Gray Computing: An Analysis of Computing with Background JavaScript Tasks

Yao Pan, Jules White
Vanderbilt University
USA
{yao.pan, jules.white}@vanderbilt.edu

Yu Sun
California State Polytechnic University, Pomona
USA
yusun@cpp.edu

Jeff Gray
University of Alabama
USA
gray@cs.ua.edu

***Abstract*—Websites routinely distribute small amounts of work to visitors' browsers in order to validate forms, render animations, and perform other computations. This paper examines the feasibility, cost effectiveness, and approaches for increasing the workloads offloaded to web visitors' browsers in order to turn them into a large-scale distributed data processing engine, which we term gray computing. Past research has looked primarily at either non-browser based volunteer computing or browser-based volunteer computing where the visitors keep their browsers open to a single web page for a long period of time. This paper provides a deep analysis of the architectural, cost effectiveness, user experience, performance, security, and other issues of gray computing distributed data processing engines with high heterogeneity, non-uniform page view times, and high computing pool volatility.**

I. Introduction

Every website visitor does some computational work on the website's behalf, such as validating input values of a web form before submission to the server, harvesting information related to the client's browser, animating page elements, or displaying complex data analytics. Website visitors implicitly opt-into performing computational work for the website owner without knowing exactly what 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*. The only hard line defining acceptable and unacceptable computing tasks is drawn by the security controls in the browser that protect the user from malicious logic.

The successful offloading of smaller tasks, such as form validation, motivates the question of whether or not there is potential to perform more computational work in this gray computing area of website visitors' browsers. Previously, browser-based JavaScript applications were single-threaded, preventing complex calculations from being performed without the user's knowledge since they would directly impact user interaction with the web page. **The emergence of new standards, such as Web Workers, which allow background JavaScript threads on web pages, offer the potential for much more significant background usage of visitor's computing resources without their knowledge**, which motivates the study of this new gray computing power.

For example, big data processing [1], where organizations process website visitor logs, user photos, social network connections, and other large datasets for meaningful information has become commonplace for large websites. Could organizations 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 privacy 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.

Scientists have effectively applied volunteer computing [2], which is a distributed computing paradigm in which computer owners donate their computing resources to research projects, to reduce costs and increase their computing power. Volunteer computing differs from gray computing in that volunteers typically directly opt-in by installing a client and have full knowledge of the computational tasks that they will be performing. Code obfuscation techniques, such as JavaScript minimization, typically make web visitors unaware of the computational tasks that they are performing on behalf of a website. Moreover, few website visitors look at the source code of webpages that they are visiting in order to determine the computational work their browsers are performing.

How much gray computing power is currently untapped? As we will show from empirical results presented in this paper, there is significant untapped gray computing power. For example, over 6 billion hours of video are watched each month on YouTube [3]. Assume that each client computer has an average processing ability of 50 GFLOPS, which is equivalent to an Intel i3 2100 [4]. Further assume that each client computer is only 25% utilized by gray big data computing tasks offloaded by YouTube, such as the client-side face detection task we present in Section IV-B. The combined processing power of the client computers would be 104 PFLOPS. Tianhe-2, the fastest super computer on record till June 2014, has a computing speed of 33.86 PFLOPS [5]. In this example, the combined processing power of the monthly web visitors to YouTube is ≈3X the largest supercomputer in the world. Even if the true gray computing power is only 1/100 of this estimate, the processing power would still make gray computing among the TOP50 fastest super computers in the world [5].

Open Question ⇒ 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. The prior research has not considered the cost effectiveness of building browser-based distributed data processing engines with gray computing or gray computing's impact on the user experience of a normal website. If gray computing can be done in a manner that is both cost-effective and does not impact user browsing experience, it warrants both significant concern and study. A number of key research challenges exist toward adopting gray computing that stem from the volatility, security, user experience considerations, and cost of performing distributed data processing with gray computing. This paper does not investigate the ethical and privacy concerns of gray computing in depth, but instead aims to determine if gray computing is an appealing enough target for both legitimate website operators and attackers to warrant further research into these issues.

