Performance Improvements in Crawling Modern Web Applications

by

Alireza Zarei

B.Sc., Amirkabir University of Technology (Tehran Polytechnic), 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

February 2014

Abstract

Today, a considerable portion of our society relies on Web applications to perform numerous tasks in everyday life; for example, transferring money over wire or purchasing flight tickets. To ascertain such pervasive Web applications perform robustly, various tools are introduced in the software engineering research community and the industry. Web application crawlers are an instance of such tools used in testing and analysis of Web applications. Software testing, and in particular testing Web applications, play an imperative role in ensuring the quality and reliability of software systems. In this thesis, we aim at optimizing the crawling of modern Web applications in terms of memory and time performances.

Modern Web applications are event driven and have dynamic states in contrast to classic Web applications. Aiming at improving the crawling process of modern Web applications, we focus on state transition management and scalability of the crawling process. To improve the time performance of the state transition management mechanism, we propose three alternative techniques revised incrementally. In addition, aiming at increasing the state coverage, i.e. increasing the number of states crawled in a Web application, we propose an alternative solution, reducing the memory consumption, for storage and retrieval of dynamic states in Web applications. Moreover, a memory analysis is performed by using memory profiling tools to investigate the areas of memory performance optimization.

The enhancements proposed are able to improve the time performance of the state transition management by 253.34%. That is, the time consumption of the default state transition management is 3.53 times the proposed solution time, which in turn means time consumption is reduced by 71.69%. Moreover, the scalability of the crawling process is improved by 88.16%. That is, the proposed solution covers a considerably greater number of states in crawling Web applications. Finally, we identified the bottlenecks of scalability so as to be addressed in future work.

Preface

This thesis is original, independent work and it is designed, carried out, and analyzed by the author, Alireza Zarei.

Table of Contents

Appendix

List of Tables

List of Figures

Acknowledgements

I wish to thank my supervisor, Dr. Ali Mesbah, for his careful supervision, guidance and support on carrying out the projects constituting this thesis. I would like to express my gratitude and appreciation for bringing about internship opportunities for me during my M.A.Sc. program. Without doubt, this work was not possible without his generous support, help and accurate attention.

I would like to thank Peter Luong, president and founder of FusionPipe Software Solutions, for creating internship opportunities which provided me experience in preparing for industry and support in completing my program. I would also like to thank him for his kind supervision and generous attitude in sharing his invaluable expertise with me during the projects. Especial thanks to Sang Mah, whose enthusiasm, extraordinary positive attitude, and tireless efforts in MITACS, bring about numerous fruitful industry experiences for students such as me at UBC.

Moreover, I would like to thank my committee members, Dr. Karthik Pattabiraman and Dr. Sathish Gopalakrishnan for kindly agreeing to participate in the committee for evaluating my thesis. I would like to thank them for their utmost generosity with their time, great improvement ideas, and constructive feedback for revising my thesis.

I would also like to thank Dr. Pooyan Fazli for providing me with his effective mentorship and valuable time during the course of the program.

Last but not least, I would like to thank our friendly community in Software Analysis and Testing (SALT) lab who helped me greatly during the course of this research with their encouragement, support and feedback.

Dedication

I would like to dedicate this work to my family; my parents whose love and support have always filled my heart with strength and hope. I wish to dedicate this work to them for giving me the opportunity to follow my heart and pursue my dreams. I would also like to dedicate this thesis to my dearest sisters, Mina and Maryam, who are the meaning of hope for me.

I would also like to dedicate this thesis to my friends whose encouragement, support and understanding have always empowered me and kept me going during the most difficult times.

Chapter 1

Introduction

Nowadays, Web applications play an important role in performing various day-to-day tasks in businesses, industry, and individuals' daily lives. Performing tasks such as on-line banking or booking flight tickets has become possible by prevalence of Web applications. Pervasiveness of Web applications in the carrying out tasks in modern societies calls for ensured reliability and enhanced quality. To that aim, various tools have been introduced to help Web developers and quality assurance professionals deliver more robust Web applications. Web application crawlers are among the most important tools in the area of Web application development and search engines. Web crawlers are used in testing Web applications as well as in realizing the indexing of the World Wide Web (WWW) by search engines.

Traditionally, Web applications were composed of a number of self-reliant documents identified by unique Uniform Resource Locators (URLs) [3]. These documents were linked to each other by means of inter-document references called hypertext links or hyper-links. As such, in order to navigate through a Web application in a Web browser, each time a hyper-link was clicked a new document was retrieved, loaded in the browser window, and the new document would replace the previous content of the browser window. This design of Web applications resulted in inefficient time and bandwidth consumption because even for a minor change in needed in the document loaded in the browser, the whole document needed to be transmitted and replaced.

As a significant progress towards the maturity of Web applications, in addition to the hyper-link based URL transition mechanism, "modern" Web applications utilize "Ajax" technologies to provide "interactivity" and "responsiveness" [4] for users. Ajax technologies including scripting languages such as JavaScrip [5] and asynchronous method calls help achieve dynamic Web Applications. Dynamic modern Web applications boost the interactivity of Web applications by minimizing the time users need to wait for the application to be responsive again after each interaction such as clicking on a photo. The advent of modern Web applications greatly improved users experience, but on the otherhand, Web crawlers have to pay a price because

crawling event driven modern Web applications is inherently more resource demanding.

Web application crawlers automate the crawling of Web applications by exploring hyper-link-based transitions and other types of transitions such as Ajax-based ones. These multi-layer tools are used for various purposes. They can be used for indexing Web applications in search engines. They are also used as a means for testing Web applications. Crawljax [4], a modern Web crawler to be outlined in this thesis, is able to explore JavaScript-enabled Web applications.

Crawling modern Web applications is inherently a resource (e.g. time, compute, memory) demanding task for various reasons. First, as apposed to crawling classic static HTML Web pages, modern crawlers need to execute JavaScripts and apply the dynamic changes to the state of the Web application being crawled. Whereas in crawling traditional HTML-based Web applications, a transition from one state to another state in the crawling process is more of a retrieval and parsing process rather than an execution one. The event driven model of modern Web applications, the execution of scripts, and navigating dynamic transitions are more time consuming than merely retrieving the pages and following static URLs. The reason is that a modern Web crawler needs to wait for a browser (or a tool that can execute JavaScript) to load the content and execute the JavaScript.

In addition, modern Web application crawlers are composed of various layers and technologies working together to make the crawling process feasible. Specially the additional components for executing JavaScript and managing the states navigations impose considerable resource requirements on the crawling process. For example, in crawling modern Web applications the crawler needs to communicate with the execution layer (e.g a browser). These inter-process communications cause a sizable time consumption on the crawling process.

In this work, we aim at improving the memory and time performances of the crawling process in modern Web applications. We investigate two components of the crawling process to improve the time and memory efficiency of the crawling. First we focus on the changes that occur in the Web applications and induce state transitions. By doing so, we aim to improve the time efficiency of the state transition mechanism of the crawling process. In addition, we investigate the memory utilization of the crawling process to improve the state coverage of the crawling. In other words, we aim at increasing the number of states crawled given a limited amount of resources (memory).

In the optimization of states transition management part, we propose

three different solutions, improved incrementally over one another, to reduce the time consumption of tracking changes in the crawling process. We employ different methods and techniques to achieve improvement in time performance.

In order to increase the number of states crawled given a limited amount of memory, we introduce a new component to the crawler to free the memory required for saving the states of the Web application in the crawling process. To this aim, we replace the current storage and retrieval component which relies on the main memory by our proposed solution to crawl a greater number of states.

The first alternative solution we proposed, utilizing a proxy, did not overtake the default state transition management. However, the second enhancement proposed, without a proxy, was able to improve the time performance of the state transition management by 253.34%. That is, the time consumption of the default state transition management is 3.53 time the proposed solution time. The third alternative solution also overtook the default mechanism by 197.48%. The second solution demonstrated the best performance among all available mechanisms.

Moreover, the scalability of the crawling process is improved by 88.16%. That is, the proposed solution covers a considerably greater number of states in crawling Web applications. This scalability improvement was achieved costing 18.20% of time overhead. Finally, we identified the bottlenecks of scalability so as to be addressed in future work.

The rest of this work is organized as follows. In chapter two we present the background information related to the concepts and technologies used in this work. In chapter three, we provide an overview of Crawljax, the crawler on which our ideas are examined. This overview is based on the reverse engineering endeavor we did to shed lights on the rapidly changing characteristics of the crawler. Chapter four covers the motivation and the research goals. What follow afterward are the two main chapters covering the improvements we targeted in the crawling process; the time performance of the state transition in chapter five and increasing the number of states crawled in chapter six. Afterward we revisit the research questions, address some of the threats to validity and the future work in the discussion in chapter seven. Finally a review of the related work finalizes the thesis.

Chapter 2

Background

In this chapter we briefly touch upon the concepts and information that constitute the foundations on which this thesis is based. We start off by shedding some lights on Web applications concepts because we will discuss Web applications as objects of the experiments throughout the thesis. Web applications are directly used in designing the experiments. They are also pivotal in explaining many parts of the thesis such as the crawling process in Web crawlers and the memory analysis we peformed. In addition we briefly explain the crawling of both traditional and modern Web applications. Afterward we proceed to pinpoint the aspects of the crawling process that we aim to improve.

2.1 Web Applications

A Web application is any piece of software that is built to be run in a common Web browser such as Mozilla Firefox®. In other words, a Web application is a special type of client-server application that uses a Web browser as its client. Web applications usually consist of two components (sides), a client side and a server side. In this thesis we focus only on the client side. The area of Web application development is diversely rich and there are manifold technologies, languages and standards used in developing Web applications. We shortly give an overview of some related technologies in the following.

Web applications constitute a considerable portion of deployed software in today's computing and business ecosystems. Web-mail services, news broadcasting Web sites and on-line banking portals are examples of Web applications used by users on a daily basis. Web applications vary greatly in complexity and size, from only a few lines of static HTML documents to hundreds of thousands of lines of code in a dynamic multi-tier application.

Technologies, languages, approaches and techniques used in creating Web applications are heterogeneously diverse. Just to name a few, HyperText Murk-up Language (HTML), eXtensible Markup Language (XML) and Cascading Style Sheets (CSS) are used for creating the content and

defining the presentation of the content in the client side user interface of Web applications. Scripting languages such as JavaScrip [5] (EcmaSript [6]) empower the client side of Web applications with dynamic execution. They also provide control over content modifications and help reduce the amount of data transferred between the browser and the server by enabling dynamic updates to the user interface. On the server side, PHP, ASP.NET, and Java technologies are popular for building the engines of Web applications in the server side. Languages and technologies used in Web development are too numerous to be mentioned here so in the rest of this chapter we merely suffice to explain the concepts that are directly of pertinence to our work.

2.1.1 HTML and DOM

The information openly published on the Internet is most effective when it can serve a vast audience. In order to maximize the reach of the content published on the Web, in early days of the advent of the Internet, a universal language that all computers could agree on was a desideratum. To that aim, HTML, developed by Tim Berners-Lee and popularized in 1990s, became the "mother tongue" of the World Wide Web (Web). HTML is used in developing Web applications to create content, retrieve documents by hypertext links, add forms for accessing services, and including multimedia in a page [7]. HTML role in Web applications is interrelated with its live transformation which is called HTML DOM.

As it is set out by World Wide Web Consortium [8] (W3C) in the specification of Document Object Model (DOM), the DOM is a language- and platform-independent convention that aims at unifying the usage of the documents in computer programs. Using the DOM interface, programs and scripts achieve a unified way of accessing and modifying documents. Numerous types of documents including XML, and XHTML documents, which are incremental improvements over HTML, are used in the area of Web applications.

When an HTML document is retrieved from a Web server, the browser parses the page and builds an object representing the page based on the DOM specification. This object is called DOM object or DOM tree. The scrips of the Web application (e.g. JavaScript) can access and modify the DOM tree. These accesses and manipulations can retrieve and modify the presentation, content and the structure of the DOM tree. The DOM is also used by the layout engines of browsers. Layout engines such as WebKit [9] use the DOM interface to render the presentation of Web pages.

2.1.2 Modern vs. Classic Web Applications

The event driven model of modern Web applications utilizes "Ajax" technologies to reduce the responsiveness gap between classic Web applications and desktop applications. Ajax is a collective short form for the combination of Asynchronous JavaScript and XML. Modern Web applications use different Ajax technologies. For example, XML and HTTP requests are used for communicating data, HTML DOM for live presentation of user interfaces, and CSS for a presentation layer over the content. In addition, JavaScript is used for DOM manipulation and making links between all the aforementioned technologies [1].

As shown in figure 2.1 from [1], the model of modern Web applications differs from that of classic Web applications in terms of data exchange method and user interactions. In classic Web applications, the application is a set of separate "whole" HTML pages with logical links. Theses pages are connected together by hypertexts links. In classic Web applications, the browser provokes an HTTP request to the server of the application once users click on a link or submit a form. The server then analyses the request and creates the response as a complete new HTML page [1].

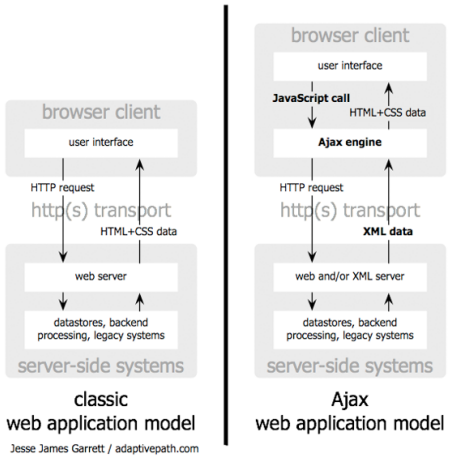

Figure 2.1: Ajax vs. classic Web applications models. Source: [1].

On the other hand, in modern Web applications, an Ajax engine enhances the user experience by eliminating the need for reloading a whole new page each time data is exchanged with the server. That is, users actions on a modern Web application result in sending asynchronous requests

to the sever of the application. In the mean time, the application is responding and not waiting on a hold for the response while the server processes the request and responds the data back. The key difference here is that instead of reloading the whole application by a new HTML page, the Ajax engine processes the response data and applies partial changes to the DOM of the application. Hence, the data exchange is minimal and the user does not need to wait for the response and a complete reload of the application [1].

