COMP520-08 Report

Parasitic JavaScript

Nick Jenkin

2008

Supervisor: Bernhard Pfahringer

Abstract

Millions of people use the Internet everyday to browse websites, but what if the website was doing a little more than just serving content? What if upon visiting a website, the web browser would start participating in a distributed computing project — without the user knowing?

This project explores a parasitic computing solution to turn web browsers into participants of a distributed computing project. Web browsers today support a scripting language known as JavaScript, which is primarily used to give an interactive user experience. This project uses JavaScript to perform distributed computing tasks.

A system was developed to distribute several sample tasks using JavaScript. To ensure the user is unaware that the computation is occurring, several techniques were developed and evaluated to ensure that the user experience is not harmed. A user study was performed to determine the effectiveness of the proposed methods.

This project shows that Parasitic JavaScript is a viable option for SETI@home style distributed computing. Parasitic JavaScript's main advantages are that is does not require installation, and it runs unobtrusively in the background. The main limiting factor is the efficiency, or rather lack thereof, current web browsers provide with regard to JavaScript performance.

Contents

1 Introduction **6**

2 Background **7**
2.1 Parasitic Computing . . . 7
2.1.1 Solving SAT Problems with TCP and HTTP . . . 7
2.1.2 Applications of Parasitic Computing . . . 9
Skype . . . 9
BitTorrent . . . 9
Gnutella . . . 9
2.1.3 Ethics . . . 10
2.2 Distributed Computing . . . 10
2.2.1 Distributing a Task . . . 11
2.3 Volunteer Computing . . . 11
2.4 JavaScript . . . 12
2.5 Parasitic JavaScript . . . 13

3 Distribution Methodology **14**

4 Server Design **15**
4.1 Architecture . . . 15
4.2 Result Integrity . . . 15
4.2.1 Work Unit Duplication . . . 16
4.2.2 Partial Work Units . . . 16
4.2.3 User Reliability . . . 16
4.2.4 Improvements . . . 16
4.3 Scalability and Cost . . . 17

5 JavaScript Client Design **18**
5.1 Architecture . . . 18
5.2 Task & Work Unit Considerations . . . 19
5.3 Timer Based Programming . . . 19
5.4 Dynamic Performance Adjustment . . . 20
5.5 Embedding the JavaScript Client . . . 22

6 Tasks **24**
6.1 Creating a Task . . . 24
6.2 Tasks Implemented . . . 25
6.2.1 Ray Tracing . . . 25
6.2.2 Perceptron . . . 26
6.2.3 N-Queens . . . 26

7 Evaluation of Dynamic Performance Adjustment **29**
7.1 Results . . . 30

8 Performance **34**
8.1 Browser Performance . . . 34
8.1.1 Results . . . 34
8.2 Distribution Overhead . . . 35
8.2.1 Results . . . 36

8.3 Improving Performance 37

9 User Study **38**
9.1 Design 38
9.2 Questionnaire 39
9.3 Procedure 40
9.4 Results 40
9.4.1 User Study 40
9.4.2 Questionnaire 42

10 Conclusions and Future Work **44**

A User Study **48**
A.1 Ethical Consent 48
A.2 Questionnaire 49

List of Figures

1 Diagram of a TCP packet being accepted and rejected by the HTTP server 8
2 Client-Server distribution used in Parasitic JavaScript 14
3 Task Life Cycle 18
4 Slow JavaScript warning in Safari 20
5 Example of a loop converted to a timer based loop. 20
6 Web browser execution space 23
7 Using an IFRAME to bypass JavaScript cross-domain security . 23
8 HTML to embed Parasitic JavaScript 23
9 Ray Tracing Algorithm 25
10 Stages of Ray Tracing 26
11 8-Queens Solution 27
12 Computer A. Calculated delay of no load and typical load conditions. 30
13 Computer B. Calculated delay of no load and typical load conditions. 31
14 Computer C. Calculated delay of no load and typical load conditions. 31
15 Computer D. Calculated delay of no load and typical load conditions. 32
16 Computer C. Calculated delay during variable load conditions of one minute intervals. 33
17 Computer D. Calculated delay during variable load conditions of one minute intervals. 33
18 Average work units completed per minute for each web browser . 35
19 Completion times with varying numbers of work units for ray tracing an image containing 1024 pixels 36
20 Estimated completion times with varying numbers of work units for ray tracing an image containing 1024 pixels 37
21 Example of the user study window 39

List of Tables

1 Computers used for evaluation 29
2 Web browsers used for performance testing. 34
3 Aggregated results of participants in the user study 41
4 Breakdown of results by pages containing Parasitic JavaScript or not 41
5 Accuracy of the participants at predicting which pages contained Parasitic JavaScript 41
6 Summary of results collected in the questionnaire 43

List of Algorithms

1 Dynamic Performance Adjustment 22
2 Counting Task 24
3 Perceptron 27
4 Genetic Algorithm 28

1 Introduction

The invention of the world wide web in 1990 provided a new avenue for people to share ideas and communicate, but the web browsers of the time provided little interactive functionality. In 1995 Netscape released a web browser that supported a new programming language that could be embedded into Hyper Text Markup Language (HTML) to provide basic interactive functionality, originally called Mocha, but now known as JavaScript. Since then JavaScript has been gradually increasing in popularity and recently there has been significant growth in its use in the field of interactive web applications such as Google Docs.

Parasitic computing is the concept of using computers to perform some form of computation without the owners being aware. This is not to be confused with hacking or a virus, but tricking a resource into performing calculations it was not designed to perform. The first system proposed solved boolean satisfiability problems using TCP packets and an HTTP server. This caused outrage around the world as server administrators feared that their servers would be abused to become calculators. While the TCP method never became popular it did raise ethical and legal questions about parasitic computing and this has stifled research in the field.

Distributed computing is often used to compute large scientific problems and some systems have even been created to allow users to participate in large distributed computing projects from home. Distribution itself is a research area in its own right, with many methods proposed for large computing clusters or public resource computing.

This report combines the ideas of parasitic computing with JavaScript to make web browsers perform behind-the-scenes distributed computation. This effectively reverses what JavaScript was originally designed to do. A system is proposed that allows for the distribution of tasks in a web browser. This system is split into a server and client, where the server controls the distribution of tasks and the client performs the computation of a task.

One of the primary goals of this project was to ensure that the user experience is not impaired. In order to meet this goal an algorithm is proposed to regulate the execution speed of Parasitic JavaScript. This algorithm was then evaluated and a user study was performed to determine if the participants were able to identify websites containing Parasitic JavaScript.

There is a large range of web browsers available today and this report also investigates how effective some popular web browsers are at running Parasitic JavaScript. In addition to this the distribution performance is evaluated to determine the amount of overhead distribution has, when compared to non-distributed methods.

2 Background

In this section the concepts used to create Parasitic JavaScript are explained. Parasitic computing and applications in use today that exhibit parasitic behaviour are discussed. Several distributed computing techniques are covered and the emergence of public resource computing are also discussed. Finally JavaScript and Parasitic JavaScript are discussed in further detail.

2.1 Parasitic Computing

Parasitic computing is the idea of creating a program that uses legitimate methods to perform some computing task [1] without the user being aware. The concept was first conceived by Barabási et al. in 2001 but has received a minimal amount of research since. It has several ethical barriers, but this has not stopped it from being used in several popular applications available today.

Parasitic computing could be considered to be related to projects like SETI-@home [2] or distributed.net [3] that allow users to donate their computing resources. The main difference between parasitic computing and volunteer computing is that with parasitic computing you do not choose to donate computing resources. While this may sound malicious, it should not be confused with a virus or malware. Where a virus may corrupt files or steal private information, parasitic computing only uses computing resources to perform some task, generally without the user being aware.

By definition the means that a parasitic computing program uses to perform a task should be legitimate. The term legitimate in this situation means that the program does not need to exploit a system in order to gain access to computing resources.

2.1.1 Solving SAT Problems with TCP and HTTP

Barabási et al. proposed a parasitic computing method whereby modified TCP packets and HTTP servers were used to solve boolean satisfiability problems (SAT) problems [1]. SAT problems involve finding a combination of values for variables within a boolean expression that results in that expression evaluating to true. For example, given the expression $(x_1 \oplus x_2) \wedge (x_3 \wedge x_4)$ a possible combination of values for the expression to evaluate to true are $x = \{T, F, T, T\}$. The algorithm Barabási et al. proposed is capable of solving 2-SAT and 3-SAT problems. 2-SAT problems have two literals within each clause and 3-SAT problems have three literals.

The SAT solver uses carefully crafted TCP packets to perform the computation. TCP packets contain checksums that are used to verify the contents of a packet to help ensure that no data corruption has occurred during transmission [4]. The algorithm modifies this checksum such that, if it is valid, the solution is correct [5]. Figure 1 shows the result of a valid and invalid solution. The valid solution continues onto the HTTP server, which will return back a response. The invalid solution is dropped at the TCP layer of the remote host.

The TCP packet contains two 16-bit words for data. The checksum is calculated by summing these two values together and taking the complement. So if the resulting sum is `1101 0101 1111 0001` it would have a complement of `0010 1010 0000 1110`. The complement is then sent with the TCP packet as

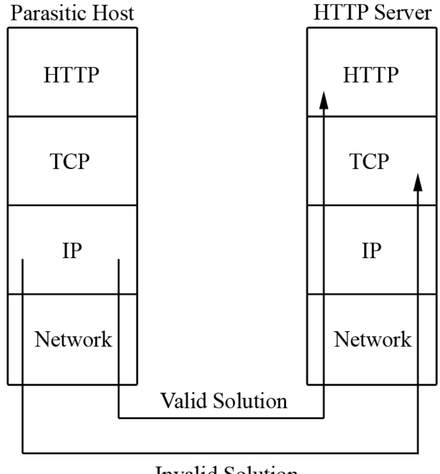


Figure 1: Diagram of a TCP packet being accepted and rejected by the HTTP server

the checksum and the remote host then sums the two 16-bit words resulting in `1101 0101 1111 0001`, which is then added to the checksum within the packet that results in a value of `1111 1111 1111 1111`. If the result does not equal `111..1` then the packet is considered corrupt and discarded. The 2-SAT solver is effectively using the TCP protocol to evaluate $a + b = c$.

To be able to use TCP packets to solve 2-SAT problems, the data and checksum values need to be manipulated. The two data values are taken from the set of possible combinations of values for a given boolean expression. Consider the expression $(x_1 \oplus x_2) \wedge (x_3 \wedge x_4)$. This expression has four variables and consequently 2^4 combinations of values. One combination is $x = \{T, F, T, F\}$, in order to determine the TCP data for this combination it must be first converted to binary. The conversion to binary involves converting each value in x into two bit zero padded values. This results in a binary value of `01 00 01 00`. Now the values must be split into two sets, $D_1 = \{x_1, x_3\}$ and $D_2 = \{x_2, x_4\}$ or $D_1 = \{01, 01\}$ and $D_2 = \{00, 00\}$. The two values of D represent the two separate data values sent within the packet. The checksum is calculated by taking the expression and applying the operators on the data values. So $(x_1 \oplus x_2) = (01 \oplus 00) = 01$ and $(x_3 \wedge x_4) = (01 \wedge 00) = 00$. The checksum is `10 11`, which is the compliment of `01 00`.

The receiver then takes the two D values and adds them together resulting in `01 01`, which is then added to the checksum, `01 00`. This results in a value of `10 01` indicating that $x = \{T, F, T, F\}$ does not make the expression true. The 3-SAT solution is generated in much the same way, with the exception that the TCP packet size is increased to three 16-bit words.

In order to be able to determine if a given expression is satisfiable, one TCP packet needs to be sent for every combination of values in the worst case scenario. Consequently this could generate a large amount of network traffic so this method is not particularly efficient. This is what ultimately makes this technique not very practical. Barabási et al. concluded that the communication-

computation ratio would need to increase before parasitic computing became viable.

Kohring proposed that instead of using TCP packets, ICMP packets would be more efficient [6]. ICMP packets are a more efficient solution to parasitic computing compared to TCP due to their connectionless nature [4]. Rather than solving SAT problems, Kohring implemented a distributed version of the game Life [7].

2.1.2 Applications of Parasitic Computing

While it is difficult to find explicit examples of real-world applications of parasitic computing, many programs available today do exhibit some parasitic behaviour. The majority of these applications can be found in the area of P2P [8]. P2P is used in a variety of applications from Voice-over Internet (VoIP) to file sharing. What makes P2P applications parasitic is related to the fact that many users are unaware of the implications of using such services. P2P systems rely heavily on users to contribute bandwidth and computing resources for the network to operate efficiently.

