebViews varies. There can be performance degradation of different degrees due to the lack of optimizations in a WebView compared to browsers as we have shown in Section 3.3.

5.2 Script Blocking Tools

There are some browser plug-ins (e.g., NoScript, AdBlock Plus) that can block all background scripts or only allow scripts from trusted domains. Users may install these plug-ins for security and performance purposes. If users are running these plug-ins while visiting the websites where gray computing is deployed, the assigned tasks will not be executed in the clients. The bandwidth cost for data transfer will be wasted.

However, the use of NoScript is not pervasive (about 2 million users [86]). Even considering other similar plugins, the percentage of users who disable JavaScript is less than 1% out of all the Internet users [87]. This is because JavaScript is used extensively in web pages and blocking JavaScript can cause many web pages to not display or function properly. Although plugins like NoScript can configure a blocking policy per site, it requires manual intervention and is not common for the majority of traditional Internet users.

5.3 User Acceptance

One common concern about gray computing is the fear that some users express of a foreign program running on their computers. Users may feel insecure when they think that some external script is running while they are browsing. However, this is no different than any current JavaScript that is executing within a browser. Browsers already assume that all web pages could have malicious JavaScript content and sandbox the code to protect users. In addition, many dynamic and interactive ads also contain JavaScript and few users actually go and check the ad's source code. Gray computing does not increase the risk profile any more than ads, which also contain JavaScript tasks and could be considered a vehicle for delivering gray computing.

6 Related Work

6.1 Volunteer Computing

The term 'volunteer computing' was first used by Luis F. G. Sarmentan [88]. He developed a Java applet-based volunteer computing system called Bayanihan [88] in 1998. In his paper, several challenges and corresponding solutions for volunteer computing paradigms were discussed such as adaptive parallelism, fault-tolerance, and performance on scalability. The growth in the number of Internet users at that time showed the immense potential of this concept and thus received wide interest. Other researchers have investigated Java-based clients for volunteer computing, such as Unicorn [89].

Volunteer computing utilizes idle computational resources from the general public. SETI@Home [90] was one of the earliest projects to prove the practicality of the volunteer computing concept. SETI@Home was designed to use the numerous personal computers of the public to process vast amounts of telescope signals to search for extraterrestrial intelligence. Some of SETI@Home's successors include Folding@Home [91], which simulates protein folding for disease research, and BOINC [15], which is a platform to hold various themes of research projects. What these projects share in common is that they all focus on non-profit scientific computing and they all require users to install a specialized client-side software in order to participate in the projects. Whereas in our paper, we analyze the highly volatile browser-based data processing domain and extend beyond scientific computing to website business operations.

6.2 Browser-based Computing

A number of researchers have investigated volunteer computing with browsers. The primary advantage of a browser-based approach is that the user only needs to open a web page to take part in the computation. Krupa et al. [92] proposed a browser-based volunteer computing architecture for web crawling. Konishi et al. [93] evaluated browser-based computing with Ajax and the comparative performance of JavaScript and legacy computer languages. In our work, we add a comprehensive analysis of: 1) the architectural changes to optimize this paradigm for websites served from cloud environments; 2) analysis of the impact of page-view time on scheduler efficiency and data transfer; 3) a cost model for assessing the cost-effectiveness of distributing a given task to browser-based clients as opposed to running the computation in the cloud; and 4) present a number of practical examples of data processing tasks to support website operators, particularly social networks, e-commerce, as well as life science.

Some researchers have noticed the potential of browser-based distributed computing and applied the infrastructure to volunteer computing. QMachine [94] is a web service incorporating volunteer web browsers worldwide together for bioinformatics computing tasks. It requires the explicit permissions from participating users. None of the existing work has discussed the cost model of browser-based distributed computing and whether there are really monetary incentives to perform such computing. Additionally, if there is monetary incentives, what are the factors that affect the profit of the computation?