As modern Web applications are richer and have a greater number of components than classic Web applications, the crawling method for each category is different. In the following we give an overview of the crawling of each of these two types of Web applications and illustrate how they are different.

2.2 Crawling Web Applications

Crawling Web applications is the process of exploring the structure and the content of Web applications. In classic Web applications, exploring the applications is carried out by following hypertext links. On the other hand, in modern Web applications, elements such as div and span can also be sources of transition form one page of the application to another page. Hence, the crawling process is to some extent more resource intensive in modern Web applications. We cover the differences in crawling the two classes of Web applications in what follows.

2.2.1 Crawling Classic Web Applications

Hypertext links are pivotal in crawling classic Web applications. A classic Web crawler, comparative to the way general purpose search engines such as Google® crawl, starts form a provided URL, retrieves the page, pareses the page and extract all absolute and relative URLs and adds them to the queue designated for URLs to be crawled next. This process is repeated recursively till the queue is emptied or certain conditions are met.

2.2.2 Crawling Modern Web Applications

On the other hand, in modern Web applications the dynamic state changes are taken into account. In crawling modern Web applications, the presence of an Ajax engine in the browser is pivotal. This makes crawling modern Web applications different in a number of ways. First, there are elements other than hypertext links (e.g. div) that can transition the state of the

application. This transitions is done to discover new states and information. In addition, the dynamic nature of JavaScripts requires the crawling process to wait for the execution of JavaScripts and to allow the DOM tree to be settled after firing events.

Because JavaScript based reactions (function invocations) can be attached to almost any element in a Web page, modern Web application crawlers, in addition to parsing each page and extracting its hypertext links, need to examine other types of elements for exploring Web applications. Crawljax [4], a modern Web application crawler, for example, can be configured to examine a set of HTML elements for exploring Web applications.

As a result of the differences mentioned above, the crawling process of modern Web applications is more detailed than that of classic applications. The crawling algorithm for a modern Web crawler such as Crawljax starts with opening a URL, extracting all hypertext links and elements such as div, button, and span and saving them in a candidates' queue. Afterward these elements are polled one by one and an event (such as click or hover) is fired on each of them. As non-href (non hyperlink) elements do not guarantee a change in the URL, a sufficient amount of time is required for allowing the dynamic changes and server communications to finish and to let the DOM settle. Then the crawler checks if more information is discovered. If so, the newly discovered parts of the application is parsed for candidate elements and any found element is added to the queue. This process continues recursively.

2.2.3 Problem and Proposed Solutions

As it was explained earlier, compared to crawling classic Web applications, crawling modern Web applications requires carrying out a greater number of steps. In particular, requirements such as examining a much wider set of HTML elements (e.g. span, href, and div), executing JavaScript code, and communicating to the server and the browser at each state make the crawling of modern Web applications more resource consuming.

Crawling modern Web applications is resource intensive. Consuming resources such as time and memory is of a considerable degree in modern crawlers. The reason is that the crawling process needs to employ additional components for performing the sub-tasks of modern crawling that does not exist in classic crawling.

There are a number of steps which are pertinent only to modern crawling. First, handling the execution of JavaScript has to be done in one way or another such as employing a browser. In addition, tracking dynamic state

changes must be handled as any interaction can lead to a new state in the application. Furthermore, managing the interactive communications between the client and the server in each dynamic state is essentially more time consuming than merely retrieving the data.

In order to mitigate the aforementioned resource demanding qualities of crawling modern Web applications, in this thesis we aim at identifying and tacking some of the performance problems of the crawling process. We specifically focus on improving the time and memory performance of the crawling process. Regarding the time performance improvement, we focus on state transition management of the crawling process. In addition, we investigate the state storage and retrieval aspects of the crawling process to improve the memory performance of the crawling process. The discussion is followed by expanding upon the performance improvements that will be discussed later on in this thesis.

2.3 Performance Improvements

Improving the performance of a system is defined as tuning the way resources are consumed by the system in achieving goals . As mentioned earlier, crawling modern Web applications imposes extra resource requirements on the underlying system the crawler is running on. Time, memory, and compute resources are examples of system resources that are consumed more intensively by a modern Web application crawler compared to classic crawlers. Hence, here we aim at improving the time and memory performance of the crawling process by tuning the way specific aspects of the crawling are carried out.

2.3.1 Time Performance Improvement

The process of crawling modern Web applications is more time consuming than the classic one because modern crawling employs more layers of tools working together to achieve the task of crawling. The time needed for performing inter-process communications, performing rounds of requests and responses between the client and the server, and waiting for the user interface to settle after interactions are among the time consuming tasks performed by the crawling process. In addition, the way specific sub-tasks such as managing the state transitions are performed in the crawling process creates optimization opportunities. In this work, we focus on the state transition management of the crawling process to optimize for reducing time consumption.

2.3.2 Memory Performance Improvement

Memory is one of the system resources that are extensively used in the crawling process. Loading layers of different tools and components that are necessary for performing the crawling is an important inducer of memory consumption. In addition, parsing HTML pages to extract HTML elements requires significant memory allocation. Another critical part of the crawling process is handling the storage, comparison, retrieval and identification of the states. In this work, we focus on improving the memory performance of this part of the crawling process to improve the scalability of the crawling process.

We utilize a graph database to overcome the obstacles hindering the scalability of the crawling process. Graph databases are shortly introduced in the next section to facilitate the presentation of the application of a graph databases to the Crawler.

2.4 Graph Databases

Graph databases are a sub-category of NoSQL databases. NoSQL databases which are also called "Not only SQL" databases are a category of databases that provide additional ways other than SQL for storing and retrieving data. A graph database utilizes graph data structure for storing and accessing data. The key is that each item in a graph database provides direct "look-up free" references to its adjacent elements. In other words, a graph database provides index-free adjacency.

Graph databases are optimized for utilization in software systems that operates on a data that is naturally presentable by graphs. The reason is that graph databases avail of graph structures, graph nodes, and graph edges for designing the data model and storing the data. Aside from the ease of modeling graph-like data, graph databases are geared to provide efficient mechanisms for performing graph-specific operations such as finding the shortest path between two nodes in a graph.

Graph databases vary in terms of capabilities, features and the scope for which they are designed. Fore example, some graph databases provide transaction handling and concurrent access while others do not. There are general purpose graph databases and some more specialized ones such as triple stores. Graph databases are widely used in industry specially by social networking Web applications. For example, FlockDB [10] is used for supporting the storage and retrieval of the underlying system of Twitter® [11].

We implement the aforementioned improvements as enhancements to Crawljax and perform empirical experiments to evaluate the suggested enhancements. Hence in the next chapter Crawljax will be discussed in details to explain its default algorithm and the suggested enhancements we proposed for the crawling process.

Chapter 3

Reverse Engineering Crawljax for Optimization

Crawljax [4] is a software tool for crawling modern Web applications. It specifically is delivered as a solution for answering the challenges of crawling Ajax-based Web applications. The challenges arise from some characteristics of modern Web applications. These characteristics include execution of scripts on the client side, asynchronous communications between the client and the server, and dynamic manipulation of DOM trees. These characteristics result in dynamic state changes in Ajax-based Web applications without a necessary change in the URL of the Web application. Crawljax captures these state changes and the transitions between states in a finite-state machine model. The structure underlying this finite-state machine is called stateflow graph by its designers and here after we refer to it as such. The stateflow graph, central to the memory enhancements suggested in this thesis, is a directed multigraph[1] with states at nodes and transitions at edges.

Crawljax utilizes selenium Web driver [12] to bring up browsers and crawl Web applications. Running Web applications in a real browser, Crawljax ensures the model it creates matches the real behavior of the Web application under crawl. The state machine created incrementally by Crawljax comprises the state-space of the Web application under crawl. The state machine also includes the navigational path between states. This state machine can be used for various purposes such as quality assurance and program comprehension.

In this thesis we propose some enhancements to the Crawljax in order to improve the state coverage and time performance of the crawling process. We explain the internal structure of Crawljax and the parts of its crawling algorithm that are directly related to this work to achieve more clarity in presenting the work performed in this thesis.

[1] A multigraph is a non-simple directed graph in which loops and multiple edges between vertices are permitted.

3.1 Crawljax High Level Algorithm

Form a high level perspective, Crawljax works as the following. A crawl session starts by providing the URL of the Web application and additional optional configurations as inputs to the crawler. A Web browser is brought up and it is driven to open the first page of the Website referenced by the URL. Afterward The loaded page is parsed and all of the candidate elements in the page such as buttons and URLs are identified to be clicked on in the next steps. Clicking on elements can lead to transitions to new states. Hence, after clicking on each candidate element, the current state of the DOM is compared to the previous state of the DOM. If the state is changed and it is not a replica it is added to the inferred state machine.

```
   input  : URL, Configurations
   output: stateFlowGraph, Logs
 1 Open the URL in the browser;
 2 Parse the loaded DOM and extract all the candidate elements;
 3 Save the candidate elements in candidateElements ;
 4 while termination conditions not met do
 5     Pick an element from the candidateElements;
 6     Fire a click event on the element;
 7     Extract the state of the Web application form the browser;
 8     Compare the new state with last captured state;
 9     if the new state is different from the previous state then
10         Compare the new state to all previous states;
11         if no replica of the state found in the stateFlowGraph then
12             Add the new state to the the stateFlowGraph;
13             Extract the candidate elements form the new state and
               store them in candidateElements;
14         Add the transition to the the stateFlowGraph;
```

Figure 3.1: Crawljax high level algorithm

3.2 Reverse Engineering Crawljax

As Crawljax is being evolved rapidly, relying on the initial papers does not suffice for a deep understanding of the "current" state of the system. So

we conducted a thorough source code investigation to be able to propose alternative solutions for optimizing the crawling process.

In this thesis, we investigate if we can utilize a graph database for storage and retrieval of data in Crawljax. As the main memory is usually smaller than secondary memories in computer systems our aim is to optimize the memory consumption by storing the data in a graph database instead of main memory. Based on our previous experiments with crawling Web applications with Crawljax, we knew that the states of the Web applications amount for a sizable memory consumption. Hence we came up with the idea of storing the stateflow graph of the Crawljax in a graph database. There were numerous graph databases options. In order to identify if there is a right choice of database for our application, we analyzed Crawljax implementation to see what criteria and requirements emerge that guide us in selecting the most amenable graph database.

3.2.1 Investigating Crawljax for Alternative Solutions

We investigated the source code of Crawljax to extract the criteria for selecting a graph database most suited to our application. Our findings from this investigation of Crawljax's internal components are as follows.

Crawljax opens a Web browser and goes to the URL of the application. After loading the index page, it examines the DOM tree of the page and extracts all the elements which are likely to change the state in case an event is fired on them. Then Crawljax fires events on these candidate elements and analyzes the changes. Finally, based on the analysis of the changes made to the DOM Crawljax incrementally builds a stateflow graph which models the states deduced by crawling the application. In order to save this model which represents the states and the transitions between them as a directed multigraph is utilized. This model is defined as follows.

The stateflow graph inferred from crawling a Web application A is denoted as a four-tuple (i,V,E,L). The meaning of these symbols are [4]:

- i: i is the initial state of the Web application. This state is captured when the page is finished loading into the browser and the *onload* event is fired by the browser.
- V: V is the set of all states in the application. Each dynamic DOM state is represented by a vertex v which is a member of the set V.
- E: E is the set of all edges in the stateflow graph. The members of E are ordered pairs of (v_1,v_2) which represent an edge from the vertex v_1

to vertex v_2. These directed edges exist in the graph when state v_2 is reachable from state v_1 by firing some event on on a clickable (a Web element that can be clicked on) c in state v_1.

- L: L is a function from E to the set of event types and DOM element properties.

This provides us with a solid mathematical understanding of the data structure that we need to provide in our alternative solutions.

There are also other important points which must be taken into consideration. If a change is detected in the state of the Web application the newly detected state is compared to all states in the state-machine. If a duplicate state is found the new state will not be added to the stateflow graph. However, regardless of the newness of the found state, a new edge is added to the stateflow graph, starting from the previous state to the current state. Edges hold additional information such as the type of the event that was fired and the element which the event was fired on. In addition the current state of the state-machine is updated to the newly resulted state. This means the crawling is continued by exploring the new state and hence a depth first traversal.

Crawljax has an option to crawl an application with multiple browsers. In the concurrent crawling sessions, all crawling nodes share the same stateflow graph but each crawler node has its own state machine. The state machine is an abstract interface on top of the stateflow graph. Hence the stateflow graph is a more concrete data structure in Crawljax. The state machines in combination with the stateflow graph insure the synchronized reading and updating of the states and navigational paths.

In other words, the state machine is divided into a global stateflow graph for all crawling nodes and multiple state machine instances, one for each crawling node. The crawling nodes are synchronized over the clickable element they are examining. This measure is taken to prevent exploring the same clickable by multiple crawling nodes.

Among numerous classes and data structures in Crawljax, there are two data structures that are of critical importance when considering the consequences of concurrent modifications. In particular, these data structures, encapsulate the logic of the steps that we need to implement for safe concurrent storing and retrieving of the data form and into the graph database.

There is a data structure that stores all the states with candidate elements and is called $statesWithCandidates$. The second data structure, which is called *cache*, is a map from states to queues of actual candidate

elements. Each queue stores a series of candidate elements which are extracted from the Web page. These elements are extracted when the browser has loaded this specific state mapped to the queue. Crawljax begins crawling by loading the index state and then running a crawling node called $CrawlTAskConsumer$ with which the first state, the index state, is crawled. The candidate elements of the first state are put into the *cache* as a starting point for the rest of the crawling.