Skype Skype [9] is an example of VoIP software that shows some parasitic behaviour. Skype has the concept of users becoming supernodes [10]. Supernodes are users who have desirable properties such as a fast Internet connection that is not behind a NAT, firewall or proxy. Studies have shown that hosts meeting the supernode criteria can be promoted within minutes of joining the Skype network. Supernodes are responsible for relaying queries between standard nodes, general protocol messages, instant messages between users and voice traffic [11]. It is not impossible to prevent Skype clients from becoming a supernode, but it does involve editing registry settings in Windows to disable it. Skype is often installed at companies and universities that commonly have high-speed Internet connections. If a large number of users were to become supernodes within one of these networks the costs could become significant.

BitTorrent BitTorrent [12] has become a popular file sharing protocol that allows for highly scalable P2P file distribution. BitTorrent users who contribute to the sharing of a file (e.g. upload data) are favoured more within the network [13]. While this is more of a symbiotic relationship between peers, it can still be considered parasitic as many users may be unaware of the fact that they are uploading data to other peers.

Gnutella Gnutella is another file sharing network. The Gnutella protocol is parasitic for the same reasons Skype is. Users on the network can become a *servent* (server + client). Servents behave very similar to supernodes on Skype: they allow other users to send query information (such as searches for files) through the servent, which then relays the query to other users on the network [14]. In order to become a servent, promotion occurs automatically once the user has connected to a certain number of nodes within the network.

2.1.3 Ethics

Parasitic computing touches some considerable ethical boundaries that should be considered. The usage of computing resources without the owners knowledge is of the most concern. This is a violation of the ACM code of ethics [15] which states members should only access computing and communication resources when authorised to do so. But what is the definition of authorised? The example proposed by Barabási et al. used publicly available resources and a modified TCP packet. Is authorisation granted given the resource is public? Crowell et al. considered the analogy of parking in a shop's car park and not going into the shop [16].

Crowell et al. [16] studied the ethics of parasitic computing and while they did not come to a decisive conclusion, they brought up several interesting points. The authors of the original parasitic computing paper were asked for their views on the ethics of parasitic computing. The group was split, with half thinking it was ethical and the other half thinking it was not. Another interesting point was the concept that CPU cycles are highly perishable, if you do not use them, they are gone.

While this project does not aim to decisively answer the question of whether parasitic computing is ethical or not, a user study was performed that included questions relating to ethics (see Section 9). Our belief is that consent should always be given in real-world applications.

2.2 Distributed Computing

Distributed computing involves using many computers to contribute to an overall goal. There are many different structures for distributed computing from clusters, where many computers act as one, to client-server or P2P architectures where servers within the distributed computing network contact other servers for instructions on what to do. The design of a distributed computer is often dependent on the requirements of a task.

The cluster computing model involves group of computers running as one [17]. This model requires tight control over all the computers within the cluster, but also low latency, high bandwidth networks are required in order to be able to carry messaging data between nodes.

Grid computing follows a client-server architecture but is restricted to a controlled environment and can be spread over a large area [18]. Consider a university which owns a large number of computers that are predominantly used during working hours. During the night it would be possible for all the computers within the network to become part of the grid computer and contribute to a task. Grid computing models are generally less reliable than clusters, that is nodes within the network could leave, but they are generally considered reliable in terms of the information they compute.

The client-server or P2P architecture involves using remote computers to request work. The client-server model is slightly less redundant than the P2P model due to the dependence on a small number of servers. The P2P model involves using clients to distribute work, much like modern P2P file sharing applications behave. This type of architecture is excellent for situations where large amounts of bandwidth is not available. Clients can request a work unit, process that work unit and send the result back.

A work unit describes the parameters to a given problem. It does not describe how to solve the problem. Work units are commonly used in client-server or P2P architectures to distribute instructions. Consider SETI@home [2] where work units are used to distribute audio data taken from satellites. SETI@home work units are about 350KB in size and can take several hours to process. The relatively small size work unit allows people on a range of different Internet connections to contribute. It also does not impose significant resource constraints on the servers that allocate work units.

2.2.1 Distributing a Task

Designing a task that can be distributed is often a complex thing to do. Consequently there has been significant amounts of research into developing methods that allow for simple design and deployment of massively distributed tasks. Several of these methods involve designing programming languages that are designed with distribution in mind. An example of this is Sawzall [19], which is a system and programming language designed from the ground up to analyse very large log files. Generalised programming languages for distribution are also available, such as Erlang [20]. Other techniques involve using libraries like OpenMP [21] that provide APIs for scalable shared memory multiprocessor applications. Ultimately the distribution technique for a task is often linked to the task itself.

2.3 Volunteer Computing

Volunteer computing is the idea that users wishing to participate in a distributed computing project install a program on their computer to donate computing resources to the project. The most famous example of this is SETI@home, which allows users to install an application that processes audio signals from radio telescopes to aid in the search for extraterrestrial life. SETI@home is built upon the BOINC [22] platform which allows for people to create tasks that users can contribute to. BOINC currently has over 300,000 active volunteers with more than 550,000 computers.

Volunteer computing or public resource computing introduces some additional complications when dealing with distribution. With the standard distribution environment where the servers are all owned and controlled by the people running the computation, there is a guarantee of a certain level of accuracy and reliability. In public resource computing this reliability guarantee does not exist, and as such requires systems in place to compensate for uncompleted work, malicious users and reduced network bandwidth.

Given that users are required to download and install an application to participate in volunteer computing, it also allows them to modify the application to produce undesirable results. Consequently checks need to be put in place to ensure that results are valid [23].

Network bandwidth can also become an issue in volunteer computing due to work units being transferred over the Internet rather than a local network. Given the wide range of Internet connection speeds available, work unit size needs to be taken into consideration as to ensure that only a reasonable amount of bandwidth is being consumed (by both the volunteer and the work unit allocation servers) [2].

One of the primary aspects seen in many volunteer computing projects is a credit system. Users are given credit for completed work units and this accounts towards an overall ranking. The BOINC project found that users were highly motivated by credit [22]. Users are also able to join teams, which then allows teams to compete against each other. This encourages users to get their friends to participate.

2.4 JavaScript

JavaScript [24] is a client-side scripting language supported by most modern web browsers. A common misconception with JavaScript is that it is related to Java, this is not the case and was merely the result of a marketing decision by Netscape [25]. The major benefit of using JavaScript is that it is supported by web browsers and does not require separate installation, compared to Adobe Flash [26] or Java Applets [27] that provide similar interactive functionality, but require installation of third-party plugins. The majority of functions that are generally performed with JavaScript include the manipulation of page layout, interactive interfaces and additional functions such as the gathering of traffic statistics.

JavaScript was originally developed at Netscape in 1995 and has seen several years of instability in terms of standardisation. In 1999 the European Computer Manufacturers Association (ECMA) released the ECMA-262 standard for JavaScript [24]. Although, ECMA called it ECMAScript. Since the standard was released all the major web browsers have implemented it, which has stabilised the language across the different web browsers.

JavaScript is an object-oriented language that provides a standard range of libraries that are generally found in most programming languages. The key difference between JavaScript and many other languages is that it is executed in a sandbox; there is no access to file I/O, threading or system information. One crucial I/O method provided is `XMLHTTPRequest`, more commonly known as AJAX [25]. `XMLHTTPRequest` allows you to asynchronously request data from a remote HTTP server. This is important because it allows the transfer of data to and from the remote HTTP server without alerting the user or interfering with the users actions.

JavaScript is by no means the perfect language. In many web browsers, JavaScript that takes a long time to execute can cause the web browser to become unresponsive. This results in the user being unable to use the web browser until execution stops. There are solutions available to this problem, which will be described in Section 5.3. An additional problem includes the ability for users to easily see and edit the JavaScript being executed. This can inadvertently result in either curious or malicious users modifying the currently executing JavaScript.

JavaScript has been increasing in popularity over recent years with the introduction of AJAX to create interactive services like Gmail and Google Docs. It is these services that are placing demand on the manufacturers of web browsers to improve the performance of JavaScript. Currently the manufactures of Mozilla Firefox, Internet Explorer and Safari are all developing new and improved JavaScript engines promising faster performance in the near future.

2.5 Parasitic JavaScript

Parasitic JavaScript is a combination of parasitic computing and JavaScript. The idea is to use web browsers to solve massively distributed problems without harming the user experience. Web browsers provide a great platform for parasitic computing as they are almost guaranteed to be installed on any modern consumer computer. Consider this in comparison to SETI@home where users have to physically download the SETI@home software to be able to donate computing time. If the volunteer computing route was taken with Parasitic JavaScript, users would only need to open up their browser and navigate to a website.

The use of web browsers and JavaScript to perform complex computation has been previously proposed by Merelo et al. [28]. The task implemented was a distributed evolutionary computing task where the server would generate candidate individuals and the web browser would evaluate the individuals sending the results back to the server. Their approach did not attempt to ensure that the user experience was not harmed, which resulted in long-running JavaScript errors appearing in the browser. The design was more specifically based around the volunteer computing approach, where users would navigate to a website in order to participate in the computation.

The system used a work unit methodology of distribution where population candidates were compiled into work units. The web browser would request a work unit using JavaScript's `XMLHTTPRequest`, it would then evaluate each candidate and return the result via an `XMLHTTPRequest` call.

There were several issues identified in their approach. Due to the server doing a significant amount of the computation in terms of generating new population candidates, their solution did not scale well. In addition to this, performance issues were identified with regard to the number of web browsers required to be running in order to equal the performance of a single machine. But most importantly, users would be unable to use their web browser while this is running due to the nature of long-running JavaScript rendering the browser unresponsive.

JavaScript has also been proposed in the past as a method of perpetrating denial of service attacks [29]. JavaScript allows for the creation of HTML elements within a website, including images. The method involved using JavaScript to constantly create hidden images that linked to a remote web server. This resulted in visitors unwittingly participating in a denial of service attack on the remote web server. The attack could be made untraceable by redirecting traffic through an anonymous proxy.

3 Distribution Methodology

Parasitic JavaScript falls under the realm of public resource computing, so many parallels can be found in similar public resource computing projects like BOINC. The distribution methodology behind Parasitic JavaScript involves a client-server model with work units being used to distribute tasks. This model is desirable for use over the Internet as the communication-computation ratio can be maximised. In comparison, a cluster model would require constant communication between the server and client.

Figure 2 shows the one-to-many server-client relationship used, with the clients (Section 5) connecting to the server (Section 4) to request work units, but also to send results.

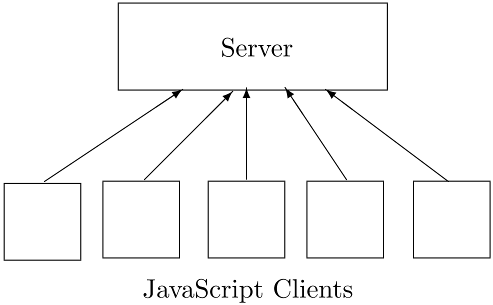


Figure 2: Client-Server distribution used in Parasitic JavaScript

In order to distribute tasks, work units are used to split a large task into many pieces. These pieces are then given to clients to compute and the results are then sent back to the server. A work unit can contain the data required to compute a task, such as settings for an algorithm or input data for processing. The same approach is used for BOINC based projects, with the primary difference being the size of the work units. BOINC projects can have large work unit sizes of several megabytes that can take several hours to compute. This is not practical in Parasitic JavaScript as website view times are generally less than a minute.

The result of this setup is that the client only contacts the server when it needs to, which reduces the amount of network traffic required and reduces the server load. It is possible to create a client that is in constant communication with the server, this would allow for a larger amount of data transfer between the client and server, which could lead to the development of tasks that process real time data.

4 Server Design

The server is responsible for managing work unit distribution and collection, and also maintaining work unit result integrity. The design has been kept fairly simple, utilising the distribution approach described in Section 3.

4.1 Architecture

The server consists of an HTTP server (Apache[1]) and a database server (MySQL [2]). The language used to develop the functionality of the server was PHP[3]. These technologies are all widely used today and infrastructure is readily available for them, so by using these technologies it potentially makes it easier for future scalability.

The database is responsible for storing available tasks, work units and results for each task. The server as whole is responsible for managing the allocation and retrieval of work units, serving the JavaScript required to complete a specific task and ensuring that the results being received are valid.

Tasks are not allocated dynamically, a web browser must first request to work on a specific task. This is done by going to the task's unique URL, for example `http://example.org/task/1/` would indicate that the web browser wishes to compute the task that has the id 1. Upon requesting this page, the JavaScript required to compute this task is sent to the browser. Once the web browser has loaded the JavaScript it requests a work unit.