MapReduce [1] is a programming model for large parallel data processing proposed by Google, which has been adapted to various distributed computing environments such as volunteer computing [95]. There are some early prototypes implementing MapReduce with JavaScript [16]. Lin et al. [95] observed that the traditional MapReduce is proposed for homogeneous cluster environments and performs poorly on volunteer computing systems where computing nodes are volatile and with a high rate of unavailability, as we also demonstrated with our derivation of μ in Section 3.7. They propose MOON (MapReduce On Opportunistic eNvironments) which extends Hadoop with adaptive task and data scheduling algorithms and achieves a 3-fold performance improvement. However, MOON targets institutional intranet environments like student labs where computer nodes are connected with a local network with high bandwidth and low latency. We focus on cloud and Internet environments with highly heterogeneous computation ability, widely varying client participation times, and non-fixed costs for data transfer and task distribution resources.

Since the execution of Web Workers does not need explicit permission from users, browser-based computing could be misused to develop a form of "resource stealing" attack. Pan et al. [96] evaluate the security risk of offloading computational tasks including password cracking and DDoS attacks to unwitting websites' visitors.

7 Conclusion

Every day, millions of users opt into allowing websites to use their browsers' computing resources to perform computational tasks, such as form validation. In this article, we explore the feasibility, cost-effectiveness, user experience impact, and architectural optimizations for leveraging the browsers of website visitors for more intensive distributed data processing, which we term gray computing. Although previous research has demonstrated it is possible to build distributed data processing systems with browsers when the web visitors explicitly opt into the computational tasks that they perform, no detailed analysis has been done regarding the computational power, user impact, and cost-effectiveness of these systems when they rely on casual website visitors. The empirical results from performing a variety of gray computing tasks, ranging from face detection to sentiment analysis, show that there is significant computational power in gray computing and large financial incentives to exploit its use. Due to these incentives and the vast potential for misuse, we believe that much more research is needed into the security and ethical considerations around gray computing.

As part of the analysis in this article, we derived a cost model that can be used to assess the suitability of different tasks for distributed data processing with gray computing. This cost model can aid future discussions of legitimately incentivizing users to opt-into gray computing. Further, we pinpoint the key factors that determine whether a task is suitable for gray computing and provide a process for assessing the suitability of new data processing task types, which can help in guiding the design of gray computing systems and user incentive programs. We also present a number of architectural solutions that can be employed to exploit cost and performance asymmetries in cloud environments to improve the cost-effectiveness of gray computing for legitimate uses.

We believe that software engineers will find additional compelling reasons for adopting gray computing as a platform for development of practical applications. We envision a business model that can be based on gray computing to utilize idle computing resources, which can bring benefit to both website owners and visitors. We also hope to raise concerns about the potential abuse of this computing model against website visitors and raise discussion on possible countermeasures in the future.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] L. F. G. Sarmenta, "Volunteer computing," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[3] D. P. Anderson, "Volunteer computing: the ultimate cloud." *ACM Crossroads*, vol. 16, no. 3, pp. 7–10, 2010.

[4] "Internet population," http://www.internetlivestats.com/internet-users/, [Online; accessed 27-September-2017].

[5] "Alexa," http://www.alexa.com/, [Online; accessed 27-September-2017].

[6] "BOINC," http://boinc.berkeley.edu/, [Online; accessed 27-September-2017].

[7] "SETI@home," http://setiathome.ssl.berkeley.edu/, [Online; accessed 27-September-2017].

[8] "Top500 supercomputer," http://www.top500.org/, [Online; accessed 27-September-2017].

[9] "EC2 Pricing," http://aws.amazon.com/ec2/pricing/, [Online; accessed 27-September-2017].

[10] L. F. Sarmenta and S. Hirano, "Bayanihan: Building and studying web-based volunteer computing systems using Java," *Future Generation Computer Systems*, vol. 15, no. 5, pp. 675–686, 1999.

[11] Y. Pan, J. White, Y. Sun, and J. Gray, "Gray computing: An analysis of computing with background javascript tasks," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 167–177.