A more detailed algorithm but still very high level of the multi-node crawling process of Crawljax is illustrated in algorithm 3.2.

```
input : URL, Configurations
output: stateFlowGraph, Logs
1  Open the URL in the browser and load the index page;
2  Parse the index and extract all the candidate elements;
3  currentState ← index;
4  stateWithCandidates ← {index};
5  cache ← {(index, CE)} such that CE = {e: e is a candidate element
   in the index};
6  stateFlowGraph ← ∅;
7  while stateWithCandidates ≠ ∅ do
8      Poll a state s from stateWithCandidates ;
9      Reset the browser;
10     Move the browser from index to s;
11     Retrieve s.candidateElements from cache ;
12     while s.candidateElements ≠ ∅ do
13         Poll an element e of s from s.candidateElements;
14         Fire a click event on e;
15         detectedState ← current state of the application in the
           browser;
16         if detectedState ≠ s then
17             Add e to crawlPath;
18             Compare detectedState with all other states in
               stateFlowGraph;
19             if detectedState ∉ stateFlowGraph then
20                 stateFlowGraph ← stateFlowGraph +
                   {detectedState};
21                 Parse detectedState for candidate elements;
22                 stateWithCandidates ← stateWithCandidates +
                   {detectedState};
23                 cache ← cache + {(detectedState, CanElements)}
                   such that CanElements = {e: e is a candidate element
                   in detectedState };
24             else
25                 crawlPaths ← crawlPaths + {crawlpath};
26             stateFlowGraph ← stateFlowGraph + {e};
27             currentState ← detectedState;
```

Figure 3.2: Crawljax high level algorithm with more details

Chapter 4

Motivation and Research Goals

In order to realize a system for crawling Ajax-enabled Web applications, numerous layers of software tools and technologies such as JavaScript execution engines and DOM processing libraries must be put in place to obtain a purposeful crawling outcome. Ajax-enabled Web applications are crawled for different purposes including, "program comprehension" and "analysis and testing of dynamic Web states". For example, the model Crawljax infers can be used for "generating a static version of the application" [4].

Because of its inherent complexity, a purposeful crawling process, capable of producing a pragmatic model for Web applications, introduces manifold areas of optimization. These areas of optimization, once considered carefully, have the potential to examine novel ideas for improving the crawling of Web applications.

In the following we explain the goals of the thesis by enumerating some challenges associated with crawling Web applications and propose our ideas for attacking these problems. We put forward two high level research questions to pinpoint the bird's-eye view of the goals of the thesis. Afterward, we elaborate on these questions by taking them to lower levels of abstraction to target more specific questions.

4.1 Motivation

Crawling Ajax-enabled Web applications is a predominantly resource consuming task for the computer system hosting the crawler. This high degree of resource consumption lies, in large part, on two sets of drivers. First, Ajax crawlers perform various sub-tasks by utilizing a multi-layer stack of tools. The second reason is that Web applications, which are the objects of the crawling, can be of considerable size in terms of the memory it requires to store their various components such as their DOM or their GUI.

In crawling Web applications, tasks such as retrieving a Web applica-

tion from its server, executing the application's scripts (chiefly JavaScript), communicating with a browser, tracking states and state transitions, and processing the content of the DOM impose a considerable memory and time consumption on the computer system hosting the crawler.

In addition, providing a mechanism for handling dynamic state transitions in Web applications imposes a high level of resource consumption on crawling. Especially, when applications have many states and the states of the Web applications are of a large size memory consumption becomes more challenging. Table 4.1 presents the data we collected about the size of the DOM trees during our experiments. The data shows that most of the Web applications in the top list of Alexa have states that on average are of a considerable size. The average length of states across all applications listed in table 4.1 is around Kilo characters. Moreover, Crawljax stores a stripped version of the DOM beside the original DOM. This stripped version has less details but is of almost the same size. In addition, the size of Java "char" type is two bytes. Hence, the size of the states in memory is on average one 928 Kilobytes (multiplying the average length of the DOM by four).

Table 4.1: Average DOM size in characters

Web Application	DOM Length
google.com	311671.49
wikipedia.org	242418.29
live.com	46303.59
twitter.com	192321.88
qq.com	603644.86
amazon.com	290661.37
linkedin.com	28188.43
baidu.com	184074.15
facebook.com	41485.17
youtube.com	310531.81
yhoo.com	304536.09
average	232348.83

In order to improve the crawling of Ajax-enabled Web applications, aiming at mitigating the memory and time consumption of the task of Web application crawling, we propose two enhancements to the crawling process. In the following, we state our enhancements targeting: 1) the improvement of

the time performance of the state transition management and 2) the memory performance of the state storage and retrieval component.

4.2 Research Questions

In this thesis, we initially defined two research questions to pinpoint the objectives of the thesis. These research questions address enhancing the performance of the crawler in crawling Ajax-enabled Web applications. In particular, we aim at conducting research to assess our ideas for improving the time consumption and memory utilization of the crawler in performing the crawling task. Our research questions address the following aspects of the crawling process:

Idea 1: Improving the time consumption of crawling by revising the process of examining state transitions in Ajax-enabled Web Applications

Idea 2: Optimizing the memory utilization for improving the crawling process memory consumption

The research questions are stated in the following and will be further broken into more specific and more detailed sub-questions. Moreover, as we made progresses in the thesis project and performed the experiments in details, analysis of the results revealed interesting facts about the time requirements and memory utilization of our proposed methods. These discoveries guided us to design further research questions to assess additional enhancements and explore further corners of the crawling process.

4.2.1 Improving Time Performance

As mentioned earlier, there are numerous aspects and sub-tasks in crawling Web applications that have the potential to be investigated for finding optimization opportunities. One important aspect of Ajax-enabled Web applications is that they can change their states dynamically, namely they are dynamically stateful. More specifically, Ajax-enabled Web applications can transition into different dynamic states as the Ajax technologies (such as asynchronous calls and script execution) change the state of the Web application upon occurrence of different events. Hence managing the changes in the states and transitions between the states are of high importance in the crawling process. As such, if we aim at modeling the states of the Web applications during the crawling process we have to provide some mechanism for keeping track of state changes in the Ajax-enabled Web applications.

Keeping track of state changes and transitions between the states imposes a significant complexity on the crawler and hence creates potentials for optimization. One of the important steps in tracking the state changes is to determine if, at specific points in time, the state of the Web application has changed or not. Another important issue is to check whether the state to which we have just transitioned is a replica (i.e. a clone of one of the previously stored states) or is an unprecedented state. As such, we think the state comparison step has the potential to be attacked for enhancing the crawling time.

Here we state our first idea for improving the crawling of Ajax-enabled Web applications as a high level research question which later on will be broken into more specific sub-questions:

RQ1: How much can the time consumption of the state transition management be improved by enhancing the state comparison mechanism?

The intuition behind this research question is that we know, from experience, that comparing the states of the Ajax-enabled Web applications is an expensive task. State comparison is time demanding because it includes several sub-tasks dealing with multiple layers of software tools. For example, accessing the states requires communicating with a browser. In addition the states are, on average, large objects. This translates to an expensive time requirement for the extraction of the state data from the browser, initiation of the state objects on the crawler and performing the comparison.

4.2.2 Increasing State Coverage

Memory utilization is another aspect of the crawling of Ajax-enabled Web applications that is of high implication to the performance of the crawler. By experience, from crawling Web applications with Crawljax in previous studies we know that the states of Web applications are usually of considerable size. States of Ajax-enabled Web applications are stored in the memory while a Web application is explored by the crawler. The large size of the states multiplied by the number of states crawled, to any given point in crawling, accounts for a sizable amount of memory consumption. We believe that this imposed consumption of main memory is one of the obstacles for increasing the coverage and comprehensiveness of crawling. As such, we believe that improving the memory consumption is a potential optimization area for achieving more coverage in crawling Ajax-enabled Web applications.

We state our second idea for enhancing the crawling of Ajax-enabled Web applications as a research question which targets the memory utilization of

the crawling process:

RQ2: To what degree can the coverage and comprehensiveness of crawling be improved by relinquishing state storage to a graph database?

What we endeavor to achieve here is we would like to assess the results of enhancing the main memory consumption on the state coverage achieved in crawling. Especially, we are interested in improving the way the states data are currently stored in the crawling process. We do this by analyzing the effects of enhancements made to the state storage mechanism on the memory performance of the crawler and on the number of states it can crawl.

4.3 Goals

The goal of this thesis is to improve the performance of crawling of Web applications during which we assess our ideas for enhancing the crawling process. In particular, we target improving time performance and memory performance of the crawling. In order to improve the time performance of the crawling, we aim at reducing the time the crawler spends on tracking state changes and the transitions between states. In addition, in order to improve the memory performance of crawling, we target the reduction of the per-state memory consumption of Web crawling so as to explore more states and increase the coverage of the crawling in Web applications.

4.3.1 Time Performance

In the process of crawling an Ajax-enabled Web application, as the application transitions to different states, the crawler requires to handle the transitions and track and store the states and navigational paths between them. In order to track the transitions, every time an event is fired on the Web application (line 14 of algorithm 3.2), the crawler checks if the Web application has transitioned to a new state or not (line 16). This task is done by comparing the states of the Web application before and after firing the event. We believe this practice results in some inefficiencies that creates optimization opportunities.

In order to improve the time performance of the crawling, we aim at improving the mechanism the crawler uses for state comparison. To perform the state comparison, the crawler maintains the two states (before and after firing events, denoted by s and $detectedState$) in the memory. These states are extracted from the browser by taking snapshots of the DOM tree of

the application (line 15). Afterward the crawler compares the two states in order to decide if an alteration has happened to the state of the application or not. However, extracting the DOM from browser, initiating a new state object, and comparing the objects after each event is very time consuming. Hence we suggest we should eliminate these steps whenever we can find faster alternatives.

If no event attribute (e.g. "onclick") is added to an HTML element, firing events (e.g "click") on the element will cause no change to the state of the Web application. If we have an alternative way that can notify us that no changes have happened to the DOM, we can safely skip the whole comparison process and save the time that would have been spent otherwise. However, the alternatives must be fast enough to overtake the current comparison mechanism.

4.3.2 Memory Performance

Memory consumption of the crawling process is significantly considerable. Expensive DOM related operations such as comparison, stripping and string replacements impose a sizable memory requirement on the crawler. In addition, IO processes, mainly used for communications with the browser add substantial memory overhead to the crawling. Furthermore, during the exploration of Web applications, the crawler builds an inferred state machine in the main memory. This huge state machine, storing the states discovered in the crawl session as well as the transitional links between the states, consumes almost half of the memory allocated to the crawler.

In order to improve the memory performance, we focus on the mechanism currently employed for managing the state machine model. Storing, retrieving and assuring the uniqueness of states are the key tasks carried out in the crawling process and are directly related to the state machine model. As a Web application is crawled, states are discovered one by one and added to the state machine. The memory is allocated gradually for each detected state. However, the main memory of the system is limited. Hence, at some point, when a specific number of states are stored in the state machine, there is no capacity left in the main memory for allocating space to crawling tasks (including adding a new state). As a result, the crawling process halts (on errors), and having covered a specific number of states the crawler quites on an out of memory error.

In order to improve the state space coverage in crawling Web applications, we have to dismantle the memory limitation. Considering the huge size of the stateflow graph, we target freeing the memory required for stor-

age and retrieval of the states and transitions between the states. We aim at freeing the memory so it can be allocated to the rest of the tasks in the crawling process. If we are able to do so, we could continue the crawling further and discover more states and subsequently cover a greater number of states than the original crawler does.

4.4 Methodology

In this thesis we investigate if application of a number of enhancements to the crawling process can result in time and memory performance optimization in crawling Web applications. Specifically, tracking the changes in DOM trees of Web applications is enhanced in three different methods as an endeavor to optimize the time spent in states comparisons. In addition we investigate how utilizing a graph database for storage and retrieval of the state machine model affects memory performance of the crawling process.

Crawljax has various extension points, called plug-ins, which can be utilized for performing additional tasks at specific points in the crawling process. We also add new extension points to the Crawljax. Availing of the current and new extension points we apply our alternative methods for tracking DOM-tree changes in the applications. In order to expedite the state comparison process we relinquish the change tracking to agents that we developed to work on the browser side instead of the crawling engine's side. This way we install an agent in the browser side and aim at bypassing the expensive DOM extraction, initiation and comparison operations. Proxies (explained in next chapter) and Crawljax plug-ins are used to install the state tracker agent.

There is a sizable graph data structure in the crawling process, called stateflow graph, which holds the main output of the crawling process. Aiming at making memory utilization more efficient, we introduce a new component to the crawler for replacing this data structure and handling the storage and retrieval of the states transitions model. The component that we introduce interfaces a graph database to store the stateflow graph structure and data in a database rather than main memory. This method is expected to reduce the memory consumption, however it might impose an IO overhead affecting both time and memory performances. It is the experimental results that will determine to what extant this method is effective in boosting the performance of the crawling.

In order to assess the effects of the proposed enhancements on the time and memory performances of the crawling process, we conduct experiments

on ten Web applications from Alexa's top Websites list. As these Web sites are complex Web applications composed of manifold components, they are not deterministically crawlable. That is, crawling the same Web application multiple times does not result in the same sequence of states each time. As such, in order to mitigate the effects of this nondeterministic behavior, we crawl each applications multiple times and take average over the collected results.

In the next two chapters we explain in details how we applied the two enhancements to the crawling process. In addition, the way experiments were conducted is discussed and the results of the experiments are also presented at each chapter. The next chapter covers the improvements we proposed for state transition management and the chapter after that discusses improving the memory consumption of the crawling process.

Chapter 5

Optimizing State Transition Management

The crawler models the Web application being crawled by inferring a state machine representing the states of the application and the navigational paths between the states. Therefore, keeping track of the client-side states and the transitions between the states is a substantial task in the crawling process. In order to manage the state transitions, the crawler communicates with the browser in which the application is running. These communications include taking a snapshot of the state of the application at certain occasions to test if any changes have been made to the state of the Web application (lines 15 and 16 of algirithm 3.2). As these sub-tasks are expensive operations in terms of the time they consume, we investigate methods of optimizing the transition management aspects of the crawling process. Throughout this chapter the concept of the "state" of a Web application comes up frequently. Hence we shed some light on what we refer to as the state early in the chapter to improve the clarity.

Web Applications States The state of a Web application at any given point of time is dependent on values of multiple components of the Web application. This multivariate concept includes both server-side and browser side variables. The browser side state itself, cab be delineated to the values of the DOM-tree elements, the values of the JavaScript variables, JavaScript functions and some less frequent elements such as Web Storage [13]. However, as the crawler focuses on the DOM-tree of the Web application, in this thesis, what we refer to as the state of the Web application is as follows:

- In the browser the state of the applications is the DOM Object's state.
- In the crawler, the state is the string representation of the DOM-tree after being processed and normalized (stripped).