This paper presents an architectural solution which addresses these research challenges and proves the feasibility of performing distributed data processing tasks with gray computing. There is a need for further study on the importance of the security, ethical, and other considerations of this type of computing resource utilization. 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.

Contributions: Although there are several past implementations [6] of browser-based distributed data processing engines, past work has not investigated the potential for more significant usage of website visitor computing resources with background JavaScript threads. This paper's main contributions, which differentiate it from previous work, are as follows:

- A complete architecture for gray computing is provided that improves cost-effectiveness by exploiting asymmetries in cloud pricing models.
- Experiments are presented that investigate the impact on user experience of popular websites, such as Facebook.
- Past research treated the computing resources of browser clients as free. The additional cost/load on the network and server when distributing data processing tasks to browser clients is considered in this work and shown to be a critical component of determining which tasks are cost-effective to use for gray computing.
- This paper considers more realistic browsing behavior versus past approaches that assumed the browser was open for long periods of time and did not consider situations where the user was only doing computation while a visitor was reading web pages on a site.
- A variety of real-world distributed data processing applications, ranging from computer vision to machine learning, are benchmarked and assessed for cost-effectiveness with gray computing.

The remainder of this paper is organized as follows: Section II presents the key research challenges investigated in this paper; Section III presents our proposed solutions to address these challenges and the corresponding empirical verifications; Section IV examines some practical applications of distributed data processing with gray computing and analyzes their suitability for this computing model; Section V describes related work; Section VI presents concluding remarks and future work.

II. Research Questions

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

Question 1: Does JavaScript provide sufficient performance for computationally intensive tasks?

Question 2: Does gray computing impact website performance in a user perceptible manner?

Question 3: What are the mechanisms for handling malicious clients and how will they impact gray computing performance?

Question 4: How cost-effective is gray computing versus commodity public clouds?

Question 5: How do you effectively allocate tasks to clients with unknown page view times?

The remainder of this paper 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.

III. Answering the Research Questions

To answer the key research questions, we built an implementation of a distributed data processing engine that can 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 [6], [7], [8], 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 [6], [7], [9], 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.** For example, 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 [9].** The load on task distribution servers can be substantially reduced 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 with 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 utilize 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.

Since 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). 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. Take 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. 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 proposed architecture, tasks are distributed to clients as follows:

1) The client requests an HTML webpage from the server.
2) The server injects an additional JavaScript file into the webpage 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.

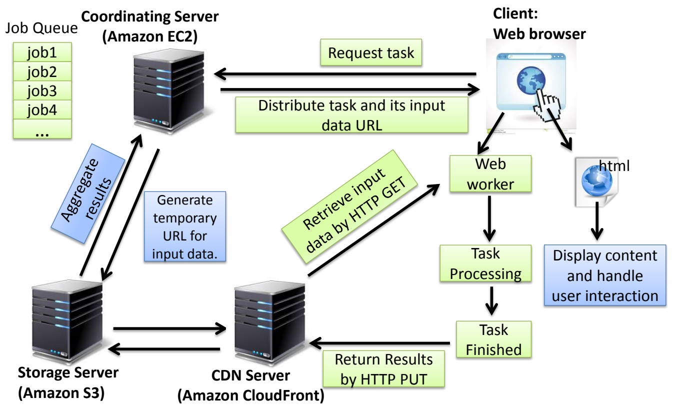

Fig. 1: Architecture of the proposed distributed data processing engine for gray computing.