The work units for a task are requested by initiating a state request, which again is performed by the web browser requesting a specific URL. The server allocates a work unit by randomly selecting one from a pool of available work units. The work unit is then allocated a password, which must be returned with the result in order for it to be accepted. The reason for randomising the selection and also the addition of a password is to prevent against malicious users guessing the id for a work unit and submitting inaccurate results. Work units also have an allocation date assigned, so in the event a work unit is not completed within a reasonable amount of time, it is reinstated into the work unit pool.

In order to allow for specific variants in the handling of each task's work units, each task has a controller which can be customised to perform specific actions. Customised actions can involve specific result integrity checks or creating new work units which succeed the work unit just completed. Generally for many tasks the default controller is sufficient.

4.2 Result Integrity

The use of public computing resources presents some additional complications when dealing with result integrity. JavaScript is a non-compiled language that can be viewed and consequently modified without significant effort. This allows malicious users to modify the computation that occurs, which can potentially lead to sabotaged results. The consequences of having results sabotaged can be

[1] Apache Web Server - http://httpd.apache.org

[2] MySQL Database Server - http://www.mysql.com

[3] PHP, programming language - http://www.php.net

severe and systems need to be in place to identify sabotaged results, but also to identify the users performing the sabotage.

Depending on the task being performed, there are many ways in which validation of results can be performed. In many cases it is ideal to specifically customise a validation solution based on the task. This might involve creating an estimation technique that can quickly calculate a range the results are expected to fall within. But there are also cases where the task might be self-healing where the result of a work unit is further improved in future work units. A task could also have easily verifiable solutions, such as n-queens (Section 6.2.3) where the solutions can be verified server-side without requiring a large amount of computation.

4.2.1 Work Unit Duplication

Work unit duplication is a simplistic approach to validation but it provides a high fidelity depending on the amount of duplication that occurs. The obvious down side to this approach is that there can be a significant amount of computation wastage.

4.2.2 Partial Work Units

Partial work units allow for a minimum amount of validation to occur while still ensuring a level of accuracy, although this method is not suited to all tasks. It is suitable for tasks such as ray tracing (Section 6.2.1). When work units contain many individual segments that do not depend on each other, it is possible to create work units which contain partial segments of other work units. This results in work units effectively being spliced into other work units. In situations where this method is applicable, this method is excellent as it minimises the duplication of work.

4.2.3 User Reliability

In situations where the user can be identified, it is possible to assign each user a level of reliability [30]. Reliability can be determined by the number and accuracy of results returned by a given user. As the reliability of a user increases, the chances of validation occurring decreases, but it should never reach zero. This method provides a level of work unit quality without significant wastage of work units.

4.2.4 Improvements

The techniques described above can be improved further by using blacklists, where users that have been identified as saboteurs are prevented from submitting results. In addition to this, work units can be distributed in such a way that a single user cannot process the same duplicated work unit or partial work unit. This can be used to ensure a single user does not self-validate work units. In order to help prevent colluding parties from submitting results, geo-location can be used to ensure validation work units are not allocated to users within the same geographic region.

4.3 Scalability and Cost

To have thousands of users running Parasitic JavaScript is going to require a significant amount of infrastructure available to assign and collect work units. The issue with this is determining at what point the cost of the infrastructure required to run Parasitic JavaScript exceeds the benefit of using Parasitic JavaScript. While this question remains open, there are potential solutions available to reduce cost.

Services like Amazon EC2[4] and Google App Engine[5] provide *cheap* scalable infrastructure that would be suitable for a project with a restricted budget. These two services take quite a different approach to scalability. Amazon EC2 utilises expandable virtual private servers that can run Apache, PHP and MySQL. Google App Engine on the other hand takes control of all the hardware and software setup and provides a Python API that allows applications to be developed that run on the Google App Engine infrastructure. The scalability of Google App Engine is implicit in that resources are allocated automatically on-demand.

[4] Amazon EC2 - http://aws.amazon.com/ec2/
[5] Google App Engine - http://code.google.com/appengine/

5 JavaScript Client Design

The JavaScript client is sent to web browsers to provide core functionality for the execution of tasks. In addition to the core functionality, the client includes a dynamic performance adjustment algorithm that is discussed in Section 5.4. The tasks that utilise the JavaScript client will be discussed in Section 6.

5.1 Architecture

The JavaScript client is split into three sections, the engine, task and monitor. The engine is responsible for retrieving new work units and sending the results to the server. It is also responsible for starting tasks once a new work unit has been retrieved. The task performs the computation of a work unit, it has two requirements placed on its implementation, it must use timer based programming (Section 5.3) and have a defined halting point. The monitor is responsible for running dynamic performance adjustment (Section 5.4). This runs independently of both the engine and task, although the engine is what first initiates the monitor.

The client-server communication is performed using JavaScript's `XMLHTTPRequest`. This gives the ability to request and send data to and from remote URLs either asynchronously or synchronously. The original implementation of the JavaScript client used the synchronous option due its relative simplicity compared to the asynchronous option. Unfortunately with large work units the browser would become unresponsive during the `XMLHTTPRequest` call, so the asynchronous option is now used. The format used to transfer data between client and server is JavaScript Object Notation (JSON) [31]. JSON is built into the JavaScript language and is consequently a lot easier and faster to deal with than other data formats like XML.

Prototype[6] was used as the base framework for the project. Prototype simplifies a lot of JavaScript's object-orientated functionality making it significantly easier to deal with. In addition all JavaScript was optimised by removing unnecessary characters to reduce the amount of data sent to the user [32].

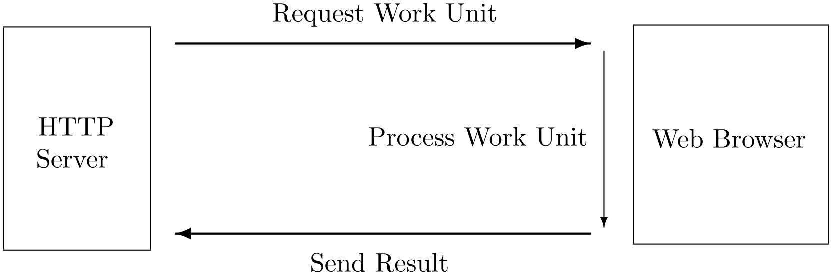


Figure 3: Task Life Cycle

Figure 3 shows the life cycle of a task. When the user first arrives at a page containing parasitic JavaScript a request is made to the server for a work unit. The user is then allocated a work unit and computation of the work

[6] Prototype JS — http://www.prototypejs.org/

unit begins. Once complete, the client sends the result back to the server and requests another work unit.

In addition to returning the result of a work unit back to the server, the client also sends additional information such as the current delay as calculated by dynamic performance adjustment (Section 5.4) and time taken to complete the work unit. This information was used to identify any gains or problems during development.

5.2 Task & Work Unit Considerations

Given the nature of how people use web browsers, some pages might be viewed for a long period of time, such as watching a video on YouTube, or pages might be viewed for a short period, such as browsing search results on Google. This needs to be taken into consideration when designing a task with regard to the length of time required to complete a work unit, but also the size in terms of data the work unit takes to transfer across the Internet.

The ideal situation for Parasitic JavaScript is to be embedded on a website that has longer than average page viewing times. The worst case scenario is a web page that is viewed for a shorter period than the time taken to complete a work unit. This would result in having incomplete work units, which would then need to be added back into the queue. While it is not difficult to reallocate incomplete work units, if Parasitic JavaScript has been embedded onto a website that attracts a large number of users with low page view times, it could become problematic.

Work unit size is also important because many Internet users today are still on slow 56k modem connections. If work units contained more data than the Internet connection is able to carry in a reasonable amount of time, this would impact on the user experience. Given that there is generally a relation between the size of a work unit and the time taken to complete the work unit, a balance needs to be found, such that the user experience is not harmed by transferring large work units and the number of incomplete work units is minimised. This balance could be found by performing tests on a wide range of Internet connections at varying speed.

5.3 Timer Based Programming

One of the issues with JavaScript is that in many web browsers it is run in the same thread as the browser window itself. This can become a problem if the JavaScript being executed takes a long period of time to complete, as the browser will become unusable. In order to help users from being completely locked out of the browser, most modern browsers include a warning as shown in Figure 4. A solution is required to prevent the browser from becoming unresponsive but also to prevent the slow JavaScript error from appearing.

JavaScript does not provide any ability to pause execution like you would in many other languages via a `wait` or `sleep` function. Fortunately JavaScript does provide one method that is similar. The `setTimeout(function, delay)` method takes two arguments, the first is the function you wish to execute and the second is the time to wait until the function is executed. JavaScript timers have several properties that resolve the issues described. They do not make the browser unresponsive, and do not alert the user of long-running JavaScript. In

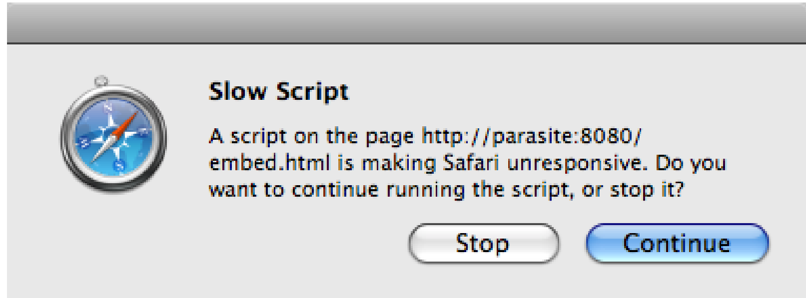


Figure 4: Slow JavaScript warning in Safari

addition, because a delay can be specified, the execution speed of the task can be controlled.

In order to use timer based programming, candidate points (steps) need to be located within a program. Ideal points are loops that contain a large number of complex iterations. As an example the ray tracing task (Section 6.2.1) uses each pixel calculation as the stepping point. One down-side to using `setTimeout` is that it does not allow for parameters to be given to the function that is going to be executed. A simple solution to this is to use global variables or alternatively the function can be a member of a class for which it has access to member variables.

Figure 5 shows the conversion required between a while loop to a timer based programming solution. The most notable change is that the conditional statement has moved inside the function and initiates the `setTimeout` function. *For* loops can be converted in a similar way, with the exception that any iterator values would have to be moved into the global scope or become member variables of a class.

```
while condition do
    ... do something ...
end while
```

```
procedure Step
    ... do something ...
    if condition then
        setTimeout(Step, delay)
    end if
end procedure
```

Figure 5: Example of a loop converted to a timer based loop.

In order for the Parasitic JavaScript framework to have control over the `delay` attribute of `setTimeout`, a wrapper function is provided for timer based programming when creating tasks. The wrapper function is discussed further in Section 6.1.

5.4 Dynamic Performance Adjustment

Computers vary in performance and this needs to be taken into consideration. Consider a website that has been embedded with Parasitic JavaScript. If a user notices that when they visit this website, their computer becomes slow

and unresponsive, what will their response be? It is likely they would not be visiting the website as often. So in order to preserve the user experience, a method needs to be created to ensure that Parasitic JavaScript runs at a speed that the computer is able to comfortably handle.

The speed of execution can be controlled by setting the `delay` parameter in the `setTimeout` function used in timer based programming (Section 5.3). While this does not slow down the execution of every instruction, it does as a whole decrease the speed at which the task is being executed. This also allows for the dynamic performance adjustment to be implicit. If tasks are created using timer based programming (Section 5.3), then they automatically obtain dynamic performance adjustment.

JavaScript does not have the ability to query the current load on a computer or even the CPU configuration. Without the ability to directly query this information, it must be estimated from the environment. There are potentially two methods of doing this. The first is to evaluate work unit completion times on a wide range of hardware. This would then give a distribution of performance measurements for which Parasitic JavaScript could adjust the `delay` parameter accordingly. The major problem with this is that every task would need to be evaluated and considering the wide range of hardware available, it could be a daunting job.

The second potential solution is to utilise an observation made during the testing of timer based programming. The `setTimeout` call often had a scheduling delay, resulting in the function being executed several milliseconds after the requested time. The scheduling delay could be used to estimate the CPU capacity or even load.

The route taken was to use the `setTimeout` method to estimate the performance of the host computer based on the observed scheduling delay. The implementation involves periodically monitoring the scheduling delay and adjusting the `delay` parameter used in timer based programing.

Algorithm 1 illustrates the dynamic performance adjustment algorithm. The δ constant is the initial value that the delay is set to when the user first loads a website containing Parasitic JavaScript. The value used currently is 20 milliseconds but there is no reason this could not be customised for each user, such that the value is set to its average calculated value for a particular user. The β constant is the rate that the dynamic performance adjustment algorithm is executed. The θ variable indicates the threshold for the difference in delay. Currently θ is set to be $\frac{5}{4}\beta$ because the threshold needs to be larger than the β value as the time difference includes the β value. α is the threshold at which a delay adjustment should occur.