The crawler representation of the state has less details. For example some white spaces such as tabs and carriage returns are removed from the

original DOM tree. Hence, chances are that minimal changes to the DOM-tree are filtered by the crawler. The reverse cannot be true as every change in the crawler side must be originated from some change in the browser DOM-tree.

5.1 Tracking State Changes in Web Applications

Currently what Crawljax does to track the changes applied to the states of Web applications is as follows. After firing an event on some specific candidate element in a page of the application, the crawler retrieves the current state of the application from the browser. For example, in photo gallery application, Crawljax clicks on one of the photo albums. As a result the GUI of the photo applications changes so that a number of photo thumbnails are loaded into the view. Crawljax retrieves the DOM of this new view of the application.

Afterward the crawler constructs suitable data holders for shaping the raw data retrieved from the browser into usable objects representing the the state of the Web application. Then it compares this newly retrieved state (photo thumbnails DOM) with the previous state (the home page) so as to test if changes have been made to the application. Performing these sub-tasks results in expensive inter-process communications (e.g. between the browser and Crawljax), string operations, object construction and a number of sizable IO operations for sending the state of the application from browser to the crawler. Here we investigate alternative approaches for tracking the state of the application aiming at eliminating or bypassing some of these operations, whenever possible, to improve the time efficiency of the crawling.

Investigating the way currently the crawler performs the transition checks, we came up with the idea of eliminating unnecessary comparisons. In particular, if no event attribute is attached to an element, firing events on the elements causes no change in the state of the Web application. Hence, in these situations, if we can be notified that there is no change in the application since last time, we can avoid:

1. Retrieving a large amount of data representing the state of the Web application from the browser

2. Processing the raw data representing the state of the application to initiate the objects representing the state

3. Performing a comparison on two usually large objects representing the previous and newly retrieved states

The key point here is that we do not aim at eliminating these steps completely; We only want to find alternative methods that are able to bypass the unnecessary comparisons. To that goal, we need to find an effective way to get notified of changes as they are made to the state of the application.

5.2 Foundations for the Proposed Techniques

We employed a number of different technologies and techniques in our proposed methods for improving the management of states transitions. We briefly touch upon each concept and different options that we had and enumerate their points of strength and disadvantages. We make an introduction to different options available for tracking DOM mutations and the proxy technology that we utilized in this project.

5.2.1 Tracking DOM Mutations

DOM mutations are any changes made to the DOM tree. The mutations include addition, deletion and modification of the content of the elements of the DOM and also structural changes to sub-trees of the DOM. Changing the name of an attribute, deleting a node in the DOM and modifying the textual content of a node are examples of DOM mutations.

Crawljax, the Web application crawler whose performance improvement is the subject of this thesis, crawls Web applications and builds a stateflow graph of the transitions between different states in a Web application. For the purpose of its crawling, Crawljax considers the current state of the DOM tree as the state of the Web application under crawl at any given point of time. As such, tracking the changes made to the DOM, namely DOM mutations, is of high importance in identifying state transitions during crawling sessions.

As Web technologies and standards have evolved over the past decade, different mechanism and APIs have become available to Web developers for tracking DOM mutations. DOM Mutation events, Mutation Observers, and finally mutation-summary library are main options for tracking DOM mutations in developing Web applications. We opt for utilizing the mutation-summary library in this thesis. We justify this choice of technology by explaining each option in next sections.

5.2.2 Mutation Events

Mutations events [14] interface which was introduced in DOM Level 2 is a mechanism for tracking the changes made to the DOM. As the design of the Mutation Events interface was considered to be flawed it is in the process for being dropped and is considered as depreciated. There are a number of performance and bug-causing issues associated with this interface: the slowness of the mechanism; triggering too many reports upon the occurrence of changes to the DOM in a synchronous way; and, leads to development of crash-prone Web applications. [15]

5.2.3 Mutation Observers

Mutation Observers provide another mechanism for receiving notifications about changes made to the DOM tree. Mutation Observers are introduced in DOM3 [16] and are designed to replace Mutation Events for tracking alterations of the DOM. Mutation Observers provide call back functions to handle the changes. This has the advantage of handling multiple changes as a group whereas in the mutation events multiple events are triggered for every single alteration.

5.2.4 Mutation-Summary Library

Mutation-Summary Library [17] is an open-source library implemented in JavaScript and provided by Google®. Being implemented on top of Mutation Observers API, the mutation-summary library provides a more reliable way for tracking the changes made to the DOM. The improvements of this library over the Mutation Observer API include the ease of using its API, and grouping and filtering DOM alterations before reporting.

The mutation-summary library works in a way that it takes snapshots of the DOM in specific intervals and then compares these snapshots in order to find and report the differences between two subsequent snapshots. This means that the transient changes that dissipate shortly, in such a short time that they do not last till the next snapshot, are not reported. As the state comparison intervals in Crawljax are significantly longer than those of Mutation Observers and mutation-summary Library, we are interested only in more durable changes that last for a considerable period of time. Hence, the summarizing feature in this technology is very amenable to our use-case. However, in use-cases where transient DOM state changes are of importance, this library should not be used [18].

5.2.5 Proxies

A proxy is a concept in computer networking which allows implementing encapsulation in the design of network architectures [19]. A proxy server, is a server intercepting the communications between a client and its server. Intermediating between the client and the server , proxies can access and modify the request from client before sending it to the server. Likewise, proxies can also access and make alterations to the response from the server before handing it to the client. Proxies are of different types and are used for various purposes.

Different types of proxies include but not limited to Open Proxies, Forward Proxies and Reverse Proxies. Providing anonymity on the Internet, filtering, caching and eavesdropping are examples of using proxies. Proxies can be implemented in different layers of computer networking. Proxy servers are in the form of software application or a physical device such as a router. In this thesis we set up a software proxy intercepting HTTP and HTTPS communication in the application layer.

Having explained the foundations upon which our proposed methods are drawn, we continue the discussion by presenting the alternative methods we came up with for improving the time consumption in the crawling.

5.3 Alternative Methods for Tracking State Changes

We aim at bypassing unnecessary comparisons. Unnecessary comparisons happen when an event is fired on some element but it does not lead to any change in the DOM-tree, even a single byte. The current crawler, however, performs the three steps mentioned above. We propose the idea of installing a light weight agent in the browser side, instead of in the crawler side, to track the changes in the DOM-tree. This agent should have the following characteristics:

- It must be able to track every single change applied to the DOM.
- It should not impose considerable time overhead on the browser and as a result on the crawling process.
- It must provide some interface that can be utilized for communicating from the crawler

We came up with three different solutions to build our agent with the aforementioned desired characteristics. However, it is the experimental results that show how efficient the solutions are. Our solutions have three components as it is shown in the figure 5.1. The injector component is responsible for installing the agent in the browser side. The agent itself is responsible for tracking DOM mutations and setting the mutation flag whenever some changes are made to the DOM. Finally a communication component that retrieving the mutation flag from the agent in the browser is its main responsibility.

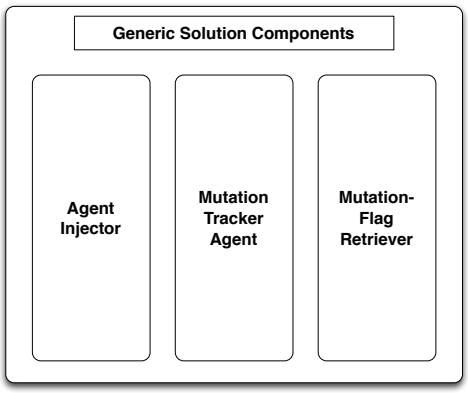

Figure 5.1: Generic solution components

We present the common approaches and technologies utilized in all alternative methods we proposed in the next section. Following the commonalities the differences are presented and each method is explained in details.

5.3.1 Shared Characteristics of the Solutions

Regarding the DOM change tracking part we utilized the Mutation-Summary library to get notified of every single change made to the DOM. The communication component of the agent was made feasible by introducing new extension points to the crawler. Finally, in order to install the agent, we made use of three different methods utilizing multiple technologies.

The Mutation-Summary library is very flexible in allowing us to define what type of changes are of importance to us. As every single change in the DOM-tree has the ability to be translated into a crawler-side state change, we set the mutation tracker capabilities to track all types of change, including textual and structural changes in forms of additions, deletions and

substitutions. In addition we designed a flag for storing the mutation status of the DOM-tree. This mutation flag is originally reset to false and upon every single change made to the DOM it is set to true. True means that some changes have been made to the DOM-tree and hence a full state comparison is required.

A new extension point was introduced to the crawler to enable communications between our agent and the crawler. As it is shown in algorithm 3.2, the Crawljax algorithm fires an event on a candidate element, waits for a certain amount of time and then retrieves the DOM-tree from the browser. However, we made it possible to add additional functionalities at the very exact point before retrieving the DOM-tree from browser. We utilized this extension point to build the communication part of our agent. What we do at this point is that instead of retrieving a huge DOM-tree from the browser, we retrieve our mutation flag; If the flag is set to true we proceed with the ordinary full comparison process, otherwise, assured that the state of the DOM has not been changed we bypass the unnecessary DOM-tree retrieval and comparison steps.

Now we move on with explaining each alternative solutions in details and present the way they differ and how they are improved incrementally over one another.

5.3.2 Proxy-Based Solution

In the first solution we utilized a proxy to install our agent in the browser side. The agent is installed in the browser-side by intercepting the information that is sent by the Web server of the application being crawled. The information sent by the Web server is processed and whenever a new page is sent to the browser the agent is appended to the page so as to track the alterations of the DOM-tree. We set up a Web server to host the Mutation-Summary library so it can be included as part of our agent. The Mutation-Summary is hooked to the application as an external JavaScript. The communication with the agent is performed utilizing the newly created extension point explained earlier. The components of the proxy-based solution are outlined in figure 5.2.

The proxy helps us achieve full control over data exchanged between the browser and the server and install our agent smoothly. However, it imposes a considerable latency on processing the information and as a result the time it saves is less than its overhead. Hence we continued improving the agent in the tow versions that are described in the following sections.

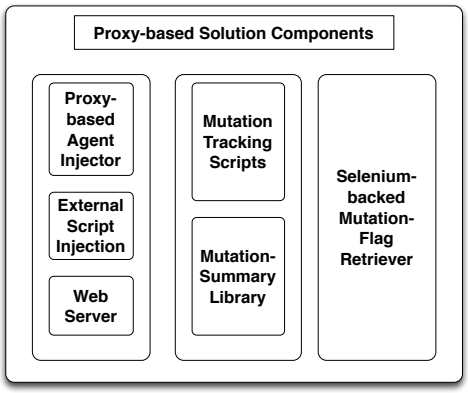

Figure 5.2: Proxy-based solution components

5.3.3 Plugin-Based Solution

As depicted in figure 5.3, in this version we retained the Web-server and the communication point, but we eliminated the proxy component to achieve time efficiency. As such, the role of the proxy, which was installing the agent in the browser side, is carried out by building a new plug-in for the crawler. This plug-in is attached to the crawler at an extension called On-Page-Load plug-in. Hence, the plug-in installs the agent every time a new page is loaded in the browser. In addition, whenever the communication component attempts to retrieve the mutation flag, it checks whether the agent is present in the browser. If the agent is not present, it injects it again so it is installed and ready to function for the next iteration. This version of solution works significantly more efficiently than the proxy-based solution and outperforms the default behavior of the crawler. We, however, aspire to further improve it as explained in the the second plugin-based method.

5.3.4 Plugin-Based Solution with In-Memory Agent

The performance achieved by the plugin-based solution realized our goal for improving the time consumption of the transition management. However, we had an intuition that placing the mutation-summary library on a Web-server and fetching it repeatedly imposes a time overhead on the solution. Hence, we designed another solution in which we eliminated the Web-server as shown in figure 5.4. In this solution, the installer component holds a copy of the library and injects it directly to the Web page simultaneously as

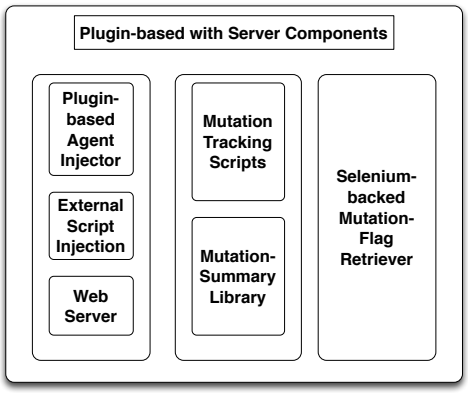

Figure 5.3: Plugin-based solution components–external injection

the core of the agent is injected. The difference here is that in the previous version, a reference to the mutation-summary library was injected in the page and the browser had to load the actual library itself, but in the new version the actual library is injected directly. The other components are the same as the Plugin-based Solution.

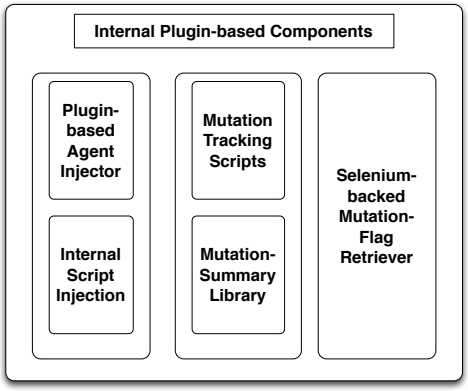

Figure 5.4: Plugin-based solution components–internal injection

Having covered the three enhancements proposed for improving the time consumption of the crawling, we present the experiments conducted to evaluate the three solutions in the next section.

5.4 Evaluation

We conducted a series of experiments to answer the research questions that shaped the objectives of this thesis. In this section we break down the first high level research question into three specific questions. Afterward, we present the design of our experiments, the methodology of conducting them and the results achieved from carrying out these experiments.

5.4.1 Research Questions Broken Down

Our first high level research question targets improving the time consumption of the crawling process. Having explained more details about our enhancements to the state transition management, we delineate RQ1 into more specific research questions to investigate the performance of our solutions. We have three research questions associated with improving the time performance of the crawling process. As explained in details earlier, these solutions are proxy-based, plug-in based with Web server and plug-in based with in-memory injection. Each research question addresses the time performance of one of these solutions. Hence we state the following sub-questions:

RQ1.a : To what extent does the proxy-based solution for managing state transitions improve the time performance of the currently employed mechanism for managing state transitions?