A. Answering Question 1 ⇒ Benchmarking JavaScript Performance on Computationally Intensive Tasks

1) Current JavaScript performance: 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. Further, a variety of highly optimized tools have been developed to automatically port C/C++ programs to JavaScript. Emscripten [10] is an LLVM (Low Level Virtual Machine)-based project that compiles C/C++ code into highly-optimizable JavaScript in the asm.js format. Entire C/C++ code bases, such as the Unity 3D gaming engine and the ffmpeg video processing library have been ported to JavaScript with Emscripten. Emscripten works by first compiling C/C++ code through Clang [11] into an intermediary representation in LLVM. It is claimed the optimized JavaScript code can achieve near native performance to C++ code. We conducted some benchmark experiments to analyze the performance of the JavaScript produced by Emscripten. The results are quite startling: the ported JavaScript codes significantly outperformed the hand-written JavaScript code, and were within 2X performance of the original native C++ code.

TABLE 1: Execution time comparison of native C++, native JS, and ported JS (from C++). JavaScript is running with V8 JavaScript engine 3.27.

Benchmark	C++	native JS	ported JS
Nbody	6.6s	16.9s	6.2s
fasta	1.3s	11.4s	2.1s

Table 1 presents the results from the comparison between native C++ code, hand-written JavaScript code, and ported JavaScript code using Emscripten. The hand-written JavaScript code is written with the same algorithm as the C++ version. The ported JavaScript code is produced using Emscripten 1.2.0 with the "-O3" optimization flag. The C++ code was compiled using LLVM gcc 5.0 with the "-O3" optimization parameter. The reason for the performance boost is that Emscripten uses asm.js, which is a highly optimized low-level subset of JavaScript. The types in JavaScript are implicitly declared and thus take time for large applications to decide at runtime. Asm.js utilizes typed arrays for speed improvements. It also eliminates JavaScript's object-oriented constructs and thus eliminates many hard-to-optimize code constructs.

B. Answering Question 2 ⇒ 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, 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.

To test whether doing computationally intensive tasks in the background will affect the user experience, we conducted experiments with Greasemonkey [12] to inject data processing tasks into popular websites. Greasemonkey is a browser extension which 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. Background Web Worker threads can be run without affecting page load time.

The second concern is the page responsiveness. Many metrics exist for page responsiveness. In this project, we choose the search box item suggestion (or auto-completion), which is a pervasive and time-critical feature of many websites, for the user experience evaluation. Search box auto-completion is a time critical task since auto-completion results must be computed faster than the user types in order to be helpful. Our aim was to test if the generation of search suggestions would be slowed down by a computationally intensive background task. Two scripts were written in Greasemonkey. 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 proposed in Section IV, 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 2 illustrate, there was no discernible difference in the search suggestion load time with and without a Web Worker computational task running. 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 for distributed data processing.

TABLE 2: Average search box hint results generation time (T) and CPU utilization.

Website	With Web Worker		Without Web Worker	
	T	Avg. CPU util	T	Avg. CPU util
Twitter	0.58s	56%	0.59s	12%
Wikipedia	0.47s	58%	0.46s	15%
Gmail	0.75s	55%	0.73s	11%

C. Answering Question 3 ⇒ Mechanisms for Handling Malicious Clients and their Associated Overhead

Since 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. A simple but effective approach to handle malicious clients it to use a task duplication and majority voting scheme. More complex strategies, such as credibility-based approaches, exist 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. $f = C/N$ is the fraction of malicious clients in 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. Notice that the server randomly distributes tasks to clients, while ensuring no client receives both a task and its duplicate and that clients have no control over which task units they will get. 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. [13], 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 2, 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.

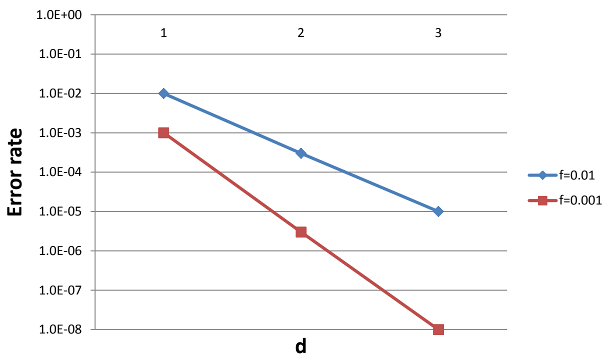

Fig. 2: Error rates for different parameter values of d and f.