The `OverTime` variable is used to count situations when the `setTimeout` function was over the θ threshold or when the timer fires early. Likewise, the `OnTime` variable is used to count situations when it was not over the threshold. In order to trigger a delay change the value of `OverTime` or `OnTime` must exceed the α threshold. The algorithm eventually stabilises the `Delay` value after a period of time. Once stabilised the algorithm is also able to adjust to changes in the environment such as increased load.

The algorithm works by exploiting both the fact that Parasitic JavaScript is running and the very nature in which web browsers are designed. Due to the single-threaded nature of JavaScript only one piece of code can be executed at any one time. This is the same for timers. Because two timers can potentially

Algorithm 1 Dynamic Performance Adjustment

```
Delay = δ
procedure DPA(StartTime, OverTime, OnTime)
    diff = CurrentTime() - StartTime
    if diff > θ ∨ diff < θ − β then
        OverTime++
        OnTime = 1
    else
        OnTime++
        OverTime = 1
    end if
    if OverTime > α then
        OverTime = 1
        Delay++
    end if
    if OnTime > α then
        OnTime = 1
        Delay−−
    end if
    SetTimeout(DPA(CurrentTime(), OverTime, OnTime), β)
end procedure
```

fire at the same time, web browsers must place them in a priority queue and they are executed when execution space becomes available [33, 34]. This is different from how timers would be created in many other languages, using CPU timer interrupts [35]. But it is this difference that allows for this type of estimation.

Figure 6 shows an example execution space where a dynamic performance adjustment (DPA) timer has been initiated to execute after 20ms, but in this case the Parasitic JavaScript task is running, which results in the timer being delayed for 10ms. Parasitic JavaScript is in-effect overloading the available execution space. Slower computers are unable to process the execution queue fast enough, and consequently timers are unable to fire, becoming delayed. By monitoring this delay, the execution speed can be lowered to a point where the web browser is capable of clearing the execution queue. Alternatively the execution speed can be increased in situations where there is no delay.

5.5 Embedding the JavaScript Client

In order for Parasitic JavaScript to be run on a website, it must be first embedded into the website. Unfortunately JavaScript has several security restrictions that need to be overcome. JavaScript restricts `XMLHTTPRequest` calls, such that it will only request a URL from the same domain the website is running on. For example, if a user visits the website `example.org` and this website attempts to make an `XMLHTTPRequest` to `example.com` it would be blocked.

An `IFRAME` is an HTML element that creates an inline frame that allows for an HTML document to be embedded into another HTML document. The contents of an `IFRAME` are kept separate from the parent document and consequently does not inherit the same cross-domain restriction from its parent.

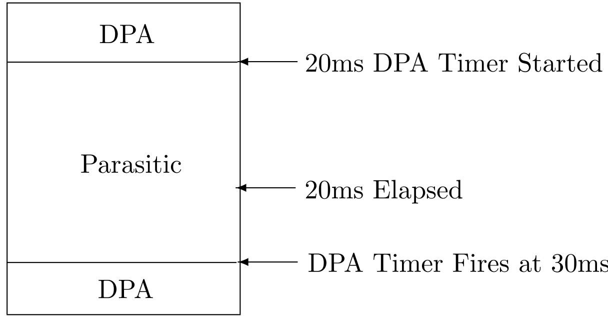


Figure 6: Web browser execution space

Cross-domain restrictions still apply though, so if the `IFRAME` is embedded from the remote URL `example.com` it may only make `XMLHTTPRequest` requests to `example.com`.

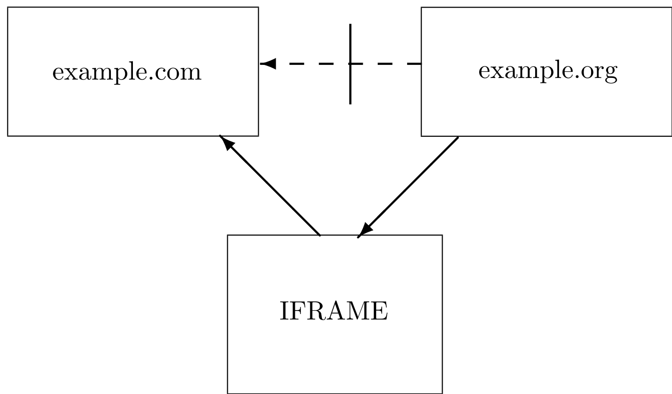


Figure 7: Using an IFRAME to bypass JavaScript cross-domain security

Figure 7 demonstrates a solution using the `IFRAME` element. The website `example.org` has been embedded with Parasitic JavaScript inside an `IFRAME`. The Parasitic JavaScript is then able to communicate to the parasitic server to request work units and send results back.

```
<iframe src="http://parasitic-server.tld/emtask/1/"
 style="display:none;" width="0" height="0"></iframe>
```

Figure 8: HTML to embed Parasitic JavaScript

Figure 8 shows the required HTML to embed a Parasitic JavaScript task. Once embedded within a website, the task will begin processing work units until the user navigates to another page. The `IFRAME` is also invisible to the user.

6 Tasks

Tasks describe a solution to a problem, they process work units and prepare the results. This section covers the creation of tasks and the tasks that were implemented during this project.

6.1 Creating a Task

One of the goals of the project was to allow for tasks to be easily created. This has been accomplished by applying two restrictions on what tasks must programmatically contain. These restrictions are that a task uses timer based programming and that the task has a defined halting point. In return tasks automatically obtain dynamic performance adjustment (Section 5.4) and client-server communication is performed implicitly.

Algorithm 2 Counting Task

```
var CountingTask = Class.create(ParasiteTask, {
   run: function() {
      this.result.count = this.state.count;
      this.step();
   },
   step: function() {
      this.result.count++;
      if(this.result.count > 100) {
         this.stop();
      } else {
         this.start_step(this.step.bind(this));
      }
   }
})
engine = new ParasiteEngine(new CountingTask());
engine.start();
```

Algorithm 2 shows the JavaScript code to create a task that counts to 100. The task extends the `ParasiteTask` class that defines several methods. The `this.stop()` method defines the halting point as required by a task, once called the work unit result is prepared and sent to the server. The data that is sent to the server is collected from the `this.result` variable, which can contain arbitrary data that can then be converted to JSON for transmission. The `this.start_step()` method is a wrapper around `setTimeout`, this integrates dynamic performance adjustment. One of the issues with the `setTimeout` function is that it executes the function given in the global scope. Consequently member variables of classes would be inaccessible. In order to get around this, JavaScript has the concept of binding a variable to its scope. This is performed by the `this.step.bind(this)` function. This effectively wraps the `this.step()` function inside another function, which is executed within the scope of the class.

Prior to the `run()` function being executed, the `this.state` variable is populated with the work unit information. In this case it would be populated with an initial value for the counter. Once the `this.result.count` variable exceeds

100, the value of `this.result.count` is sent back to the server and an additional work unit is retrieved.

6.2 Tasks Implemented

Three tasks were implemented to demonstrate a range of functionality that JavaScript is capable of performing. These three tasks cover three different areas of computer science. Ray tracing (Section 6.2.1) covers both the rendering of images and mathematical calculation, the perceptron (Section 6.2.2) covers data processing and data mining, and n-queens (Section 6.2.3) covers search using genetic algorithms.

6.2.1 Ray Tracing

Ray tracing [36] is an algorithm for rendering 3D objects. The ray tracing algorithm computes one pixel per iteration and does not require knowledge of previously computed pixels. This makes it ideal for distribution as work units can be separated into groups of pixels. While ray tracing is not necessarily a practical example, it demonstrates JavaScript's ability to compute something it was never designed to compute.

Figure 9 gives an overview of how the ray tracing algorithm works. The camera traces a ray that is emitted from the camera (as opposed to the light source, which is possible but not efficient). The ray is tested for intersection with other objects within the scene. If the ray does intersect an object, the closest intersected object to the camera is found. An additional ray is then sent to the light source(s) and shadowing information is calculated. In the event the ray does not intersect an object, the background colour is used.

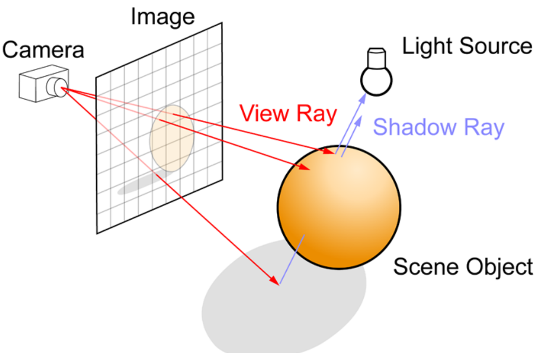
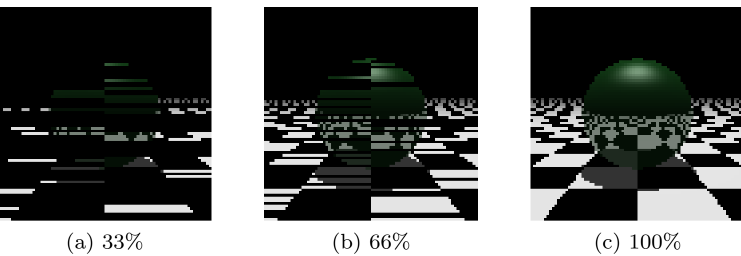


Figure 9: Ray Tracing Algorithm[7]

The JavaScript implementation stored the model of the image within the code. This was done to reduce the overall work unit size. While it is possible to send the model with a work unit, it is more efficient to store it within the code as it does not need to be transferred with every work unit. The work units contained the section of pixels to compute. Work units would range from 50 to 200 pixels depending on the size of the image. The data sent back upon completion of a work unit contained the hexadecimal colour values of the pixels. Work

[7]Image from of http://en.wikipedia.org/wiki/Image:Ray_trace_diagram.svg

units are pre-generated with the number of work units being determined by the size of the image. Figure 10 shows the result of the ray tracing distribution at three stages: 33% completed, 66% completed, and the final result.



(a) 33% (b) 66% (c) 100%

Figure 10: Stages of Ray Tracing

6.2.2 Perceptron

The perceptron [37] is a simple neural network used in machine learning. The perceptron alone is not an ideal candidate for distribution as it requires previous knowledge of past calculations. In order to allow for more streams of computation cross validation was used. Cross validation [38] is a technique used in the evaluation of machine learning algorithms. The experiment is run multiple times with subsets of data used for training and evaluation. The final accuracy measurements are taken by averaging the accuracy of all the experiments run. This makes cross validation an ideal solution for allowing multiple users to contribute to the data mining task at any one time.

Algorithm 3 shows the basic perceptron algorithm, α is the learning rate and the ActivationFunction can be either a sigmoid or threshold function. The algorithm iterates over the rows of attributes and adjusts the weights depending on whether there was an error in classification. Generally perceptron implementations continue iterating over the data set until there are no errors. This is not the case in this situation due to the impracticality of sending the entire data set within a single work unit. Consequently the resulting weights from this calculation are then used to initialise the weights for the next portion of the data set in another work unit.

The perceptron task used an artificially created data set, the data set was designed to be linearly separable to ensure the algorithm would terminate. Each work unit contained a partial amount of this data set. Work units were not fully pre-generated like in ray tracing (Section 6.2.1) but rather an initial set was generated and upon completion of a work unit, another work unit was created. This was due to the algorithm requiring weight information from a previous work unit. Evaluation of the results is performed server-side as this does not take a significant amount of time.

6.2.3 N-Queens

The n-queens [39] puzzle involves placing n queens on an $n \times n$ size chess board such that none of the queens are able to attack each other. The n-queens problem is interesting because it is NP-hard, but also because there exists a volunteer computing project, NQueens@home [40] that allows users to help find

Algorithm 3 Perceptron

```
weights = []                                          ▷ Initialised from work unit
for all data rows as attributes, target do
    input = 0
    for all attributes do
        input += weights_i × attributes_i
    end for
    error = target - ActivationFunction(input)
    if |error| > 0 then
        for all attributes do
            weights_i += α × error × attributes_i
        end for
    end if
end for
```

solutions for increasing board sizes. Figure 11 shows a solution to an 8×8 board, but it is not the only solution. In total there are 92 possible solutions to the 8×8 board. At 15×15 there are already 365,596 possible solutions. A genetic algorithm was used to locate solutions.