RQ1.b : To what extent does the Plug-in-based solution for managing state transitions improve the time performance of the currently employed mechanism for managing state transitions?

RQ1.c : To what extent does the Plug-in-based Solution with in-memory agent for managing state transitions improve the time performance of the currently employed mechanism for managing state transitions?

We will investigate these question and explain the experiments conducted to answer these questions in details in the following sections. We shed lights on how we investigated the effects of replacing the default behavior of the crawler by our alternative solutions.

5.4.2 Experimental Design and Methodology

We designed a serious of experiments to compare the enhanced versions with the default version of the crawler to answer our research questions. In each experiment we evaluate the enhancements made to the crawler against the

default behavior of the crawler. In each enhancement proposal additional functionality has been introduced to the crawling algorithm. These enhancements help track the mutations of DOM trees to bypass unnecessary comparisons. On the other hand, these enhancements can potentially introduce time overheads to the system. Hence we devised a number of experiments to measure the variables pertaining to the efficiency and correctness of the various solutions in hand.

In these experiments the main variable measured is the time consumption of default vs. enhanced crawling. However, we measure additional variables to achieve more insights over the performance and the correctness of the solutions. First, we measure the time overhead of the mutation tracker agent i.e. the proposed solution. In addition, we measure the time spent on comparing identical DOM trees whose comparison is unnecessary. We performed more measurements whose variables are explained in the following.

First and foremost, we measure the time consumption each alternative solution imposes as an extra overhead on the system. This time is divided into two categories. First, there is an overhead associated with installing the agent in the application. This is the time required to intercept and modify the responses sent from the server for the proxy-based solution and plug-in injection time for the two other solutions. The second category is the time each solution needs to retrieve the mutation flag. Furthermore, we measure the time spent on performing unnecessary comparisons which must be bypassed. These unnecessary comparisons are bypassed by the enhanced solutions. Another variable that we measure is false negatives. A false negative happens when the mutation flag shows no change in the state of the DOM but actual DOM comparison informs us of mutations in the state of the application. The total time of each experiment, the size of the DOM-trees, number of events fired, number of bypassed comparisons, and total number of states crawled are additional variables that we measure in the experiments.

Experimentation Objects In these experiments we use the top most visited Websites from Alexa list. These Web applications crawled as the objects of the experiments are listed in table 5.1 along with the IDs assigned to them. These IDs are specially used for effiency in using the limited space in the tables and charts illustrating the experiments results.

The dynamics of Web application responses require careful attention in designing the experiments. One important issue that needs careful attention is that when performing an experiment on the client side of a Web applica-

Table 5.1: State transition management experiments objects

ID	Web Application
ggle	www.google.com
wkpd	www.wikipedia.com
live	www.live.com
twtr	www.twitter.com
qq	www.qq.com
amzn	www.amazom.com
lnkn	www.linkedin.com
tbao	www.taobao.com
yndx	www.yandex.ru
yhoo	www.yahoo.com

tion, the server side characteristics should be taken into consideration. It is quite possible that inputing the same sequences of events to the GUI of a Web application result in two different responses from the server. In order to mitigate the effects of the non-determinism induced by the server side and the network characteristics, we repeat each experiment five times for each Web application.

We measured the time consumption by means of the System class time measurement capability. The number of maximum states for each experiment was initially set to 30. However after conducting the experiments we observed that if some applications are allowed to run for a maximum of 24 hours some of them could use up the default heap space and result in a heap space error. Hence, we changed the heap maximum size and repeated the experiments with the maximum number of states set to 10 state. The results of performing the experiments along with precise definition of the variables measured are presented in the next section.

5.4.3 Results

Each solution is evaluated by crawling all the object Web applications. For each experiment, one Web application from the objects is crawled by the crawler. The time saved by the solution and the time overhead imposed on the crawler by the solution is measured in each experiments. The experiments are run five times and the average (arithmetic mean) is taken over the results. Result for the proxy-based, plugin-based with Web server,

and plug-in-based solutions are presented in the figures 5.5, 5.6, and 5.7 respectively.

Designing the experiments, we strove to measure all the meaningful metrics that are of importance to the evaluation of our proposed methods. Here we explain these variables before presenting the results in the tables and charts.

Saved Time : The time improvement achieved by introducing the new solution. That is the time spent on unnecessary comparisons (i.e. The time our solutions save). This includes the time that is spent on communicating with the browser and retrieving the state of the application. The additional overheads such as constructing objects for states are also included in this variable. This is the main variable that proposed enhancements must maximize.

Overhead : The time overhead imposed by the new solution to the crawler. This is the main variable that proposed solutions must minimize. Overhead time is the sum of flag time and proxy time for the proxy-based solution. This is the sume of flag time and plug-in time for the plugin based solutions. These are defined next.

Proxy : The time consumed by the proxy. In the first solution which proxy plays a significant role we measure the time overhead of intercepting communicants for injecting our agent.

Plugin : The time consumed by the "plug-in" based injections. In the two plug-in-based solutions we measure the time overhead of injecting our agent.

Flag : The 'time' consumed for retrieving the mutation "flag". This includes the time required for communication with the browser, executing javaScript in the application and retrieving the value of the mutation flag.

Time : The total time of an experiment. That is the time beginning from start of a crawl session until either reaching the maximum number of states or quiting the experiment for other reasons such as exhausting the state space of the application under crawl.

Events Fired : Total number of "events fired" on a Web application by the crawler. The crawler fires events such as the click event on specific elements in the applications under crawl.

Bypassed : The number of comparisons that are bypassed because the agent notified the crawler that no change has been made to the state of the Web application.

States : The number of states crawled in the experiment.

FN : The number of false negatives. False negative in this contexts are cases where the mutation flag suggests bypassing a comparison because no change has been reported by the agent, but the actual comparison reveals that changes have been made to the state of the application and have been recognized by the agent.

FP : The number of false positives. In this context a false positive happens when existence of changes are reported by the agent but the actual comparison decides the change is not significant enough to be considered as a state change.

Having covered the variables and acronyms that are to be used in the results we present the results achieved from conducting the experiments in the following section.

RQ1.A: Time Performance of the Proxy-Based Solution The proxy-based solution was thoroughly assessed by crawling the experiment objects multiple times. Figure 5.5 shows the main variables that were measured in the experiments:

Improvement: The time the proxy-based solution saves by eliminating unnecessary comparisons and communications.

Overhead: The time overhead imposed by the proxy-bases solution.

As it is presented in figure 5.5, proxy-bases solution time overhead is more than the time it saves by improving the crawling process. The new solution does not overtake the default behavior significantly. The new solutions performs similar to the default version for four of the the ten cases. The default crawler performs significantly better on six objects out of ten.

The proxy-based solution does not improve the time performance of the state transition management process.

Table 5.2 presents the the experiments results in more details. For each Web application the time saved by bypassing unnecessary comparisons are

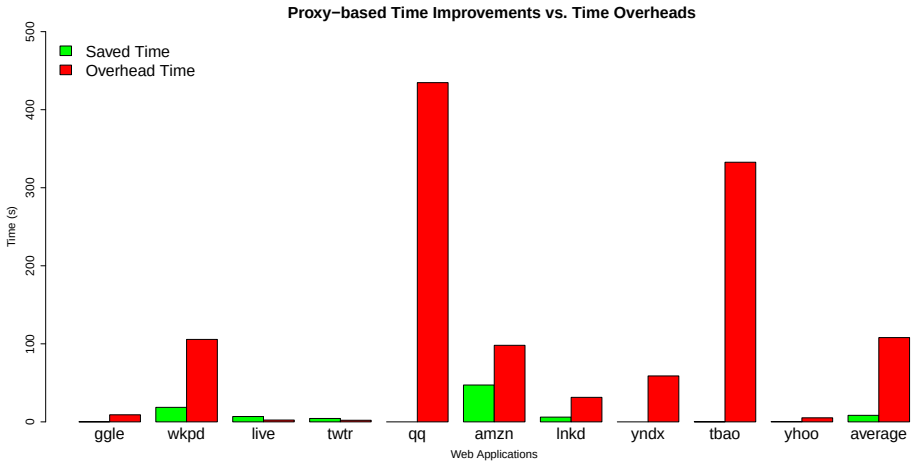

Figure 5.5: Proxy-based solution time performance: the time overhead imposed by the proxy-based solution is much greater than the time it saves by bypassing unnecessary comparisons. Hence, the default crawling performs more efficiently. The bars represent the averages taken over 5 rounds of experiments presented in table 5.2.

listed. In addition, the time overheads, one for retrieving the flag and another one imposed by the proxy, are shown separately. These two items are also summed up together as the total overhead and are listed under the "Time Overhead" title. The total time of the experiments are also included in table 5.2. The values shown in table 5.2 are averages taken over five rounds of experiments. Further statistical measures about the results are also provided; standard deviation in table 5.3, absolute value of coefficient of variation (i.e. relative standard deviation) in table 5.4, maximum values is table 5.5, and minimum values in table 5.6.

Table 5.7 presents the detailed data about the number of events fired on experiment objects and how many of these events resulted in new states. It also includes how many times the agent bypassed unnecessary comparisons. The column, "Events Fired" shows the total number of events fired on the application. In the default crawler, each fired event results in a comparison to check whether the state of the application is changed or not. However, the proposed solution bypasses a number of these comparisons. These measured numbers of bypassed comparisons are listed in the "Bypassed" column. The total number of states crawled in the experiments are included in the "State"

Table 5.2: Proxy-based time consumption (milliseconds); the proxy-based "time overhead" is much greater than the time improvements i.e. "saved time".

ID	Saved Time	Proxy Time	Flag Time	Time Overhead	Total Time
ggle	138.2	8931.6	130	9061.6	57113.4
wkpd	18601.4	83642.8	22008.2	105651	243323.4
live	6851.4	1606.6	739.4	2346	97208.2
twtr	4359.8	1558.8	496.8	2055.6	202418.8
qq	0	434564.8	97.4	434662.2	951066.2
amzn	47224.2	97183.8	914.8	98098.6	341746.6
lnkd	6087.8	30472.2	949.6	31421.8	114203
yndx	0	58768.8	91	58859.8	67407.2
tbao	132.6	332581.8	97.2	332679	616755.6
yhoo	327	5152.2	46.6	5198.8	26142.6
average	8372.24	105446.34	2557.1	108003.44	271738.5

column. False negative and false positives are also presented in "FN" and "FP" columns receptively.

RQ1.B: Time Performance of the First Plugin-Based Solution The first pulgin-based solution, which utilizes a Web server for hosting the mutation-summery library is assessed for its time performance. We present the results of the experiments in figure 5.6. We performed the experiments on the object Web applications to compare the time performance of the proposed plugin-based solution against the default crawling process. In these experiments multiple variables are measured. However, the main two variables that are present in figure 5.6are as following:

Improvement: The time the solution saves by eliminating unnecessary comparisons and communication steps when there is no change in the state of the Web application after firing an event.

Overhead: The time overhead imposed by the plugin-based solution.

As it is shown in figure 5.6 the sever-utilizing plugin-based proposed solution performs more efficiency in the carrying out the state transition

Table 5.3: Standard deviations of proxy-based time consumption

ID	Saved Time	Proxy Time	Flag Time	Time Overhead
ggle	9.44	973.08	5.24	977.29
wkpd	2470.24	6258.46	2855.96	8666.19
live	1039.96	157.61	82.49	205.57
twtr	406.34	87.45	50.11	76.06
qq	0.00	22790.28	6.27	22786.71
amzn	4365.87	14216.21	104.32	14300.70
lnkd	891.01	3187.10	137.90	3298.76
yndx	0.00	4028.57	12.88	4036.75
tbao	12.30	27018.00	9.26	27025.11
yhoo	36.93	290.98	3.71	290.22

management tasks. As it is depicted in the bar chart in figure 5.6, on average the proposed solution consumes less time than the default crawling process. In addition, in half of the cases the proposed solution consumes considerably less time than the time the default crawler spends on performing unnecessary comparisons. It is only in one case that the proposed solution is more of an overhead than an improvement, and in four cases out of ten the difference is not very significant.

The plugin-based solution, utilizing a web server, improves the time performance of the state transition management process by 253.34%. That means the time consumption ratio of default to proposed is 3.5334, which means the time consumption is reduced by 71.69%.

Aside from figure 5.6 which shows the gist of the results of the experiments, we present the details of the data gathered in the experiments on the the sever-utilizing plugin-based proposed solution in table 5.8. As it is shown in table 5.8 time improvements of the solution in the experiments are listed by Web application. The improvements are listed in the “Saved Time” column, and the time overhead is provided as a total number under the “Time Overhead” title. The time overhead is also broken down in to its constituents, plugin time and mutation flag retrieval time in columns “Plugin Time” and “Flag Time” respectively. The values presented in table 5.8 lists the averages taken over results of five rounds of experiments. Further

Table 5.4: Relative standard deviations percentages (absolute values of coefficient of variation) of proxy-based time consumption

ID	Saved Time	Proxy Time	Flag Time	Time Overhead
ggle	6.83	10.89	4.03	10.78
wkpd	13.27	7.48	12.97	8.20
live	15.17	9.81	11.15	8.76
twtr	9.32	5.60	10.08	3.70
qq	Not defined	5.24	6.43	5.24
amzn	9.24	14.62	11.40	14.57
lnkd	14.63	10.45	14.52	10.49
yndx	Not defined	6.85	14.15	6.85
tbao	9.27	8.12	9.52	8.12
yhoo	11.29	5.64	7.97	5.58

statistical measures about the results are also provided; standard deviation in table 5.9, absolute value of coefficient of variation (i.e. relative standard deviation) in table 5.10, maximum values is table 5.11, and minimum values in table 5.12.

Table 5.13 sheds more lights on the details of how the aforementioned time performance improvements are achieved by the sever-utilizing plugin-based proposed solution. The variables presented in this table have the same meaning as those presented in table 5.7.

RQ1.C: Time Performance of the Second Plugin-Based Solution The second plugin-base solution, which does not utilize a Web server for hosting the mutation-summary library and instead injects it directly, was also assessed thoroughly in the same manner as its two predecessors. We present the gist of the experiments carried out on the ten object Web applications in figure 5.7. As it is shown in the bar chart, the second plugin-based solution performs more efficiently than the default crawling process. As it is shown in the average bars, the proposed solution consumes less time than the default crawler on average. Moreover, in six cases out of ten, the proposed solution takes over the default crawler. It is only in one case that the proposed version consumes significantly more time than the default version.