D. Answering Question 4 ⇒ 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 decide whether it is cost-effective to deploy an application through gray computing and estimate the cost savings. In our cost model, we choose cloud computing services as our benchmark. We chose to compare against cloud computing because cloud computing has a quantifiable computing cost that we can compare against.

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

$$\begin{aligned} C_{browser} &= C_{transfer} + C_{request} + C_{distribute} \\ &= C_{toclient} + C_{toserver} + C_{request} + C_{distribute} \\ &\approx C_{toclient} + C_{toserver} + C_{request} \end{aligned} \qquad (2)$$

$C_{transfer}$ 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.

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

$$\begin{aligned} C_{toclient} &= k \times I \times P_{transferout} \\ &= k \times I \times (P_{OriginToCDN} + P_{CDNout}) \\ C_{toserver} &= O \times P_{transferin} \\ &= O \times (P_{toCDN} + P_{toOrigin}) \end{aligned} \tag{3}$$

$C_{toclient}$ is the cost to fetch input data for computing from an S3 storage server; I is the original size of input data to be transferred to clients for processing; $P_{OriginToCDN}$ is the unit price to transfer data from an 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.

The actual data transferred from server to client will be more than the original size of input data I. One reason is as described in Section III-C, 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 $k = d + \sum_{n=1}^{\infty} d(1-\mu)^n = d/\mu$. 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 successfully processed before a browser client leaves the site).

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

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

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}:

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

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 proposed 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.

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

Since 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 indicate an application is suitable for gray computing.

E. Answering Question 5 $\Rightarrow$ An Adaptive Scheduling Algorithm for Gray Computing

The clients in gray computing are website visitors' browsers, which are not static and unreliable. The clients 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 μ is, 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 3. 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. But the computing time of the assigned tasks is what we can change. 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. Therefore, there is a tradeoff between using smaller tasks and larger 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 [14], [15]: the probability a visitor leaves a web page decreases as time passes. This implies most visits are short. Our adaptive scheduling algorithm works by assigning tasks of smaller size to clients first and increasing the subsequent task sizes until a threshold

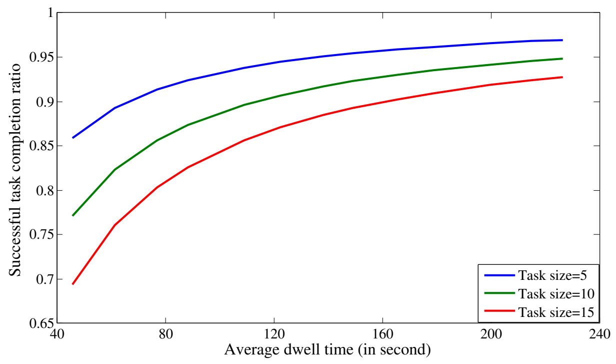

Fig. 3: The relationship between average dwell time and successful task completion ratio μ for different task sizes.

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 scheduler. The Hadoop scheduler 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 3. 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 3: 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

IV. Examples of Gray Computing Applications

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 applications are cost-effective in a gray computing model.

A. Experiment 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 run on Mozilla Firefox 31.0.

Cloud instance: We use 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 4 and 5. $P_{unit} = \$0.07/hour$. We choose a duplication factor $d = 2$ for most use cases, because it guarantees a relatively high accuracy as we have shown in Section III-C. We also choose 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: $U = T_{cloud} \times I \times P_{unit} - C_{toclient} - C_{toserver} - C_{request}$ for each task as shown in Table 6. A data processing task is cost-effective for distributed processing with browsers if the U is a positive number.