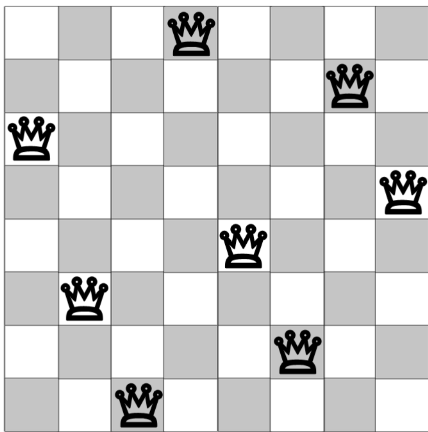


Figure 11: 8-Queens Solution

While genetic algorithms are not the most efficient method of finding solutions to n-queens, they are an interesting approach. The basic genetic algorithm consists of four steps, population generation, population evaluation, breeding and mutation. In terms of distribution, genetic algorithms allow for a great reduction in the amount of work unit traffic required. Because all the steps can be performed within the client, the only time data needs to be sent to the server is when a result is found.

Algorithm 4 shows the basic algorithm used for the n-queens solver. The population consists of a sequence of numbers representing the y-axis position. The x-axis position is determined by the order in which the numbers occur. The Breed function takes two members of the population and splits the sequence of numbers at a random point. The split sequences are then swapped around

Algorithm 4 Genetic Algorithm

```
Population = GeneratePopulation(n)
Solutions = []
for all Population as x_i do
    Child1, Child2 = Breed(x_i, x_{i+1})
    if IsSolution(Child1) then
        Solutions.add(Child1)
    end if
    if IsSolution(Child2) then
        Solutions.add(Child2)
    end if
    if α > Random() then
        Mutate(Child1)
        Mutate(Child2)
    end if
    Population.add(Child1)
    Population.add(Child2)
    CullPopulation()
end for
```

to produce two new children. Each child is then checked to see if they are a solution to n-queens, in which case they are added to the list of solutions.

The next step is to provide some form of mutation. α is the mutation rate, which is currently set to 0.01. The mutation rate specifies that probability that a mutation occurs. In the event mutation does occur, the Mutate function is called which randomly selects a single y-axis position to modify to a random value. While the single point mutation is sufficient for low values of n, higher values could require multiple mutations. The children are then added to the population.

The CullPopulation function evaluates each member of the population and sorts them according to an evaluation function. The evaluation function returns a numeric value indicating the total number of possible attacks for the given queen positions, a value of zero would indicate a solution. The members of the population which represent the worst 40% according to the evaluation function are removed. Replacements are then randomly generated via the GeneratePopulation function.

Genetic algorithms can be used to solve a wide range of problems, and consequently this n-queens problem can be modified to solve other problems. But more importantly, genetic algorithms are also inherently parallel [41], which makes it an ideal candidate for Parasitic JavaScript tasks. An additional benefit is the significant reduction in work unit sizes and the relative fault tolerance of genetic algorithms.

There are many alternate ways to distribute a genetic algorithm [42], and the n-queens task is by no means the most effective. The algorithm could be improved by including a sample of the population within the result of a work unit. This would then allow for population seeding within work units, which could ultimately increase the rate at which the genetic algorithm finds solutions.

7 Evaluation of Dynamic Performance Adjustment

Dynamic performance adjustment (Section 5.4) was evaluated to determine how effective it is at determining a delay in relation to the hardware specifications of a computer. In addition to this, the ability of the algorithm to adjust to performance changes on the computer dynamically was also evaluated.

In order to evaluate the delay in relation to the hardware specifications of a computer, four computers with a range in specifications were used. Table 1 lists each computer and their specifications. The age of the computers ranges from months to eight years old.

	Operating System	**CPU**	**Memory**
A	Windows XP	Intel Core 2 Quad 2.5GHz	3GB DDR2
B	Mac OS X 10.5	Intel Core 2 Duo 2.4GHz	2GB DDR2
C	Mac OS X 10.4	PPC G4 1.33GHz	768MB DDR
D	Mac OS X 10.4	PPC G4 400MHz	448MB SDRAM

Table 1: Computers used for evaluation

Mozilla Firefox 3 was used to perform the evaluation, primarily because it runs on both Windows and Mac OS X. Due to the similarity in which the JavaScript `setTimeout` function is implemented among browsers [33, 34], only using a single web browser to evaluate dynamic performance adjustment was sufficient.

The experiment setup consisted of a dedicated server for work unit distribution that was connected to the client over a local area network. The task the client performed was ray tracing (Section 6.2.1). During the experiment the delay value calculated by dynamic performance adjustment was recorded every five seconds for a period of two minutes, resulting in 24 values. The experiment was repeated five times for each computer. The average of each five second interval was taken from the five repetitions.

The algorithm was tested under two different conditions, the first without any additional applications running and no additional websites open within the web browser. The second was to simulate a typical user environment: several web browser windows are open directed at several websites. The websites were: YouTube.com, a video sharing website; Facebook.com, a social networking website; and Slashdot.org, a news website. In addition to these websites, the website containing Parasitic JavaScript was open in a window. YouTube was the primary window and was playing a video.

A second experiment was performed to determine how well dynamic performance adjustment performed under varying load conditions. This experiment was run for seven minutes under the same conditions as the typical load in the first experiment, with the difference being that the YouTube video was stopped and started in an alternating fashion for one minute intervals. This resulted in four no load periods and three typical load periods. The experiment was repeated five times and the average of each five second interval was taken from the five repetitions.

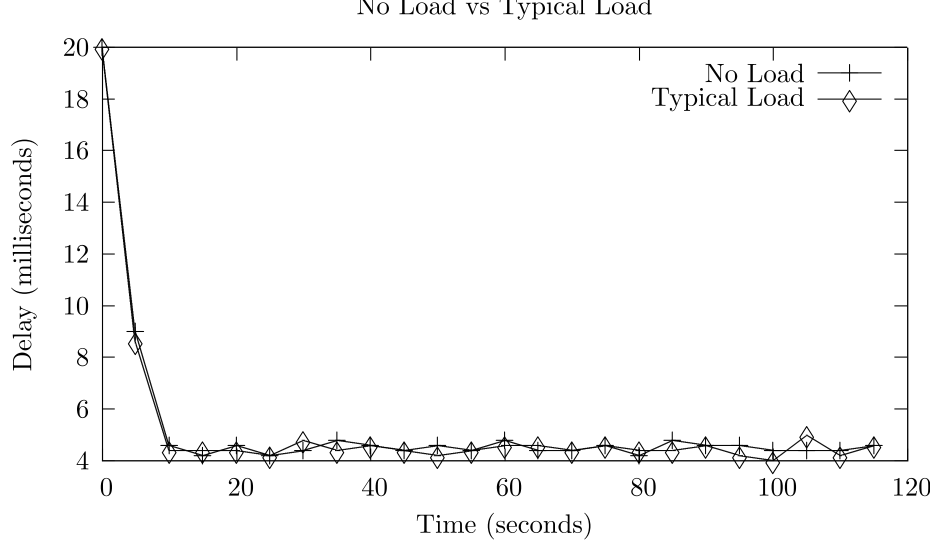


Figure 12: Computer A. Calculated delay of no load and typical load conditions.

7.1 Results

The results for the two fastest computers (A & B as in Table 1) are shown in Figures 12 and 13. These graphs show the delay calculated by dynamic performance adjustment over a period of two minutes. The two lines represent the no load and typical load experiments for each computer.

The results were as expected for the computers A and B with a small delay value being calculated for both the no load and typical load experiments. Both of these computers are several months old and consequently should easily be able to handle Parasitic JavaScript, which is clearly shown in the graphs.

The two slower computers (C & D) as shown in Figures 14 and 15 fall into the range that dynamic performance adjustment is targeted at. These computers are several years old and represent what might be your standard family computer. An interesting result that both computers C & D produced was that under the no load experiment, the delay values remained low. While this was unexpected, it does indicate that these computers are capable of running Parasitic JavaScript at a reasonable speed when the computers are not in use.

The typical load experiments show the impact of dynamic performance adjustment. In both cases the average delay was significantly higher than that of the no load experiment, with computer C having an average delay of 16.56 milliseconds and computer D having an average of 37 milliseconds. These results indicate that for older computers dynamic performance adjustment was able to adjust the execution speed of Parasitic JavaScript depending on the current computing environment. There is also a clear distinction between the difference in hardware specifications of each of the computers, with computer C, the faster of the two computers having a lower delay value.

The results of the variable load experiment are shown in Figures 16 and

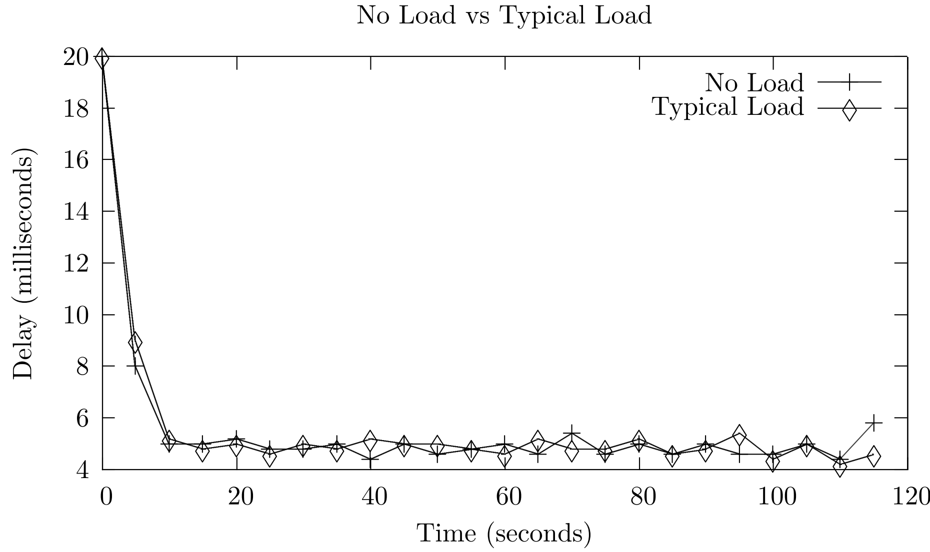


Figure 13: Computer B. Calculated delay of no load and typical load conditions.

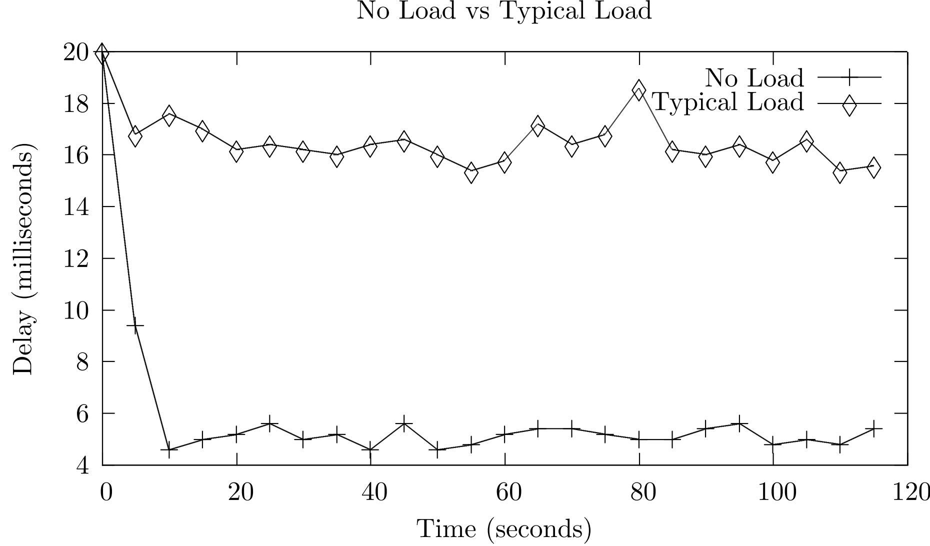


Figure 14: Computer C. Calculated delay of no load and typical load conditions.