Table 5.5: Maximum values of proxy-based time consumption (milliseconds)

ID	Saved Time	Proxy Time	Flag Time	Time Overhead
ggle	152	10273	137	10406
wkpd	20647	9 7	24209	115775
live	7468	1783	820	2603
twtr	4664	1683	568	2140
qq	0	469333	103	469427
amzn	51946	107874	6	108880
lnkd	6757	32909	1054	33953
yndx	0	63470	101	63569
tbao	147	369165	107	369267
yhoo	377	5615	52	5661

The plugin-based solution, not utilizing a Web sever, improves the process of state transition management by 197.48%. That means the time consumption ratio of default to proposed is 2.9748, which means the time consumption is reduced by 66.38%.

In addition to figure 5.7 which presents the essence of the experiments results, we shed more lights on the details of the time consumption by the second plugin-based solution in 5.14. The variables presented in this table have the same meaning as those in table 5.8. Here again, the time overhead is presented in two ways. First, it is broken down to its two components, the plugin time and the flag time. In addition, it is presented as the sum of its two constituents in the column "Time Overhead". The time improvements of the proposed solutions is listed in the column "Saved Time". The averages taken over the results of five rounds of experiments are listed in table 5.8. Further statistical measures about the results are also provided; standard deviation in table 5.15, absolute value of coefficient of variation (i.e. relative standard deviation) in table 5.16, maximum values is table 5.17, and minimum values in table 5.18.

Table 5.19 presents the data achieved from experiments which illustrates in details the manner in which the second plugin-based solution bypasses unnecessary comparisons and improves the time efficiency of the crawling process. The variables presented in this table are the same as those presented in tables 5.13 and 5.7.

Table 5.6: Minimum values of proxy-based time consumption (milliseconds)

ID	Saved Time	Proxy Time	Flag Time	Time Overhead
ggle	129	8038	124	8164
wkpd	14511	75281	17168	92449
live	4	1445	609	2105
twtr	3665	1449	442	1965
qq	0	408490	88	408591
amzn	40142	72889	770	73659
lnkd	4569	24990	742	25732
yndx	0	55830	69	55901
tbao	118	299326	85	299411
yhoo	294	4894	42	4942

As presented in the tables and figures so far, in summary, the proxy-based solution does not improve the time performance of the state transition management process. The first plugin-based solution, utilizing a web server, improves the time performance significantly. On average the first plugin-based solution improves the time performance by 253.34%. That means the time consumption ratio of default to proposed is 3.5334, which means the time consumption is reduced by 71.69%. The third solution, which is plugin-based and does not utilize a Web sever, improves the process of state transition management by 197.48%. That means the time consumption ratio of default to proposed is 2.9748, which means the time consumption is reduced by 66.38%.

We will elaborate more on the results when revisiting research questions in the discussion chapter. Having addressed state transition management improvements we continue with presenting the improvements made to the scalability of the crawling in the next chapter.

Table 5.7: Proxy-based bypassed comparisons: the detailed information on events fired, comparisons that were categorized as unnecessary, false negatives and false positives.

ID	Events Fired	Bypassed	States	FN	FP
ggle	10	1	10	0	0
wkpd	169.8	154.2	10	4.2	6.6
live	72.6	62.4	10	8.8	1.2
twtr	50.8	41	10	8.6	0.8
qq	9	0	10	0	0
amzn	92.8	83.2	10	0	0.6
lnkd	100	90.4	10	0	0.6
yndx	9	0	10	0	0
tbao	10	1	10	0	0
yhoo	17.8	7.6	10	0	1.2

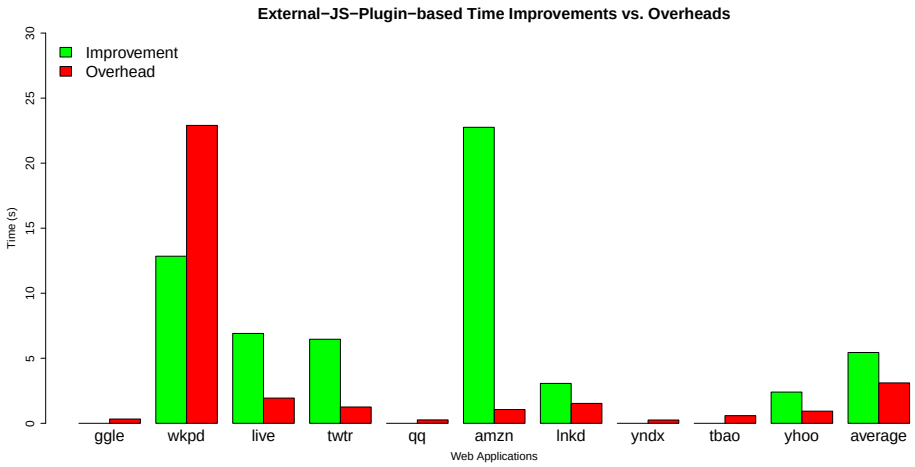

Figure 5.6: External js-plugin-based solution times: the first plugin-based solution, which employs a Web server and injects scripts externally, overtakes the default process. The time improvements on average is much greater than the overheads imposed. The bars represent the averages taken over 5 rounds of experiments presented in table 5.8.

Table 5.8: External js-plugin-based solution times (milliseconds): the time improvements by the first plugin-based solution is much greater than the time overheads it imposes on the crawling process.

ID	Saved Time	Plugin TIme	Flag Time	Time Overhead	Total Time
ggle	0	136.8	190.8	327.6	57113.8
wkpd	12846	144.6	22757.2	22901.8	243323.6
live	6908.6	727.4	1211.8	1939.2	97208.4
twtr	6461.8	147.4	1106.6	1254	202418.8
qq	0	101.8	159	260.8	951066
amzn	22757.2	120.2	939.2	1059.4	341746
lnkd	3066.2	96.4	1432.8	1529.2	114203
yndx	0	109	144	253	67407.2
tbao	0	412.2	175.2	587.4	616755.8
yhoo	2400.4	509	424.4	933.4	26142.6
average	5444.02	250.48	2854.1	3104.58	271738.52

Table 5.9: Standard deviations of external-js plugin-based time consumption

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0.00	12.76	26.54	22.57
wkpd	1808.80	6.77	2015.18	2004.42
live	691.93	53.09	160.99	122.64
twtr	1078.02	15.06	127.21	76.19
qq	0.00	8.17	14.71	16.62
amzn	1354.61	9.65	68.53	76.86
lnkd	353.50	10.45	75.71	111.24
yndx	0.00	8.46	6.44	24.64
tbao	0.00	34.71	24.94	62.02
yhoo	184.15	48.40	33.26	83.67

Table 5.10: Relative standard deviations percentages (absolute values of coefficient of variation) of external-js plugin-based time consumption

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	Not defined	9.32	13.90	6.88
wkpd	14.08	4.68	8.85	8.75
live	10.01	7.29	13.28	6.32
twtr	16.68	10.21	11.49	6.07
qq	Not defined	8.02	9.25	6.37
amzn	5.95	8.03	7.29	7.25
lnkd	11.52	10.84	5.28	7.27
yndx	Not defined	7.75	4.47	9.73
tbao	Not defined	8.42	14.23	10.55
yhoo	7.67	9.50	7.83	8.96

Table 5.11: Maximum values of external-js plugin-based time consumption (milliseconds)

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0	151	211	356
wkpd	14130	151	25032	25882
live	8085	785	1457	2058
twtr	8337	160	1329	1354
qq	0	111	178	279
amzn	24122	130	1023	1154
lnkd	3372	114	1506	1682
yndx	0	119	151	280
tbao	0	467	192	628
yhoo	2664	554	465	1036

Table 5.12: Minimum values of external-js plugin-based time consumption (milliseconds)

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0	124	147	294
wkpd	9765	134	20029	20840
live	6286	654	1090	1764
twtr	5751	130	1029	1181
qq	0	91	141	239
amzn	20484	110	854	985
lnkd	2579	88	1332	1391
yndx	0	97	135	225
tbao	0	379	131	478
yhoo	2184	440	386	813

Table 5.13: External js-plugin-based bypassed comparisons: the detailed information on events fired, comparisons that were categorized as unnecessary, false negatives and false positives.

ID	Events fired	ByPassed	States	FN	FP
ggle	9	0	10	0	0
wkpd	170.6	158.4	10	1	3.2
live	72.2	62.6	10	0	0.6
twtr	69.4	59.8	10	0	0.6
qq	9	0	10	0	0
amzn	64.6	54	10	0	1.6
lnkd	100.4	91	10	0	0.4
yndx	9	0	10	0	0
tbao	9	0	10	0	0
yhoo	18.8	9.8	10	0	0

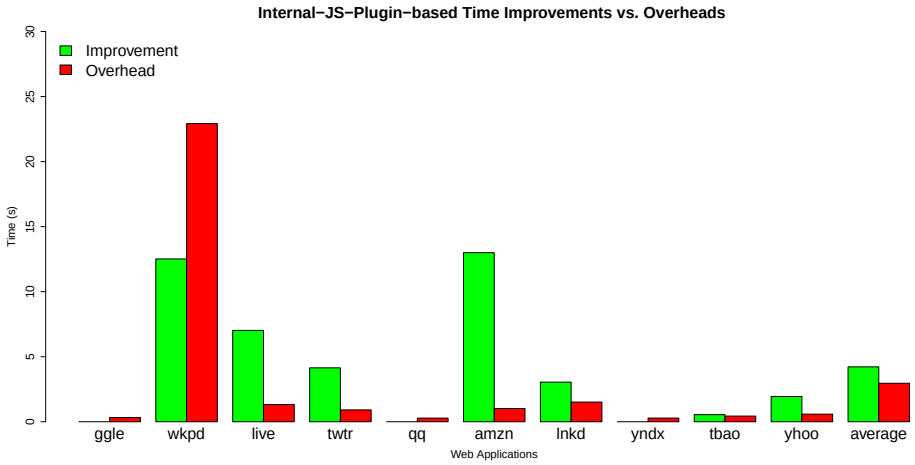

Figure 5.7: Internal js-plugin-based solution times: the time the second plugin-based solution improves is greater than the time overheads it imposes on the process. hence, the second plugin-based solution also overtakes the default state transition management system. The bars represent the averages taken over 5 rounds of experiments presented in table 5.14.

Table 5.14: Internal js-plugin-based solution times (milliseconds): the second plugin-based solution overtakes the default process. The time it improves is greater than the time overhead it imposes on the crawling process.

ID	Saved Time	Plugin Time	Flag Time	Time Overhead	Total Time
ggle	0	163.8	160.6	324.4	112488.6
wkpd	12516.6	169.4	22748.8	22918.2	196764.2
live	7024.2	261.4	1063.4	1324.8	96941.2
twtr	4144.8	123	787.8	910.8	190713.8
qq	0	147.6	133.4	281	233494.2
amzn	12997.6	459.6	558	1017.6	131194.4
lnkd	3046.6	127	1387	1514	115787
yndx	0	102.6	179.2	281.8	65686.2
tbao	546.2	245.8	190.4	436.2	550140.2
yhoo	1941	290	296	586	650855
average	4221.7	209.02	2750.46	2959.48	234406.48

Table 5.15: Standard deviations of internal-js plugin-based time consumption

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0.00	19.28	16.29	14.12
wkpd	1914.28	30.95	1883.50	2698.28
live	629.92	29.74	78.32	80.70
twtr	192.48	9.67	92.94	68.77
qq	0.00	18.15	6.99	25.42
amzn	656.91	37.29	43.82	141.02
lnkd	314.51	5.34	44.67	152.09
yndx	0.00	12.34	9.81	17.43
tbao	33.74	32.67	17.53	35.49
yhoo	191.88	18.49	23.44	35.89

Table 5.16: Relative standard deviations percentages (absolute values of coefficient of variation) of internal-js plugin-based time consumption

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	Not defined	11.77	10.14	4.35
wkpd	15.29	18.26	8.27	11.77
live	8.96	11.37	7.36	6.09
twtr	4.64	7.86	11.79	7.55
qq	Not defined	12.29	5.23	9.04
amzn	5.05	8.11	7.85	13.85
lnkd	10.32	4.20	3.22	10.04
yndx	Not defined	12.02	5.47	6.18
tbao	6.17	13.29	9.20	8.13
yhoo	9.88	6.37	7.91	6.12

Table 5.17: Maximum values of internal-js plugin-based time consumption (milliseconds)

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0	195	176	347
wkpd	15774	224	24571	27275
live	7726	300	1148	1406
twtr	4393	135	899	992
qq	0	178	142	311
amzn	13517	505	619	1243
lnkd	3381	135	1456	1665
yndx	0	123	193	309
tbao	603	300	217	473
yhoo	2195	316	324	623

Table 5.18: Minimum values of internal-js plugin-based time consumption (milliseconds)

ID	Saved Time	Plugin Time	Flag Time	Time Overhead
ggle	0	149	133	308
wkpd	11139	150	20701	20397
live	6321	237	967	1218
twtr	3937	110	701	828
qq	0	134	128	252
amzn	12217	409	510	905
lnkd	2711	120	1332	1349
yndx	0	91	167	262
tbao	518	218	171	396
yhoo	1766	266	266	533

Table 5.19: Internal js-plugin-based bypassed comparisons: the detailed information on events fired, comparisons that were categorized as unnecessary, false negatives and false positives.

ID	Events fired	ByPassed	States	FN	FP
ggle	9	0	10	0	0
wkpd	170.2	158.4	10	1.2	2.8
live	73.2	63.4	10	0	0.8
twtr	49.6	40	10	0	0.6
qq	9	0	10	0	0
amzn	34.6	25.4	10	1.8	0.2
lnkd	100.4	91.2	10	0	0.2
yndx	9	0	10	0	0
tbao	13.8	4.8	10	0	0
yhoo	16.6	7	10	0	0.6

Chapter 6

Increasing the Number of States Crawled

In this chapter, we make the case for improving the scalability of crawling modern Web applications. As mentioned earlier, crawling modern Web applications is a resource intensive process. In particular, managing storage and retrieval of states of Web applications has the potential to overwhelm the memory of the crawler. The memory required for storing the states' data is likely to be of considerable size in modern Web applications. This mainly depends on the average size of the states as well as the number of states in the application under crawl. As the number of states in the state machine increases, the memory allocated to the state machine grows larger and larger. This continues up to a point where no further memory allocation is possible because all memory available to the crawler is already allocated. At this point, the crawler breaks and the crawling process stops. To tackle this problem, we aimed at improving the memory utilization so that a "greater number of states can be crawled" before all memory is allocated.