[12] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[13] R. C. Edgar, "MUSCLE: multiple sequence alignment with high accuracy and high throughput," *Nucleic acids research*, vol. 32, no. 5, pp. 1792–1797, 2004.

[14] F. Smedberg, "Performance Analysis of JavaScript," Master's thesis, Linkoping University, Department of Computer and Information Science, 2010.

[15] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing*. IEEE, 2004, pp. 4–10.

[16] S. Ryza and T. Wall, "MRJS: A JavaScript MapReduce Framework for Web Browsers," *URL http://www.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf*, 2010.

[17] R. Cushing, G. H. H. Putra, S. Koulouzis, A. Belloum, M. Bubak, and C. De Laat, "Distributed computing on an ensemble of browsers," *Internet Computing*, vol. 17, no. 5, pp. 54–61, 2013.

[18] J.-J. Merelo, A. M. García, J. L. J. Laredo, J. Lupión, and F. Tricas, "Browser-based distributed evolutionary computation: performance and scaling behavior," in *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*. ACM, 2007, pp. 2851–2858.

[19] P. Langhans, C. Wieser, and F. Bry, "Crowdsourcing MapReduce: JSMapReduce," in *Proceedings of the 22nd international conference on World Wide Web companion*. International World Wide Web Conferences Steering Committee, 2013, pp. 253–256.

[20] F. T. Leighton and D. M. Lewin, "Content delivery network using edge-of-network servers for providing content delivery to a set of participating content providers," 2003, uS Patent 6,553,413.

[21] "Express," http://expressjs.com/, [Online; accessed 27-September-2017].

[22] "Unreal Engine in JavaScript," http://www.davevoyles.com/asm-js-and-webgl-for-unity-and-unreal-engine/, [Online; accessed 27-September-2017].

[23] "Octane," http://octane-benchmark.googlecode.com/svn/latest/index.html, [Online; accessed 27-September-2017].

[24] "Computer Language Benchmarks Game," http://benchmarksgame.alioth.debian.org/u32/benchmark.php?, [Online; accessed 27-September-2017].

[25] WebCL, "http://www.khronos.org/webcl/," [Online; accessed 27-September-2017].

[26] J. Duda and W. Dlubacz, "GPU acceleration for the web browser based evolutionary computing system," in *17th International Conference on System Theory, Control and Computing (ICSTCC)*. IEEE, 2013, pp. 751–756.

[27] "Emscripten," http://kripken.github.io/emscripten-site/index.html, [Online; accessed 27-September-2017].

[28] "Asm.js," http://asmjs.org/, [Online; accessed 27-September-2017].

[29] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 301–312.

[30] http://kripken.github.io/emscripten-site/docs/getting_started/FAQ.html, note = [Online; accessed 27-September-2017].

[31] "Google Web Toolkit," http://www.gwtproject.org/, [Online; accessed 27-September-2017].

[32] "Quake," https://code.google.com/p/quake2-gwt-port/, [Online; accessed 27-September-2017].

[33] "Eclipse Graphical Editing Framework," http://gefgwt.org/, [Online; accessed 27-September-2017].

[34] A. Puder, V. Woeltjen, and A. Zakai, "Cross-compiling java to javascript via tool-chaining," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2013, pp. 25–34.

[35] "pyjs," http://pyjs.org/, [Online; accessed 27-September-2017].

[36] "Opal," http://opalrb.org/, [Online; accessed 27-September-2017].

[37] "Closure compiler," https://developers.google.com/closure/compiler/, [Online; accessed 27-September-2017].

[38] http://blog.venturepact.com/8-high-performance-apps-you-never-knew-were-hybrid/, note = [Online; accessed 27-September-2017].

[39] "Battery Status API," https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API, note = [Online; accessed 27-September-2017].

[40] "Network Information API," https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API, note = [Online; accessed 27-September-2017].