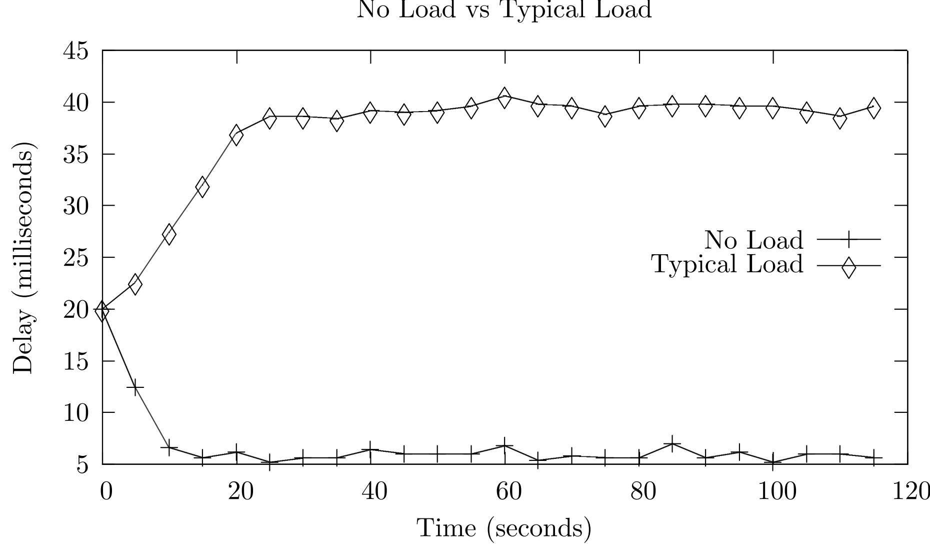


Figure 15: Computer D. Calculated delay of no load and typical load conditions.

17. The results for computers A & B are not shown here, as the results were very similar to those in Figures 12 and 13, which was expected. The results clearly show the one minute intervals of typical load, with computers C & D both producing vivid peaks between the intervals. The peaks demonstrate how the dynamic performance adjustment algorithm was able to adjust the delay during variable load.

An interesting difference between the two computers is the steepness shown in the peaks. Computer C has a rapid increase in delay, whereas computer D has a slower increase in delay. This indicates that on slower computers the time taken to increase the delay to a stabilised level takes longer. In order to improve the responsiveness of the delay modification it might be suitable to increase the modification occurring for situations where the delay has been increasing or decreasing constantly.

Overall the results were excellent, with dynamic performance adjustment increasing the delay on the slower computers C & D during the typical load experiment and variable load experiment. The faster computers A & B produced low delay values during both experiments, which was expected.

The results shown here do not only indicate that dynamic performance adjustment is suitable for Parasitic JavaScript, but also that dynamic performance adjustment may have applications in general web development. The popularity of interactive web applications has been increasing since the advent of AJAX, but the impact these services have on low-end computers is not generally taken into account. Giving web developers the ability to adjust the rate at which their web applications are executed may help improve the user experience for users who do not have the latest computer available.

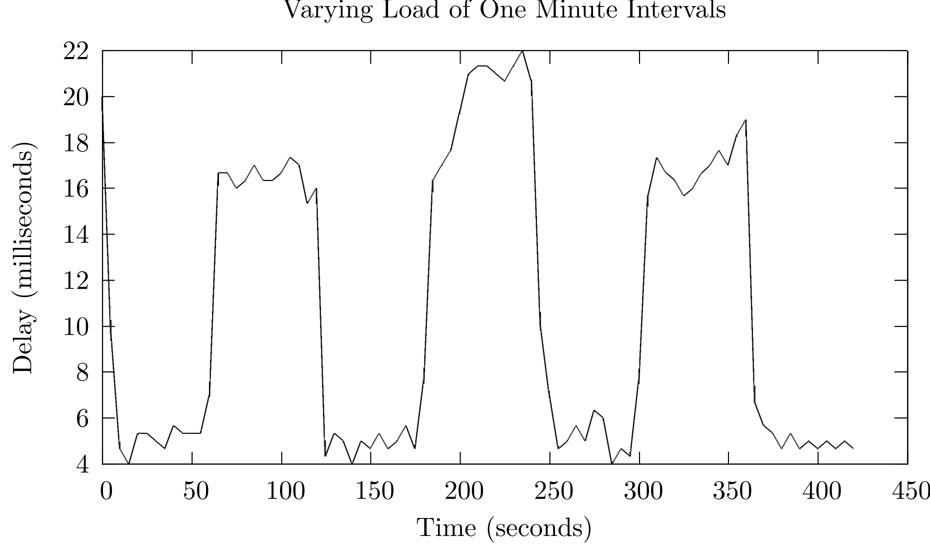


Figure 16: Computer C. Calculated delay during variable load conditions of one minute intervals.

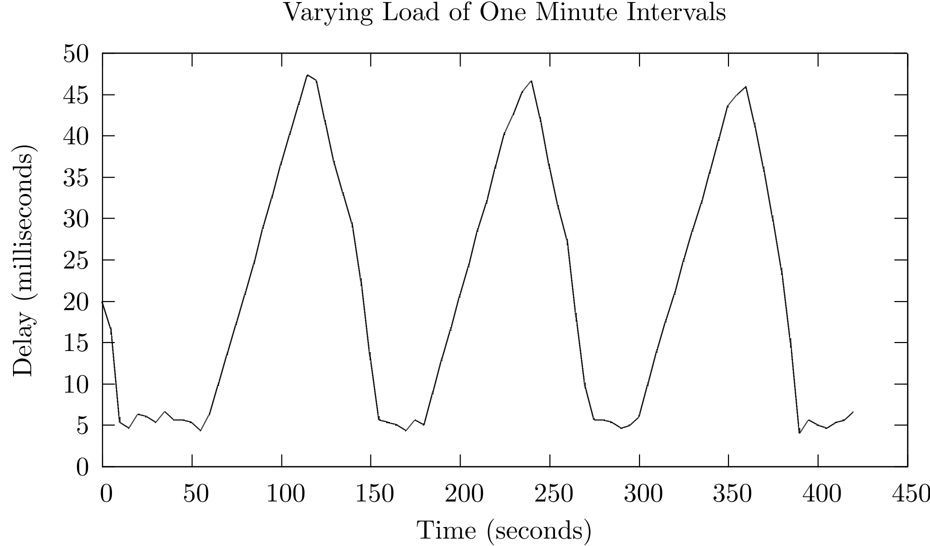


Figure 17: Computer D. Calculated delay during variable load conditions of one minute intervals.

8 Performance

Parasitic JavaScript works on a range of web browsers, but they all have slightly different ways of implementing JavaScript and as such their performance differs. The purpose of this section is determine how well Parasitic JavaScript runs on a variety of web browsers, what the overheads of distribution are and to explore possible methods of improving performance.

8.1 Browser Performance

The browser performance experiment aimed to find the browser that was able to complete the most ray tracing (Section 6.2.1) work units over a period of a minute. The performance experiment tested five modern web browsers which are shown in Table 2. Google Chrome is a new web browser that represents a shift in the way JavaScript is typically executed. In the other web browsers JavaScript is parsed and executed using a recursive tree parser. In Google Chrome this is not the case, JavaScript is compiled in real-time to native machine code [43].

Name	**Version**
Google Chrome	0.3
Mozilla Firefox	3.0.3
Opera	9.6
Safari	3.1.2
Internet Explorer	7.0

Table 2: Web browsers used for performance testing.

The experiment consisted of running the ray tracing task for twelve minutes on each of the five web browsers and averaging the one minute intervals. The first and last one minute intervals were discarded. Ray tracing was configured to have a work unit size of 100 pixels. The experiments for Google Chrome, Mozilla Firefox, Opera and Internet Explorer were all performed on the same computer running Windows XP. Safari was tested on a computer running Mac OS X. The two computers are several months old and have similar specifications. There were no additional applications running during the experiment and only Parasitic JavaScript was open in each web browser.

8.1.1 Results

The results for the browser performance experiment are shown in Figure 18. The graph shows the average number of work units completed over a period of one minute for each of the tested web browsers. Google Chrome clearly outperforms the other web browsers, producing more than twice the number of work units per minute. Firefox and Safari are equal in terms of performance completing 51.9 and 51.1 work units per minute respectively. Opera and Internet Explorer were the worst performing of the five web browsers, completing 34.8 and 35.9 work units per minute respectively.

Google Chrome represents a significant shift in the way JavaScript is executed in web browsers. The increased usage of JavaScript in web-based applications has brought on a need to improve the performance of JavaScript. The

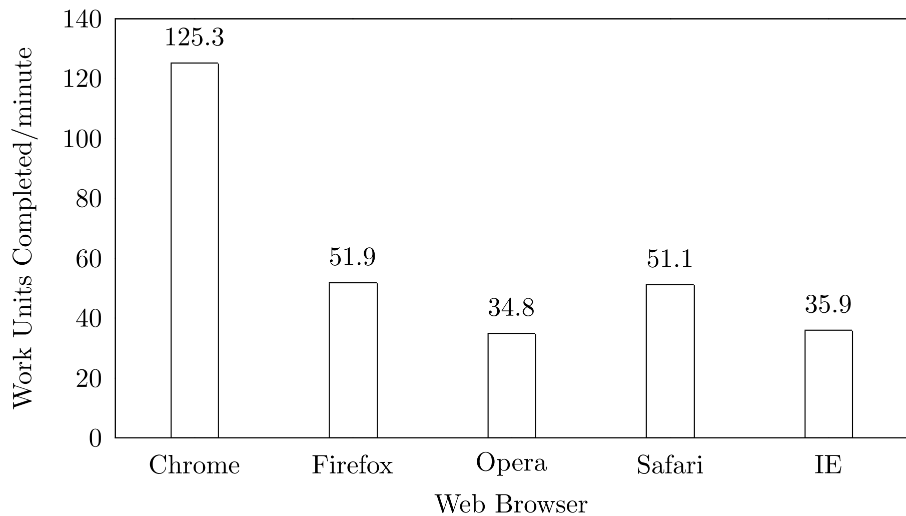


Figure 18: Average work units completed per minute for each web browser

result of this is that the creators of Mozilla Firefox, Safari, and Internet Explorer all have improved JavaScript engines in development that aims to match or improve on the performance Google Chrome achieves.

As the JavaScript engines of these web browsers do get faster it improves the viability of Parasitic JavaScript. This is because having faster JavaScript engines both allows for increased work unit completion rates, but more importantly it allows for an increase in work unit size. This then allows for a reduced amount of communication between the server and client, improving the communication-computation ratio and ultimately reducing cost.

8.2 Distribution Overhead

Distributing a task often has some overhead associated with it: in Parasitic JavaScript the overheads include retrieving a work unit, parsing it, packing the result and sending it. In order to determine how large this overhead is an experiment was conducted with varying work unit sizes for the ray tracing task (Section 6.2.1).

The experiment setup consisted of calculating the completion time for ray tracing an image that has 1024 pixels. The number of pixels contained in a ray tracing work unit determines its size. A logarithmic scale was used to determine the number of work units for a given image, which ensured the sizes of each work unit were equal for a given image. Ten work unit sizes were tested given by the range, $[2^1, 2^2, ..., 2^{10}]$. As an example the image for 2^2 contained four work units, each work unit computing $\frac{1024}{2^2} = 256$ pixels.

Mozilla Firefox 3 was used to perform the experiment running on Windows XP and a Quad Core 2.5Ghz CPU with 3GB of RAM. The server (Section 4) was connected over the local network. This experiment does not take into

consideration network speed, which would certainly play a role in the completion times of work units in an environment where Parasitic JavaScript was running over the Internet.

The start and completion times of each work unit were recorded by the server. The overall completion time was calculated by taking the start time of the first work unit issued and the completion time of the last work unit issued. During this experiment random work unit assignment was disabled such that work units were issued in order. The experiment was repeated ten times and the results for each work unit size were averaged.

8.2.1 Results

The results for the distribution overhead experiment are shown in Figure 19. The graph shows the completion times in seconds for each of the work unit sizes. The results clearly indicate that there is a distribution overhead, with the experiment with one work unit taking 11 seconds and the experiment with 1024 work units taking 61 seconds.

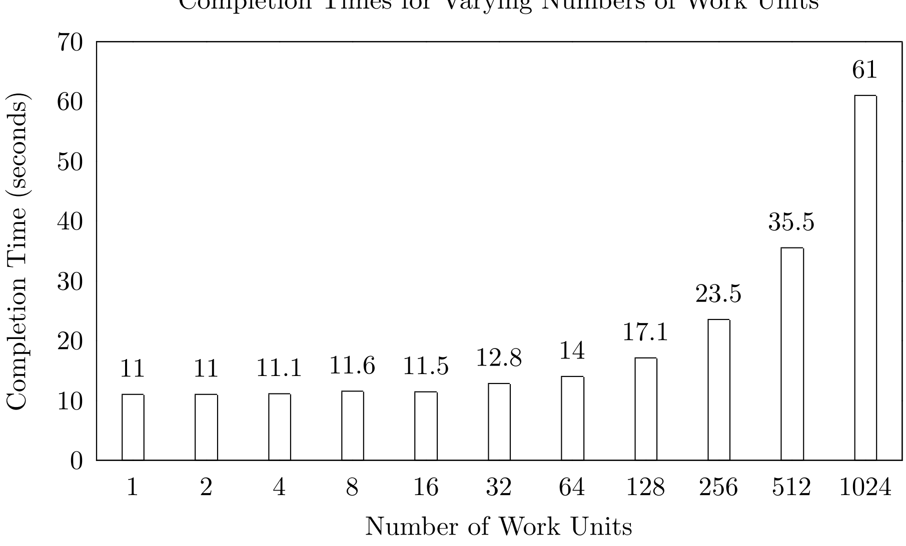
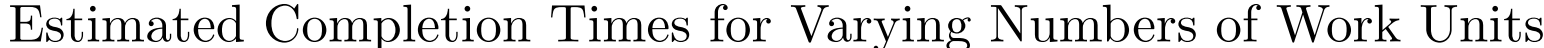


Figure 19: Completion times with varying numbers of work units for ray tracing an image containing 1024 pixels

It is possible to estimate the overhead of a single work unit by taking the experiment with 1024 work units. The 1024 work units were completed in 61 seconds. By looking at the experiment with one work unit it can be determined that the computation takes about 11 seconds. This gives 50 seconds of distribution overhead for the 1024 experiment. This implies that a single work unit requires $\frac{50}{1024} = 48.83ms$ of overhead. Figure 20 shows an estimation of the overall experiment based on the 48.83ms value. The graph is almost identical to Figure 19 which indicates that the distribution overhead is linear in comparison to the number of work units.