In this chapter, first we briefly explain the background knowledge and foundations on which the proposed enhancements are laid. As such, we touch upon the way memory management is achieved in Java because Crawljax, the crawler on which we examined our abstract ideas, is implemented in Java. This is done to pave the way for explaining the proposed improvements, the experiments, and the memory analysis. The memory management is followed by explaining how we chose a graph database for application in our solution. In fact, in the process of realizing our ideas into functional measurable capabilities, we needed to select a graph database for creating an alternative solution for improving the memory consumption in the crawling process. Hence early in this chapter, we explain how the comparison is done among all viable options for selecting a graph database.

In what follows, we continue with touching upon the foundation of memory management to help present the suggested improvements, the memory analysis and the experiments.

6.1 Memory Management

The way memory is allocated and freed in the crawler does not solely depend on the way the tool is implemented. In fact, it is only the memory allocation that is performed directly in the manner the design and implementation of the crawler dictates. However, the deallocation part of the memory management is performed automatically by lower level layers of the Java technology which complicates memory measurements and optimizations in this project. Hence we provide a brief introduction to the founding concepts of memory management in Java Virtual Machine on which the crawler is executed.

6.1.1 Java Virtual Machine Heap

JVM heap is the part of JVM memory in which all class instances and arrays reside. All threads in a JVM share the same heap. JVM heap is created at the beginning when the JVM starts up. The heap can be of either a variable or an invariable size. In the former case, the heap can increase in size if the application requires more memory. Likewise, the heap size can also be reduced if the allocated heap is too large for the current consumption requirements of the application [20, 21].

There are also a number of JVM options that can be set (e.g. as command line arguments) to specify the initial, minimum, and maximum size of the JVM heap. Aside from being limited by the maximum size option (-Xmx), the size of the physical memory available to the underlying system on which the JVM runs is an upper limit for the heap size.

JVMs can also implement memory management processes to elevate the efficiency of memory utilization. These management processes are sometimes referred to as garbage collection (GC) processes. In order to make room for creating new objects, memory management processes remove the objects that are not necessary to be detained in the memory anymore.

However, if an application consumes the heap memory up to a degree that there is no more space left for creating further required objects, the out of memory error happens for the JVM and the application execution stops:

> If a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an OutOfMemoryError.
> — JVM Specification [21]

6.1.2 Garbage Collection

Garbage collection or memory management in Java is the process of removing the objects that will not be used in the future so as to to prepare allocatable space for new objects in the memory. Contrary to C and C++ languages which require the programmer to be in charge of object deallocation, Java takes care of clearing memory from unused objects. In what follows, memory management concepts such as the nursery, young generation and old generation are described to shed light on how JVM performs the memory management task.

As it is depicted in figure 6.1, there are two significant divisions in the JVM heap: the nursery (or young generation) and the old generation. The nursery is the part of the heap where new objects are allocated. When the nursery is full the memory management process performs a garbage collection in the nursery to make room for new objects. At the same time, the objects that are still in use, and as a result are not collectible, are promoted to another part of the heap which is called old generation. The garbage collection on the nursery is called young or minor collection. When the old generation space is full the garbage is collected in that place too. The GC on old generation space is called old or major collection.

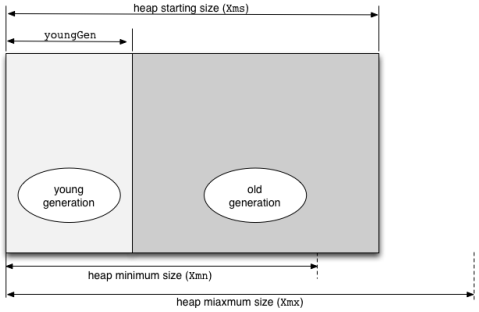

Figure 6.1: Java heap divided into generations and the runtime options. Source: [2].

The concept of having a nursery in the heap is introduced to improve the performance of garbage collection. As "most objects are temporary and short-lived," [21] a two generational heap allows implementing two types of GC to improve the efficiency. As the most recently allocated objects are in the nursery, the GC required for the nursery could generally be much faster than the GC of the old generation space or the GC of a single-generational

heap (a heap that treats old and new objects in the same manner in terms of collecting them).

Another improvement in the garbage collection process is that the JVM reserves a part of nursery for the most recent objects. This area is called the keep area and it is exempted from the first upcoming young collection. The reason behind this is that the objects that are just created in the keep area are very likely to survive the first next young collection and hence be promoted to the old generation. The problem here is that once the objects are promoted to the old generation space, they become immune to young collection and hence are kept in memory till the next major garbage collection. However, the keep area objects are either cleared or promoted by the second upcoming young collection.

Having covered the foundation of memory management in Java which will help clarify the memory related concepts throughout the chapter, we continue with discussing the qualities identified as requisite for alternative methods aiming at improving the memory consumption.

6.2 Criteria for Alternative Solutions

Optimizing the memory utilization is one of the goals of this work. As mentioned previously, there are multiple drivers that cause extensive memory consumption in modern Web application crawling. We, however, focus on the memory requirements driven from managing the storage of states in the crawling process. If the state space of a Web application is of considerable size and states of the application are sizable on average, the crawling process runs out of memory at some point and stops crawling. In order to mitigate this limitation, we propose our ideas for improving the scalability of the crawling process.

Investigating the scalability problem of the crawling process, we put forward the idea of creating an alternative method for storing the sate machine containing the states of the application. As the state machine is concurrently accessed and it plays a significant role in the crawling process, our alternative solutions must deliver the following qualities:

- It must provide safe concurrent write and read capabilities for storing and retrieving states.
- It must not impose a considerable time requirement on the crawling process.

- It must have the flexibility for designing a suitable graph structure similar to the structure of the state machine which is a graph data-structure in the crawler.
- The memory overhead imposed by the execution of the solution and of the communications with it must not defy the purpose of the solution. That is, it must use less memory than the default crawler.

Taking the above characteristics into consideration, searching for solutions to achieve our goal of improving memory consumption, we put forward the idea of utilizing a graph database.

Graph databases are optimized for utilization in applications whose data can naturally be presented in a graph like form. In addition, some graph databases provide transaction management and multiple access capabilities that are able to handle concurrent accesses. However, there are many graph databases, with various complexities and capabilities. We have to conduct a comparison to decide which graph database is most suitable for being used in our solution. Furthermore, the time and memory overheads imposed by the selected database will be the focus of our research questions, and they will be extensively investigated in our experiments.

In order to perform this comparison, we need to set out the criteria for performing the comparison including what functionalities need to be supported and what qualities must be held by the database. In the following section we explain how we extract the criteria for selecting the database by reverse engineering the crawler.

6.3 Criteria for Selecting a Graph Database

Based on the investigation of Crawljax, we discovered the required functionalities that the alternative solutions must provide. A preliminary research on the concepts and characteristics of graph databases also helped finding a number of additional criteria for comparing current graph databases. We wield the criteria to sift the available options and choose the right graph databases that fulfills our requirements the best. The guidelines for choosing the database is explained in what follows. They include required functionalities, licensing, partial locks, the ability to have complex nodes in the graph, and productivity and maintenance issues such as offering an API for Java and Maven public repository availability.

6.3.1 Required Functionalities

Based on the investigation of the source code of Crawljax, we deduce that any alternative solutions we provide for the Crawljax state-storage and retrieval mechanism have to provide the following functionalities:

- Concurrent reading from the stateflow graph
- Concurrent writing to the stateflow graph.
- Insuring uniqueness when adding states.
- Insuring the presence of the start and end nodes when adding an edge in the stateflow graph.
- Managing the count of states already in the graph in a block-free and thread safe manner.
- Feasibility of implementing a directed multigraph data structure.
- Finding the previous state of a given state in the state-machine.
- Calculating the path to a given state from the index state.
- Getting all outgoing edges from a node.
- Finding all incoming edges to a node.
- Finding outgoing nodes from a given node or at least Finding the target node of an edge.
- Finding the target node of an edge.
- Checking if an edge exists between two nodes.
- Calculating the shortest paths from a given vertex to another vertex, ideally by Dijkstra.
- Finding all nodes
- Finding all edges

We also identified a number of implicit requirements that we needed to fulfill. These requirements, extracted from investigation of the crawler logic and code are presented briefly as follows. First, we should be aware that whenever changes are made to the state of the application under crawl, the

state must be compared to all previous states in the stateflow graph. Hence we need to have a method for performing these comparisons to ensure the uniqueness of states added to the stateflow graph. However, if the states are stored as primary keys in the database, or if the database does not allow duplicate vertices, the database itself might have an optimized way of performing the comparisons. On the other hand, we might not have to store the whole state object in the stateflow graph and instead it might suffice to store the IDs of the states or their hashed representations. In that case, we have to provide mechanism for linking these states' keys to actual states' data.

6.3.2 Licensing

It is very important for us to ensure that the licensing characteristics of the graph database matches the Crawljax open source licensing. In particular, we consider if it is a propriety software or it is available under some type of open-source license too. As Crawljax itself is offered under the generous Apache version 2 license, we would like to avoid imposing any limitations on users of the crawler. To that aim, we avoid choosing more restricted licenses and proprietary licenses.

6.3.3 Partial Locks

In the default version of the crawler, the methods that add states to the multigraph are synchronized over the entire stateflow graph. Thus when a crawling node is examining the uniqueness of a new state or when a crawling node is adding the new state to the graph, the whole stateflow graph is locked. What is ideal for us is to find a graph database that is able to provide a facility to update, add or delete states and edges without locking the entire graph. Hence ability to lock the database partially is of critical importance to us in the selection of a graph database.

6.3.4 Storing Objects in Nodes and Edges

Crawljax stores various information in nodes and edges of the stateflow graph. As the states and the navigational paths between them comprise multiple objects, our database must be able to store these objects or at least a transformation of the objects in its nodes and edges.

6.3.5 API for Java and Maven Availability

As Crawljax is completely implemented in Java and it utilizes Maven dependency management facilities, we prefer to use a database that provides a Java API and it can be added as a dependency to Crawljax via public maven repositories.

Having covered the criteria and guidelines for selecting a graph database for application in our solution, we go on with presenting the actual comparison and selection of the database in the next section.

6.4 Comparison of Graph Databases

We conducted a comparison among the available graph databases in order to select the most applicable graph database for our application. The comparison is based on the criteria listed in section 6.3. In addition, to those criteria, we considered the design flexibility and the maintenance implications of our design as well. In particular, although some of the triple stores seemed to be promising in terms of the maturity of the software product, we decided not to proceed with triples stores. The reason was to avoid imposing unnecessary complexity to the crawler system. This complexity would be introduced by having to transform a graph like data to a set of complex relationships that could be shaped into triples. Hence, to eliminate the need for this transformation, we omitted triple stores from our final choices.

Table 6.1 shows the comparison of the databases that we considered for utilization in our solution. The shortenings used in the table have the following meanings:

Funct. : The functionalities listed in the guidelines in section 6.3.1

J & M : API for Java and Maven public repository availability.

SCONE : Storing complex objects in nodes and edges.

GDB : Graph database name.

PL : Partial locks.

As this comparison is performed in a more qualitative than quantitative manner, we use a three level scale for each criteria. These levels are depicted as the background color of the cells of the table. These levels are shown in the table 6.2.

Table 6.1: Graph databases comparison: Neo4j emerged to be the most suited graph to our application.

GDB	Licensing	J & M	PL	Funct.	SCONE
Neo4j	GPLv3	Yes	Yes	Yes	Yes
FlockDB	APL2	Not Maven	Yes	No	No
GraphDB	APL2	Not Maven	Yes	Yes	HyperGraph
AllegroGraph	Proprietary	Not Maven	Yes	Partially	TripleStore
BigData	GPLv2	Yes	Yes	Yes	RDF
Filament	BSD	Poor	Yes	Poor	No
Graphbase	Proprietary	Not Maven	Yes	Yes	RDF
HyperGraphDB	LGPL	Yes	Yes	Complex	HyperGraph
InfiniteGraph	Proprietary	Yes	Yes	Yes	Yes
InfoGrid	AGPLv3	Yes	Yes	Partially	No
Titan	APL2	Immature	Yes	Yes	Yes

Table 6.2: Graph databases comparison scale levels

Level	Color
Acceptable	Green
average	Yellow
Not very acceptable	Red

Based on the comparison presented in the table 6.1, it emerged that Neo4j is the graph database most suited for utilization in our application. As it is shown in the table 6.1 this graph database satisfies all our desired characteristics. Especially its extensive locking options is very useful for concurrent access requirements of the crawling process. It also provides an optional multilevel caching that can optimize the time performance of the crawling. It is implemented in Java and is available in Maven public repository. Furthermore, its open-source license fits well with the Crawljax license.

Having covered the foundations on which the scalable solution is based, the discussion is continued by presenting our ideas for improving the number of states crawled in the next section.

6.5 Scalable Solution

Having selected Neo4j as our choice of graph database, we built an alternative solution for storage and retrieval of the state-machine. Aiming at freeing the memory needed for storing the state machine, we utilized the graph database to delegate all the state storage functionalities to secondary memory. We utilized the graph database capabilities to provide all the previous functionalities of the in-memory stateflow graph. In addition, we wrapped the solution in the very same interface as that of the in-memory stateflow graph. Introducing a new component to the crawler, we aimed at increasing the scalability of crawling by exploring a greater number of states.

Handling concurrent accesses to the state-holder data structure is one of the important aspects of the crawling algorithm. Our alternative solution handles multiple concurrent accesses by utilizing the database's transaction management capabilities. The Neo4j graph database transactions support the ACID properties and ensure the safety of concurrent read and write operations.

As the API we provide for the proposed state storage solution is the same as the default one, the changes are encapsulated only within the scope of the storage component. However, as one of the internal differences of the proposed solution is that it has to serializes the live objects into persisted byte arrays so as to be compatible with the graph database acceptable data types. Hence, we changed numerous classes with minor alterations to fulfill the requirement for serializability. Bearing in mind that introducing a database component to the system brings about time and memory trade-offs, we describe them in the following.