TABLE 4: A subset of Amazon cloud service type and price (As of February 2015)

Service	Subtype	Price	ECU
EC2	t2.micro	\$0.013 /Hr	varied
	m3.medium	\$0.07 /Hr	3
	m3.large	\$0.14 /Hr	6.5
	c3.large	\$0.105 /Hr	7

TABLE 5: A subset of Amazon cloud service type and price (As of February 2015)

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

B. Use Cases

Scenario 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 to them. 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

TABLE 6: Comparison of cost saving per GB for different tasks. The higher this value, the more computationally intensive the task, and the more suitable the task for distributed data processing with browsers.

Task	Cost savings \$ per GB
Rainbow table	∞
Face detection	0.09
Sentiment analysis	0.07
Image scaling	0.008
Word count	-0.16

clients to compute without input data. However, there are some circumstances that 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 client. In these cases, no extra data transfer costs are incurred. Facebook has 350 million (as of 2013) photos uploaded every day and face detection in photos is a relatively computationally intensive computer vision task. Our hypothesis was that significant cost savings could be obtained through gray computing.

Experiment 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 the JavaScript facial detection algorithm, we used an opensource implementation of the Viola-Jones algorithm [16]. 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 7. 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. Facebook has 350 million photos uploaded by users each day. Suppose the average resolution of the photos being uploaded is 2 million pixels, which is less than the average resolution of most smartphone cameras, such as the 8 million pixel (8 megapixel) iPhone 6 camera. It takes $1.7s \times 3.5 \times 10^8/3600h = 165,278h$ of computing time for an EC2 m3.medium instance to process these photos. With our utility function $U = T_{cloud} \times I \times P_{unit} - 1/\mu \times d \times I \times P_{transferout} - O \times P_{transferin} - C_{request}$, where $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring data to the clients. 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 only needs to issue 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 choose a duplication factor $d = 1$ since this application is not very sensitive to security. Thus, $C_{request} = 350 \times 10^8/5 \times 0.0075/10^4 = 5250$, and $U = 165278 \times 0.07 - 5250 = \$6319$ 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 \$2.3 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.

Scenario 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 Scenario 1, 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 becomes huge, the saved computation cost could be substantial.

Experiment 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 in the real-world and did not want to focus on the differences in efficiency of the algorithms. We chose bicubic interpolation 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 8.

Analysis of Results. To get a quantitive 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 $0.55s \times 3.5 \times 10^8/3600h = 53,472h$ for one EC2 m3.medium instance to scale and compress. With our utility function $U = T_{cloud} \times I \times P_{unit} - k \times I \times P_{transferout} - O \times P_{transferin} - C_{request}$. Again $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring data to the clients. We assume the client processes 10 photos at a time (which takes around 10s on a browser with average hardware). We choose a duplication factor $d = 1$ since this application is not very sensitive to security. $O = 3.5 \times 10^8 \times 80k/10^6 = 2.8 \times 10^4 GB$. $C_{request} = 350 \times 10^8/10 \times 0.0075/10^4 = 2625$. Therefore, $U = 53472 \times 0.07 - 2.8 \times 10^4 \times 0.02 - 2625 = \$558$ is saved each day by distributing this task to browsers, which aggregates to \$203,670 dollars 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.

TABLE 7: 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 8: 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

Scenario 3: Sentiment Analysis

Overview. Sentiment analysis [17] 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.

Experiment Setup. There are many machine learning algorithms proposed for sentiment analysis in the literature [17]. 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 [18] 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 record 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 input data, the cost savings is $1000/0.2 \times 0.7/3600 \times 0.07 = \$0.07$. To get a quantitive 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 $350 \times 10^6 \times 1/10^6 = 350GB$ comments in total. If Facebook wants to analyze the attitude of every comment of the photos, the cost is $350 \times 0.07 \times 365 = \$8943$ a year.

Scenario 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.

Experiment 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).

TABLE 9: Computing time comparison for wordcount.