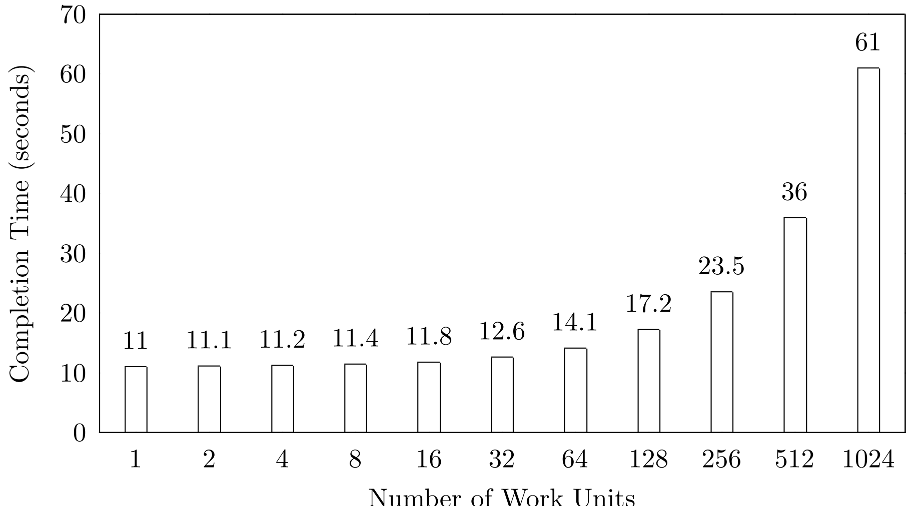


Figure 20: Estimated completion times with varying numbers of work units for ray tracing an image containing 1024 pixels

8.3 Improving Performance

In order for Parasitic JavaScript to be a viable distribution method the communication between the client and server needs to be reduced, but also the computation the client performs needs to be maximised. The communication aspect represents the size and amount of work units being sent to clients, while the computation represents the amount of computation performed.

Communication can be reduced by modifying the size of work units, but this is not necessarily solving the problem. The simplest approach is compression, many web servers include functionality to enable automatic compression of data being sent from the server, the down side is that JavaScript does not provide any native compression utilities to compress result data. Another approach is to design work units such that they contain the smallest amount of data possible for the computation to occur. Ultimately, while these methods may have some merits the gains in most cases would not be significant compared to the possible gains in computation.

Computation provides several avenues for improvement, be it improving the efficiency of an algorithm or utilising plugins like Google Gears[8]. Google Gears is an extension to Mozilla Firefox, Internet Explorer and Google Chrome that includes several improvements to JavaScript. The most relevant feature is allowing for JavaScript to be threaded. This would reduce the requirement for timer based programming (Section 5.3) and dynamic performance adjustment (Section 5.4) which have an overhead due to the `setTimeout` delay. The only issue with Google Gears is that the user is required to install it, but in situations where it has been installed it is possible to utilise the threading functionality.

[8]Google Gears - http://gears.google.com/

9 User Study

One of the goals of the project was to create a system that did not interfere with the user experience. In order to evaluate whether this was accomplished a user study was designed to test the users ability of being able to detect Parasitic JavaScript. In addition to the user study a questionnaire was included to establish the views people have with regard to Parasitic JavaScript. This user study was given ethical consent, which can be viewed in Appendix A.1.

9.1 Design

The goal of the user study is to establish whether users are able to identify websites containing Parasitic JavaScript. This is inherently difficult to do because the participants are not necessarily aware of potential side effects Parasitic JavaScript might have. In order to identify a suitable user study, a pilot study was performed, which allowed several issues to be identified with the initial design.

The user study was first designed so a participant was given a series of twelve websites that had been randomly embedded with Parasitic JavaScript. After browsing each page the participants were asked if they thought the page contained Parasitic JavaScript. They were able to answer “Yes”, “No” or “Don't Know”. Once answering the question they were told the correct answer. After viewing the twelve unique websites they would then be shown their overall results. During the pilot study, this method was found to be confusing as participants found it difficult to compare the completely different websites. This resulted in several changes to the user study design.

The final design consisted of the participants viewing three websites, YouTube.com, a video sharing website; NZHerald.co.nz a news website; and Wikipedia.org, an online encyclopedia. The websites were selected because they all represent what would be an ideal platform for Parasitic JavaScript. This is because each of these websites have a large amount of content, be it a video or an article. Parasitic JavaScript is more effective on websites with long page view times, as this allows for more work units to be computed.

The participants viewed each website five times, each time Parasitic JavaScript had a fifty percent chance of being embedded into the website. Participants would proceed by answering the question “Do you think this page contains Parasitic JavaScript?”, for which they were able to answer “Yes”, “No”, or “Don't Know”. Figure 21 shows the screen participants were given, with the question at the top of the window and the website they were evaluating at the bottom.

In order to allow participants to compare between websites, they were given another window displaying the current website they were reviewing that did not contain Parasitic JavaScript. This window was created in a separate web browser process to ensure there was no interference between the two windows.

During the experiment participants were able to browse freely around the website, with the exception and refrain from using the “View Source” feature of the web browser. The reason for not allowing users to use the “View Source” feature, which allows them to view the source code of a website and consequently see the Parasitic JavaScript, was because the user study is about the user experience rather than their technical ability.

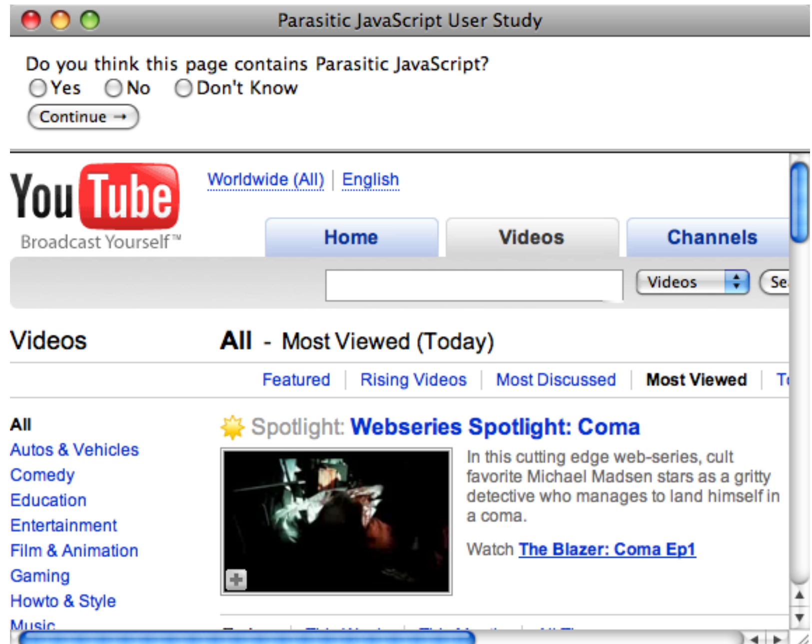


Figure 21: Example of the user study window

9.2 Questionnaire

The goal of the questionnaire was to determine if the users felt the user experience was harmed and their views on real-world deployments of Parasitic JavaScript. The questions targeted both charitable and commercial applications of Parasitic JavaScript, where the charitable example was related to donating computing time to a good cause, and the commercial was enabling Parasitic JavaScript in exchange for some service.

The questionnaire was web based and contained six questions which were either yes/no answers or multiple choice answers. The questions asked were:

- For the websites that included Parasitic JavaScript, do you think the user experience was hindered or impaired?
- Would you mind having websites use Parasitic JavaScript?
- If you visited a website that included Parasitic JavaScript and gave you the option to disable or enable it, what option would you pick?
- Parasitic JavaScript can be used to compute many things. Would you be more inclined to accept Parasitic Javascript if it was performing a charitable task?
- If you visit a website running Parasitic Javascript, do you think its important they let you know what they are calculating?
- Do you think it would be acceptable if YouTube.com (or another video website) used Parasitic Javascript in exchange for allowing users watch videos?

A copy of the questionnaire given to participants has been included in Appendix A.2.

9.3 Procedure

The experiment consisted of eight participants who were all undergraduates or graduates in computer science. The experiment was restricted to people with a background in computer science primarily because the task being performed is one of perception. The participant has to perceive that a website contains Parasitic JavaScript and as such they need to have an understanding of the background behind Parasitic Javascript and what it aims to achieve.

Participants were either provided with a computer or were able to use their own. In all cases the server (Section 4) was on a local area network. This was to ensure that Internet connection speed did not play a role in the user study. While in real-world situations network traffic is definitely a concern, it is out of the scope of this user study.

The experiment was setup as described in Section 9.1. In addition to this participants were required to sign a research consent form. Participants were given an explanation of Parasitic JavaScript and instructed on the reason for the user study. The participants were then able to begin the experiment, which took about fifteen minutes.

Once the participant finished the experiment they were given the questionnaire with six questions relating to the user experience, their acceptance of Parasitic JavaScript and their ethical views of Parasitic JavaScript. Participants were also encouraged to discuss any additional views or concerns they had about Parasitic JavaScript.

9.4 Results

The results of the user study and questionnaire are outlined below. The number of participants for this experiment was low at only eight, keeping this in mind the results below may not be representative of the general population especially with regard to the questionnaire.

9.4.1 User Study

The eight participants of the user study evaluated each website a total of forty times (five times per participant). The aggregated results for the user study are shown in Table 3, which shows the total number of times a website contained Parasitic JavaScript and the number of times each possible answer was selected. Due to Parasitic JavaScript being assigned randomly to websites the number of times it was embedded was not equal for each website. Overall the majority of participants answered "No" more often than "Yes" or "Don't Know".

During the experiment it became clear that many of the participants were treating the user study as a competition, that is they wanted to get the correct answer as many times as possible. This resulted in participants carefully examining the differences between the websites, rather than just rushing through the experiment.

Table 4 shows a breakdown of the aggregate results with the results for websites containing and not containing Parasitic JavaScript being split. The

Website	Parasitic JavaScript	Yes	No	Don't Know
YouTube	18	9	21	10
Wikipedia	21	9	26	5
NZ Herald	20	12	21	7

Table 3: Aggregated results of participants in the user study

main observation that can be seen in this table is that there appears to be no connection between the website containing Parasitic JavaScript and the total number of results for each possible answer.

Website	Parasitic JavaScript	Yes	No	Don't Know
YouTube	No	5	10	7
	Yes	4	11	3
Wikipedia	No	5	12	2
	Yes	4	14	3
NZ Herald	No	8	9	3
	Yes	4	12	4

Table 4: Breakdown of results by pages containing Parasitic JavaScript or not

The overall accuracy of the participants is shown in Table 5. The table shows three different accuracy values for each website, with an overall combined accuracy. The "Yes Accuracy" represents correct answers for websites containing Parasitic JavaScript, likewise the "No Accuracy" represents the correct answers for websites not containing Parasitic JavaScript. The combined accuracy represents both websites containing and not containing Parasitic JavaScript.

The results indicate that participants were only able to correctly identify pages that contained Parasitic JavaScript 20% of the time, while they were able to identify pages that did not contain Parasitic JavaScript 51% of the time. The results here clearly indicate that the participants were not able to identify pages containing Parasitic JavaScript accurately.