[41] "HTML5," https://en.wikipedia.org/wiki/HTML5, note = [Online; accessed 27-September-2017].

[42] "Web Workers," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers, note = [Online; accessed 27-September-2017].

[43] Tampermonkey, "http://tampermonkey.net/," [Online; accessed 27-September-2017].

[44] https://greensock.com/js/speed.html/, note = [Online; accessed 27-September-2017].

[45] https://webperf.ninja/2015/jank-meter/, note = [Online; accessed 27-September-2017].

[46] https://technet.microsoft.com/en-us/library/cc749115(v=ws.11).aspx, note = [Online; accessed 21-September-2017].

[47] Greasemonkey, "http://www.greasespot.net/," [Online; accessed 27-September-2017].

[48] "Electroly," https://wiki.mozilla.org/Electrolysis/, [Online; accessed 27-September-2017].

[49] J. Cipar, M. D. Corner, and E. D. Berger, "Transparent contribution of memory," in *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*. USENIX Association, 2006, pp. 11–11.

[50] http://www.wordstream.com/blog/ws/2016/02/29/google-adwords-industry-benchmarks, note = [Online; accessed 27-September-2017].

[51] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello, "Characterizing result errors in internet desktop grids," in *Euro-Par 2007 Parallel Processing*. Springer, 2007, pp. 361–371.

[52] L. F. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002.

[53] W. Wei, J. Du, T. Yu, and X. Gu, "Securemr: A service integrity assurance framework for mapreduce," in *Annual Computer Security Applications Conference, 2009. ACSAC'09*. IEEE, 2009, pp. 73–82.

[54] S. Zhao, V. Lo, and C. G. Dickey, "Result verification and trust-based scheduling in peer-to-peer grids," in *Fifth IEEE International Conference on Peer-to-Peer Computing*. IEEE, 2005, pp. 31–38.

[55] W. Du, M. Murugesan, and J. Jia, "Uncheatable grid computing," in *Algorithms and theory of computation handbook*. Chapman & Hall/CRC, 2010, pp. 30–30.

[56] K. Watanabe, M. Fukushi, and S. Horiguchi, "Optimal spot-checking for computation time minimization in volunteer computing," *Journal of Grid Computing*, vol. 7, no. 4, pp. 575–600, 2009.

[57] K. Aberer and Z. Despotovic, "Managing trust in a peer-2-peer information system," in *Proceedings of the tenth international conference on Information and knowledge management*. ACM, 2001, pp. 310–317.

[58] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[59] Y. S. Van Der Sype and W. Maalej, "On lawful disclosure of personal user data: What should app developers do?" in *Requirements Engineering and Law (RELAW), 2014 IEEE 7th International Workshop on*. IEEE, 2014, pp. 25–34.

[60] S. Spiekermann and L. F. Cranor, "Engineering privacy," *IEEE Transactions on software engineering*, vol. 35, no. 1, pp. 67–82, 2009.

[61] I. Omoronyia, L. Cavallaro, M. Salehie, L. Pasquale, and B. Nuseibeh, "Engineering adaptive privacy: on the role of privacy awareness requirements," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 632–641.

[62] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 21–30.

[63] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[64] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 3, 2007.

[65] "Facebook hits," http://www.webpronews.com/facebook-gets-100-billion-hits-per-day-2010-07, [Online; accessed 27-September-2017].

[66] "S3 pricing," http://aws.amazon.com/s3/pricing/, [Online; accessed 27-September-2017].

[67] C. Liu, R. W. White, and S. Dumais, "Understanding web browsing behaviors through weibull analysis of dwell time," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 379–386.

[68] W. Weibull, "A statistical distribution function of wide applicability," *Journal of applied mechanics*, 1951.

[69] "Hadoop," http://hadoop.apache.org/.

[70] G. O. Karame, A. Francillon, V. Budilivschi, S. Čapkun, and V. Čapkun, "Microcomputations as micropayments in web-based services," *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, p. 8, 2014.