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

Empirical Results. The experiment results in terms of computing speed are shown in Table 9. For 1GB of input, $C_{request}$ is very small, the cost savings is $C_{cloud} - C_{toclient} = 1/0.38 \times 6/60 \times 0.109 \times 2 - 0.085 \times 2.5 = \$ - 0.16$.

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 Scenarios 1 and 2.

Scenario 5: Rainbow Table Generation

Overview. The final 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 [19] 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, $C_{toclient} = 0$.

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

Empirical Results. The results of the experiment are shown in Table 10. 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.

TABLE 10: Computing time comparison for generating a rainbow table.

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

Analysis of Results. Suppose an attacker compromises websites with a cumulative traffic of 1 million visitors per day and an average browsing time of 15 minutes. The attacker would be able to wield the equivalent of $10^6 \times 0.25hour \times 0.8/2 \times 365day = 3.65 \times 10^7$ computing hours a year in a browser. To accomplish the same task with an Amazon EC2 m3.medium instance for a year, the yearly cost would be $3.65 \times 10^7/4 \times 3 \times 0.07\$/hour = \$1,916,250$. $C_{request} = (2.5+2) \times 10^6 \times 900/10 \times 365 \times 0.0075/10^4 = \$110,869$. With our utility function $U = C_{cloud} - O \times P_{transferin} - C_{request}$, the yearly cost saving U is around \$1.8Million. The near-native computing speed of JavaScript with browser and the low data transferred per unit of computational work makes this an effective processing task to distribute.

V. Related Work

The term "volunteer computing" was first used by Luis F. G. Sarmentan [20]. He developed a Java applet-based volunteer computing system called Bayanihan [20] in 1998. SETI@Home [21] 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 [22], which simulates protein folding for disease research, and BOINC [23], 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 websites' business operations.

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. [24] proposed a browser-based volunteer computing architecture for web crawling. Finally, Konishi et al. [25] 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) 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 and e-commerce.

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 [26]. There are some early prototypes implementing MapReduce with JavaScript [6]. Lin et al. [26] 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 high rate of unavailability, as we also demonstrated with our derivation of μ in Section III-E. 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.

VI. 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 paper, 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 it. 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 paper, we derived a cost model that can be used to assess the suitability of different data processing tasks for distributed data processing with gray computing. This cost model can aid future discussions of ways of legitimately incentivizing users to opt-into gray computing. Further, we pinpointed 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 aid in guiding the design of gray computing systems and user incentive programs. We also presented 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.

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. Sarmenta, "Volunteer computing," Ph.D. dissertation, Citeseer, 2001.

[3] "Youtube statistics," https://www.youtube.com/yt/press/statistics.html.

[4] "Intel processors," http://www.intel.com/support/processors/sb/CS-017346.htm.

[5] "Top500 supercomputer," http://www.top500.org/.

[6] 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.

[7] 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, IEEE*, vol. 17, no. 5, pp. 54–61, 2013.

[8] 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.

[9] 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.

[10] "Emscripten," http://kripken.github.io/emscripten-site/index.html.

[11] "Clang," http://clang.llvm.org/.

[12] Greasemonkey, "http://www.greasespot.net/."

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

[14] 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.

[15] W. Weibull, "Wide applicability," *Journal of applied mechanics*, 1951.

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

[17] 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.

[18] B. Pang and L. Lee, "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.

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

[20] L. F. Sarmenta, "Bayanihan: Web-based volunteer computing using java," in *Worldwide Computing and Its Applications(WWCA'98)*. Springer, 1998, pp. 444–461.

[21] 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.

[22] Folding@Home, "http://folding.stanford.edu/."

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

[24] 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), 2012*. IEEE, 2012, pp. 184–190.

[25] 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, 2007. CCGRID 2007*. IEEE, 2007, pp. 661–672.

[26] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "Moon: Mapreduce on opportunistic environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 95–106.