Website	Yes Accuracy	No Accuracy	Combined Accuracy
YouTube	0.22	0.45	0.35
Wikipedia	0.19	0.63	0.40
NZ Herald	0.20	0.45	0.33
Total	**0.20**	**0.51**	**0.36**

Table 5: Accuracy of the participants at predicting which pages contained Parasitic JavaScript

Overall the results of the user study indicate that participants were unable to accurately identify websites containing Parasitic JavaScript. In addition to this there appears to be no correlation in the results between pages containing and not containing Parasitic JavaScript. Although these results were obtained from a sample size of only eight participants the accuracy of all the participants

was low, with the highest individuals combined accuracy being 47%. The lowest combined accuracy was 20%.

If the user study was to be repeated it again, changing the question being asked to participants from "Do you think this page contains Parasitic JavaScript?" to "Can you notice any difference between the page not containing Parasitic JavaScript and this page?" may allow for a larger audience of participants as they would not need to understand Parasitic JavaScript.

9.4.2 Questionnaire

The questionnaire gave an insight into what the participants of the user study thought about Parasitic JavaScript. While the sample size for this questionnaire was small, four of the six questions showed a clear majority in answers.

The first question asked the participants if they thought that Parasitic JavaScript harmed the user experience. All the participants answered no to this question, which supports the results of the user study.

The second question asked participants if they would mind websites using Parasitic JavaScript. The possible answers participants could select were: I would not want to have websites use Parasitic JavaScript; It is OK if I know about it; and I have no problem. Seven of the eight participants said it would be OK if they knew about it.

The third question relates to the second, as participants are asked whether they would enable or disable Parasitic JavaScript if given the option. Five of the participants said they would disable Parasitic JavaScript, while three said they would enable it. The results of this question were inconclusive.

The fourth question queried participants if they would be more inclined to accept Parasitic JavaScript if it was performing a charitable task. Seven of the eight participants answered yes to this question, which may indicate that Parasitic JavaScript is more suited to charitable environments rather than commercial.

The fifth question asked participants if they believe it is important websites running Parasitic JavaScript make you aware of what task they are performing. All eight participants answered yes to this question, again this indicates that commercial environments where the task being performed is more likely to be kept discrete may not be suitable for Parasitic JavaScript.

The final question queried the participants views on a commercial application of Parasitic JavaScript. The question asked if it would be acceptable if YouTube or another video sharing website installed Parasitic JavaScript in exchange for allowing users to watch videos. Five of the participants indicated that it would be acceptable, while three thought it would not be. The results of this question were inconclusive.

Overall the questionnaire did answer several questions about the participants views on Parasitic JavaScript. Although to come to any conclusive conclusions a larger questionnaire would need to be performed. Table 6 lists a summary of the results.

Question	**Answers**	**Results**
For the websites which included Parasitic JavaScript, do you think the user experience was hindered or impaired?	Yes No	0 8
Would you mind having websites use Parasitic JavaScript?	I would not want to have websites use Parasitic JavaScript It is OK if I know about it I have no problem	0 7 1
If you visited a website which had Parasitic JavaScript and gave you the option to disable or enable it, what option would you pick?	Disable Enable	5 3
Parasitic JavaScript can be used to compute many things. Would you be more inclined to accept Parasitic JavaScript if it was performing a charitable task?	Yes No	7 1
If you visit a website running Parasitic JavaScript, do you think its important they let you know what they are calculating?	Yes No Maybe	8 0 0
Do you think it would be acceptable if YouTube.com (or another video website) used Parasitic JavaScript in exchange for letting users watch videos?	Yes No	5 3

Table 6: Summary of results collected in the questionnaire

10 Conclusions and Future Work

Parasitic JavaScript enables web browsers to be turned into nodes of a distributed computer without the knowledge of the user, and all that is required from the user is that they visit a website containing Parasitic JavaScript. The uses for this are both intriguing and scary. Intriguing in the sense that this could allow users to actively contribute to projects similar to SETI@home just by visiting a website, this would not require any installation on the users behalf and thus could potentially increase the available number of contributors.

Parasitic JavaScript is scary because it can be used for more than just ray tracing or solving n-queens, there is no reason it could not be used to brute-force encrypted or hashed passwords. JavaScript gives the flexibility to do many of the things any other programming language is able to do. The commercial applications are also interesting: while the price of computer time is not well defined, websites like YouTube.com that allow users to watch videos that last several minutes are ideal candidates for Parasitic JavaScript. They may be able to use Parasitic JavaScript as a form of exchange for watching videos. Considering the cost involved in running a service like YouTube.com it might be possible to off-load video conversion and compression tasks to users.

As JavaScript becomes more popular, the demand for new features will most likely increase, even today the HTML5 specification [44] includes JavaScript functionality to store arbitrary data into a local database. This could allow for complex calculations that require large amounts of data storage, but also for calculations that are persistent between web pages. This would reduce the requirement of having work units completed in a reasonably small amount of time and also improve the communication-computation ratio. The manufactures of web browsers are also putting significant amounts of effort into improving the performance of JavaScript, which ultimately means faster work unit processing for Parasitic JavaScript.

One aspect of Parasitic JavaScript which was not covered significantly in this report was network usage. Given the large range of Internet connections available, with many users still on 56k modems, functionality to reduce or increase the amount of data being sent between the client and server would be desirable. This in combination with dynamic performance adjustment would ensure that all sides of the user experience are covered.

Barabási et al. suggested in the original parasitic computing paper that the communication-computation ratio would need to improve for parasitic computing to become viable. Given that Parasitic JavaScript allows for a large amount of flexibility in comparison to the TCP SAT solver (Section 2.1.1), the communication-computation ratio has definitely been improved. Parasitic JavaScript also obtains the same obscurity that the TCP method has, as it is difficult to detect for the non-inquisitive user. Dynamic performance adjustment (Section 5.4) helps ensure the users computer is not overloaded, this improves on the TCP method which could result in a denial of service attack in cases where the server being used to solve the TCP SAT problem is not able to handle the TCP packets.

The ethical questions about Parasitic JavaScript still remain. While the questionnaire (Section 9.4.2) did give an insight into peoples' views of the idea, the sample size was not large enough to draw any concrete conclusions. There are arguments to both sides of the equation: on one side it can be argued that

CPU cycles are perishable, on the other side resources are being used without permission which is generally considered theft. Ultimately if users are given the choice, then Parasitic JavaScript is acceptable.

In conclusion this report introduces the concept of Parasitic JavaScript, a system that allows web browsers to contribute to a distributed computing project without harming the user experience. Unlike other public resource computing projects, Parasitic JavaScript does not require installation or any interaction from the user. Practical applications of Parasitic JavaScript may not be viable as yet, but as JavaScript performance improves and new features such as persistent local database storage in HTML5 are released, the potential certainly increases.

Acknowledgements

The author would like to thank his supervisor Bernhard Pfahringer for his support and all the participants of the user study.

References

[1] A.L. Barabási, H. Jeong, J.B. Brockman, and V.W. Freeh. Parasitic computing. *Nature*, 412(6850):894–897, 2001.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[3] Distributed.net. *URL: http://distributed.net.*

[4] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Reading, MA, 1994.

[5] A.L. Barabási, H. Jeong, J.B. Brockman, and V.W. Freeh. Parasitic Computing: Supplementary Material. 2001.

[6] GA Kohring. Implicit Simulations Using Messaging Protocols. *International Journal of Modern Physics C*, 14(02):203–213, 2003.

[7] M. Gardner. Mathematical Games: The Fantastic Combinations of John Conways New Solitaire Game Life. *Scientific American*, 223(4):120–123, 1970.

[8] D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. 2003.

[9] Skype Limited. Skype. *URL: http://www.skype.com.*

[10] S.A. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internel Telephony Protocol. *Arxiv preprint cs.NI/0412017*, 2004.

[11] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS06)*, pages 1–6.

[12] B. Cohen. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, 6, 2003.

[13] A.R. Bharambe, C. Herley, and V.N. Padmanabhan. Analyzing and Improving BitTorrent Performance. *Microsoft Research, Microsoft Corporation One Microsoft Way Redmond, WA*, 98052:2005–03.

[14] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. *Proceedings of International Conference on Peer-to-peer Computing*, 101, 2001.

[15] ACM Code of Ethics and Professional Conduct. *URL: http://www.acm.org/about/code-of-ethics.*

[16] R.N Barger and C.R. Crowell. The Ethics of "Parasitic Computing": Fair Use or Abuse of TCP/IP Over The Internet? *Information ethics*, 2005.

[17] M. Baker, G. Fox, and H. Yau. Cluster Computing Review. *Northeast Parallel Architectures Center, Syracuse University, Nov*, 1995.

[18] F. Berman, G. Fox, and A.J.G. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* Wiley, 2003.

[19] R. Pike. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[20] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programing in Erlang.* Prentice-Hal, 1996.

[21] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE COMPUTATIONAL SCIENCE & ENGINEERING*, pages 46–55, 1998.

[22] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[23] C. Germain and N. Playez. Result checking in global computing systems. *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 03), San Francisco, California*, pages 23–26, 2003.

[24] E. ECMAScript. ECMAScript Language Specification, 1999.

[25] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly Media, Inc., 2006.

[26] Adobe Flash Player. *URL: http://www.adobe.com/products/flashplayer/.*

[27] Java Applets. *URL: http://java.sun.com/applets/.*

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

[29] Bennett Daviss. Take a byte. *New Scientist*, August 2002.

[30] L.F.G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[31] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). *Request for Comments*, 4627, 2006.

[32] S. Souders. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, Inc., 2007.

[33] Apple. WebKit Timer Source Code. *URL: http://trac.webkit.org/browser/trunk/WebCore/platform/Timer.cpp*.

[34] Mozilla. Firefox Timer Source Code. *URL: http://mxr.mozilla.org/firefox/source/xpcom/threads/TimerThread.cpp*.

[35] A. Silberschatz. *Operating Systems Concepts with Java*. John Wiley & Sons, 7th edition, 2007.

[36] A.S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[37] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[38] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36(2):111–133, 1974.

[39] Kelly D. Crawford. Solving the n-queens problem using genetic algorithms. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 1039–1047, New York, NY, USA, 1992. ACM.

[40] NQueens@home. *URL: http://nqueens.ing.udec.cl/*.

[41] T.C. Belding. The distributed genetic algorithm revisited. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 114–121. Morgan Kaufmann, 1995.

[42] E. Alba and J.M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[43] Google. Google Chrome/V8 - Design Elements. *URL: http://code.google.com/apis/v8/design.html*.

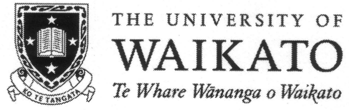
[44] W3C. HTML 5 Specification. *URL: http://www.w3.org/html/wg/html5/*.

A User Study

A.1 Ethical Consent

School of Computing & Mathematical Sciences
The University of Waikato
Private Bag 3105
Hamilton
New Zealand

Phone +64 7 838 4021
www.scms.waikato.ac.nz



11th August 2008

Nick Jenkin
C/- Department of Computer Science
University of Waikato

Dear Nick

Request for approval to perform several experiments involving human participants

I have considered your request for approval to perform several experiments involving human participants during August this year, to contribute to your COMP520 project.

The purpose of this experiment is to determine the effectiveness of the optimisation methods used to ensure the user experience is not harmed. In addition to the primary experiment, a questionnaire will be used to identify views on the ethics of this project.

The procedure described in your request is acceptable. I note your statements that confidentiality and participant anonymity will be strictly maintained, all information gathered will be used for statistical analysis only and no names or other identifying characteristics will be stated in the final or any other reports. Data will be kept secure and stored in the Usability Laboratory archive under the control of the Usability Laboratory Manager.

The research participants' Bill of Rights and the Research Consent form comply with the requirements of the University's human research ethics policies and procedures.

I therefore approve your application to undertake the experiment.

Yours faithfully

Mike Mayo
Department of Computer Science
Human Research Ethics Committee
School of Computing and Mathematical Sciences

A.2 Questionnaire

Parasitic

For the websites which included Parasitic JavaScript, do you think the user experience was hindered or impaired?

◯ Yes ◯ No

Would you mind having websites use Parasitic JavaScript?

◯ I would not want to have websites use Parasitic JavaScript
◯ It is OK if I know about it.
◯ I have no problem.

If you visited a website which had Parasitic JavaScript and gave you the option to disable or enable it, what option would you pick?

◯ Disable
◯ Enable

Parasitic JavaScript can be used to compute many things. Would you be more inclined to accept Parasitic JavaScript if it was performing a charitable task?

◯ Yes ◯ No

If you visit a website running Parasitic JavaScript, do you think its important they let you know what they are calculating?

◯ Yes ◯ No ◯ Maybe

Do you think it would be acceptable if YouTube.com (or another video website) used Parasitic JavaScript in exchange for letting users watch videos?

◯ Yes ◯ No