[71] D. Hinds, "Micropayments: A technology with a promising but uncertain future," *Communications of the ACM*, vol. 47, no. 5, p. 44, 2004.

[72] "Facebook statistics," http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/, [Online; accessed 27-September-2017].

[73] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1. IEEE, 2001, pp. I–511.

[74] "Bicubic interpolation," https://en.wikipedia.org/wiki/Bicubic-interpolation, [Online; accessed 27-September-2017].

[75] B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Foundations and trends in information retrieval*, vol. 2, no. 1-2, pp. 1–135, 2008.

[76] ——, "A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts," in *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2004, p. 271.

[77] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology-CRYPTO 2003*. Springer, 2003, pp. 617–630.

[78] "Forbes hacked," http://www.washingtonpost.com/blogs/the-switch/wp/2015/02/10/forbes-web-site-was-compromised-by-chinese-cyberespionage-group-researchers-say/, [Online; accessed 27-September-2017].

[79] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[80] W. R. Pearson, "Comparison of methods for searching protein sequence databases," *Protein Science*, vol. 4, no. 6, pp. 1145–1160, 1995.

[81] E. G. Shpaer, M. Robinson, D. Yee, J. D. Candlin, R. Mines, and T. Hunkapiller, "Sensitivity and selectivity in protein similarity searches: a comparison of smith–waterman in hardware to blast and fasta," *Genomics*, vol. 38, no. 2, pp. 179–191, 1996.

[82] "FASTA Program," http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml, [Online; accessed 27-September-2017].

[83] M. Larkin, G. Blackshields, N. Brown, R. Chenna, P. A. McGettigan, H. McWilliam, F. Valentin, I. M. Wallace, A. Wilm, R. Lopez *et al.*, "Clustal w and clustal x version 2.0," *Bioinformatics*, vol. 23, no. 21, pp. 2947–2948, 2007.

[84] "MUSCLE," http://www.drive5.com/muscle/, [Online; accessed 27-September-2017].

[85] Z. R. Yang, R. Thomson, P. McNeil, and R. M. Esnouf, "RONN: the bio-basis function neural network technique applied to the detection of natively disordered regions in proteins," *Bioinformatics*, vol. 21, no. 16, pp. 3369–3376, 2005.

[86] https://addons.mozilla.org/en-US/firefox/addon/noscript/, note = [Online; accessed 27-September-2017].

[87] https://gds.blog.gov.uk/2013/10/21/how-many-people-are-missing-out-on-javascript-enhancement/, note = [Online; accessed 27-September-2017].

[88] L. F. Sarmenta, "Bayanihan: Web-based volunteer computing using Java," in *Worldwide Computing and Its Applications*. Springer, 1998, pp. 444–461.

[89] T. Ong, T. Lim, B. Lee, and C. Yeo, "Unicorn: voluntary computing over internet," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 2, pp. 36–51, 2002.

[90] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[91] Folding@Home, "http://folding.stanford.edu/," [Online; accessed 27-September-2017].

[92] T. Krupa, P. Majewski, B. Kowalczyk, and W. Turek, "On-demand web search using browser-based volunteer computing," in *Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE, 2012, pp. 184–190.

[93] F. Konishi, S. Ohki, A. Konagaya, R. Umestu, and M. Ishii, "RABC: A Conceptual Design of Pervasive Infrastructure for Browser Computing based on Ajax technologies," in *Seventh IEEE International Symposium on Cluster Computing and the Grid*. IEEE, 2007, pp. 661–672.

[94] S. R. Wilkinson and J. S. Almeida, "Qmachine: commodity supercomputing in web browsers," *BMC bioinformatics*, vol. 15, no. 1, p. 176, 2014.

[95] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "Moon: MapReduce on opportunistic environments," in *Proceed-*

ings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, 2010, pp. 95–106.

[96] Y. Pan, J. White, and Y. Sun, "Assessing the threat of web worker distributed attacks," in *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2016, pp. 306–314.