6.5.1 Memory Performance Trade-Offs

Delegating the management of the states and transitions between the states to a component that predominantly does not use main memory for storing data frees a considerable amount of space in the main memory. On the other hand, utilization of a database introduces an additional memory overhead. The database (which is not an in-memory one) requires significantly frequent IO operations. These IO operations are memory consuming. In addition, we serialize the states and transitions before storing them in the database. Likewise, we deserialize them before making them alive and use them in the crawling process. The serialization and deserialization processes are of considerable memory overheads. The transactions conducted by the database cause substantial memory overhead as well.

6.5.2 Time Performance Trade-Offs

In the current status of the crawler, the whole state machine is locked when a process wants to add a new state to the data structure. This puts other processes in hold state and increases the time consumption of the crawling. In the scalable solution however, partially locking transactions expedite the process of crawling by eliminating the need for locking the entire state-machine. On the other hand, the scalable solution needs to communicate with the hard disk very frequently. This means the additional latency of the IO operations has the potential to impose a significant time overhead on the crawling. It is however the evaluation that answers how significant it can be. In the next session we discuss the evaluation of the scalable solution.

6.6 Evaluation

In the scalable crawling solution, we aimed at increasing the number of states crawled by improving the memory performance of the crawling. To this aim, we utilized a graph database to free the main memory from being allocated for state storage purposes. We aimed at answering the research questions by conducting experiments addressing the correctness, memory performance and time performance of the solution.

We conducted a series of experiments to answer the research questions that shaped the objectives of this work. In this section we present the design of our experiments, the methodology of conducting them and the results achieved from carrying out these experiments. We discuss the evaluation of the scalable crawling solution and results of the related experiments in the following.

6.6.1 Research Questions

Considering the time and memory trade-offs of the scalable solution, we are interested in investigating how our solution performs in real crawling sessions. As such, we designed a number of research questions to shed light on the applicability of our solution. We defined three research question to pinpoint the goals of the enhancements carried out in this work. These research questions address the correctness, scalability and performance of the proposed solution compared to the default version of the crawler.

In particular we address three different aspects of the proposed solution for improving the scalability of crawling. We investigated if the solution performs the task of crawling in a correct manner. In addition, we measured how scalable the new solution works compared to the default version of the crawler. The time performance of the alternative solution was investigated too. In addition, new research question emerged from witnessing interesting observations during the experiments.

To delve into the goals of the work, here we break down the second research question into three more specific research questions to pinpoint the objectives of the research.

Correctness The first sub-question addresses the correctness of our graph database based solution. This is indeed important because the state-machine plays a very critical role in the crawling process. Especially taking into account the concurrent aspects of the crawler.

RQ2.a : Does (and to what extent) the proposed solution perform the task of crawling a Web application correctly?

This research question, inherently, dictates a type of experiment that is very similar to what is generally carried out in the process of testing software artifacts. Thus, we need to define what correctness means in this context. In fact, similar to software testing, we need an oracle to decide whether the task of crawling is performed correctly or not.

Scalability: Memory Performance The second subquestion addresses the scalability of the new mechanism for managing states storage and retrieval.

RQ2.b : To what extent can the proposed crawler crawl Web applications in a more scalable manner than the default version of the crawler does?

In the context of this work, we define the scalability of a crawler as the ability to crawl and store more states within a given Web application utilizing a certain amount of main memory. Hence, this RQ aims to determine whether the proposed version is able to overtake the the default version in crawling a larger number of states within a Web application while the maximum Java heap space is limited to a certain amount of memory.

Time Performance In the third sub-question we target the time consumption of the proposed solution compared to the default crawler.

RQ2.c : How does the graph database-backed solution perform in crawling Web applications in terms of time consumption?

This research question examines in particular the time the proposed version requires to crawl a Web application compared to the default version of the crawler. We are interested to see which version crawls a specific number of states in a Web application in a shorter amount of time, given a limited amount of memory.

Having stated and explained the research questions highlighting the objectives of the work, we move on with the discussion of how we pursued the answers to these questions by designing the experiments.

6.6.2 Experimental Design and Methodology

Correctness Experiments As the very first step, the correctness must be defined in the context of this work. In order to achieve this goal, we need to

find a baseline for assessing our new solution. This baseline is naturally the original crawler. Hence, we assume that the default version of the crawler crawls a Website correctly. As such, correctness, in this context, means performing the task of crawling in the same manner as the default version of the crawler does . This, in turn, suggests that correctness means producing the same output as the the original version. So the default version can be utilized to achieve an oracle for assessing the proposed version. Moreover, this oracle can be used to determine to what extent the new version performs in equivalence to the default version. As mentioned in the introduction, the crawler builds a model out of the crawling of a Web application in form of a stateflow graph. We utilize this stateflow graph model to compare whether our proposed crawler produces the same output as the default version.

There are two important issues that need to be addressed before we use the stateflow graph for shaping our oracle. First, we should beware that we are implicitly assuming the default version of the crawler is deterministic. Deterministic in this context means if we crawl the same Web application repeatedly, the same stateflow graphs are built as the result of the crawling sessions. In addition, we need to have a Web application that its state transitions remain the same over the course of multiple crawling sessions. In other words, the Web application must not behave randomly, must not have time-based behavior or any other types of behavior that leads to different stateflow graphs over multiple crawling sessions.

The first step in this experiment is to either find or build a deterministic Web application for being used as objects of the correctness experiments. Building one such web application seemed to be more feasible, if not the only option, because we still cannot use the crawler for finding deterministic Websites. The reason is that we do not know if the crawler is deterministic yet. As such, we built a Web application that has multiple clickables. Firing click events on these clickables causes the state of the application (the DOM-tree) transition to numerous states. The key point here is that the application is designed in a way that clicking on a specific sequence of elements will always result in the same sequence of state transitions. This Web application is assured to work deterministically being tested thoroughly and manually. Having a simple and manageable design, the application is completely deterministic.

Having confidence on the determinism of the object application, we need to investigate if the crawler has a deterministic behavior too. Determinism for the crawlers means that, given the Web application is deterministic, the partial stateflow graph created by the crawler is identical over multiple crawl sessions. This means if we set a limit on the total number of states crawled,

and crawl the Web application twice, the resulting stateflow graphs, SFG_1 and SFG_2 must have the following properties:

- The number of states found in both stateflow graphs, SFG_1 and SFG_2, must be equal.
- All the states present in SFG_1 must be present in SFG_2 too and vice versa.
- Given there is a transition T_1 between States S_1 and S_2 in SFG_1, there must be a transition T_2 in SFG_2 which joins counter parts of S_1 and S_2.

By partial stateflow graph we mean the states of the Web application is not exhaustively discovered to a point that no further state is detectable in the Web application.

In order to assess if these qualities hold in the crawler, we conduct an experiment to answer the following research question:

RQ2.0 : Is the default version of the crawler deterministic in crawling Web applications?

In order to answer this question, we crawl the object application we developed ten times and compare the stateflow graphs to see if they hold the above qualities. The comparisons result shows that the stateflow graphs have the above criteria. This means we can rely on the deterministic nature of the crawler to form our correctness oracle.

Having discussed the determinism of the crawler, we proceed with investigating if the graph database-backed solution works correctly. To assess the correctness of the proposed solution, we crawl the deterministic Web application with both the default crawler and the crawler enhanced by our solution. Afterward, we compare the two state-flow graphs based on the properties listed above.

In order to further investigate the correctness of the proposed solution, we do not limit the experiments to the deterministic Web application developed by ourselves. Having ensured the default crawler is deterministic, we use it to find more deterministic Web applications. The solution we came up with for finding additional deterministic Web applications is that we crawl a Web application multiple times with the default browser and compare the stateflow graphs. If the stateflow graphs were equal then we deduce the application is deterministic and can be used in correctness experiments.

We started running the determinism test on top Alexa Websites to check if we could find objects for our experiments. However, these most visited Web applications are highly complex and we found none of the top ten were deterministic. We also tested other simple applications that we expected they might be deterministic but still none of them were. What we did we started crawling random Websites hoping that we could find some deterministic Web applications. Fortunately we found a number of Web applications that they were deterministic. So we used them as objects of the correctness experiments. The application that found to be deterministic are listed in the table 6.3 and the Websites crawled and were not deterministic are listed in the appendix A .

Table 6.3: Correctness experiments objects

States Crawled	Web Application
50	www.al-awda.org
50	www.martinkrenn.net
50	www.thething.it
50	www.c-level.org
50	www.rprogress.org/index.htm
50	www.math.mcgill.ca det
1	www.isc.org/downloads/BIND
1	www.hunyyoung.com
2	home.planet.nl/mooij321
1	www.antique-hangups.com
3	www.project451.com
1	lmuwnmd.wpengine.com/...

Scalability Experiments As we finished the correctness experiments, having assured that the new solution crawls Web applications in the same manner as the default crawler and produces the same output, we proceeded with the memory performance experiments. However, the first results showed that our solution was not able to outperform the default version in terms of the number of states it crawled. We concluded that it is either a sign of inefficient memory consumption in our solution, or an indicator of too huge a memory overhead imposed by the solution .

As a result, we conducted a comprehensive memory analysis on the de-

fault crawler and the new solution to understand how exactly the memory is consumed by different components of the crawler and finding inefficiencies in them. This memory analysis gave us insight into the internal behavior of the crawler and helped resolve the inefficiencies. The memory analysis is presented after the experiments results in section 6.7.

Conducting the scalability experiments, we investigated whether the new version could improve the number of states crawled compared to the default version of the crawler as stated in the research question RQ2.b. To this aim, we crawled a number of top Alexa Web sites by the proposed crawler and the default crawler to compare memory utilization in the two versions of the crawler. The important variables and facts about the experiments are as follow.

- Top 10 Alexa Websites are crawled as the objects of our scalability experiments.
- We limit the memory by setting the maximum Java heap size to one Gigabyte.
- The most important criteria for comparison is the number of states crawled given the limitation on the memory usage.
- For each crawling experiment the default version and the enhanced crawler are run by the same configurations.
- The maximum number of states is set to be unlimited so the crawler crawls till it uses up all the memory, given the Web application has enough states.
- DOM size for each state is measured and recorded.

To evaluate the memory performance of the proposed solution we crawled ten most popular Websites from the Alexa top sites. With each versions of the crawler, we crawled each Web application five times and then took the average of the results over the five crawling sessions. In each experiment we collected the average DOM size of the states in characters. Most importantly the number of states crawled up to the very point before throwing the out-of-memory exception was recorded in the experiments. The objects of the scalability experiment are presented in table 6.4.

As mentioned in the background, there is always a limitation on the amount of memory available to Java heap, be it a start up option or the maximum capacity of the system on which it is running. We, however, set

Table 6.4: Scalability experiments objects

ID	Web Application
ggle	www.google.com
wkpd	www.wikipedia.com
live	www.live.com
twtr	www.twitter.com
qq	www.qq.com
amzn	www.amazom.com
lnkn	www.linkedin.com
baid	www.baidu.com
fb	www.facebook.com
yaho	www.yahoo.com

this limitation to one Gigabyte. We selected 1 Gigabyte as the upper limit because in a system used by developers for programming, such as the system we have been using, 1 Gigabyte is almost the maximum amount that can be allocated to the Java heap while being able to run other tasks, e.g. the experiments in our case, smoothly without having the system freeze. The experiments end when the crawler allocates all the Java heap memory and throws an out of memory error because there is no more memory left to allocate to the requests of the crawling process.

We had various options for measuring memory consumption in the experiments. We utilized VisualVM and Yourkit Java profilers for monitoring the memory consumption of the two versions of the crawler. In section 6.6.3 we discuss the results of the experiments.

Time Performance Experiments As mentioned in the explanation of RQ2.c, time performance in this context is assessed on a holistic approach and is based on time required to crawl Web applications by the two versions of the crawler. In particular, we aim at investigating which version requires less time to crawl a certain number of states in the object Web applications. There are a number of important differences between these two versions of the crawler that are determining in the result of the experiments. In particular we are interested in investigating the counter effects of the following characteristics.

First, the scalable version utilizes a database that resides in a secondary

memory (hard disk). This means the database requires to perform a considerable number of I/O requests and as a result it might consume significant time while storing and retrieving data. On the other hand, and more interestingly, the database does not have to lock the stateflow graph to add or update the data in contrast to the default version which locks the stateflow graph and as a consequence, threads cannot update the model simultaneously in the default version.

The objects of these experiments are the same as the scalability listed in the table 6.4. We crawl each of these Web applications five times with both crawlers and measure the time. The maximum number of states is set to 100 and the memory of the Java heap is set to a maximum of one Gigabyte. As mentioned in the scalability experiment, there is always a maximum for the Java heap. If we had not set the maximum manually it would have been the default amount for the maximum. Hence, given the fact that all objects were crawlable up to 100 states within utilizing 1 Gigabyte of memory, we opted to set this upper limit which also goes well with scalability experiments too.

Having covered the design and methodology of the experiments, we present the results of the experiments in the following section.

6.6.3 Results

The results of the experiments are presented here to answer the three research questions that were explained in the previous sections. These research questions targeted the correctness, scalability, and time performance of the proposed solution.

RQ2.A: Correctness We conducted the correctness experiments on the the Web application listed in table 6.3. In addition, we ran the experiment on the deterministic Web application that we developed as well.

The results show that the new solution works correctly in crawling all of the Web applications crawled. Hence, the answer to RQ2.a is positive.

The scalable crawler produces completely the same output as the default crawler. In fact, the first correctness experiment that we ran was not positive but that led us to find the bugs and fixed them to achieve a point that the results were positive for all Web applications.

RQ2.B Scalability The gist of the results of the scalability experiments are presented in the bar charts of figure 6.2 and 6.3. As it is presented in

figure 6.2, the result show in all experiments the proposed solution performs more efficient and demonstrates more scalability than the default version. In addition, as it is shown in figure 6.3, the proposed crawler achieves improvements in all experiments with an overall average of 88.16%. Hence, the answer to question RQ2.b is:

The proposed solution improves the scalability of crawling process by 88.16% on average.

Table 6.7 shows how the new solution outperforms the default version while tested on different Web application objects. This table shows the details of the results.

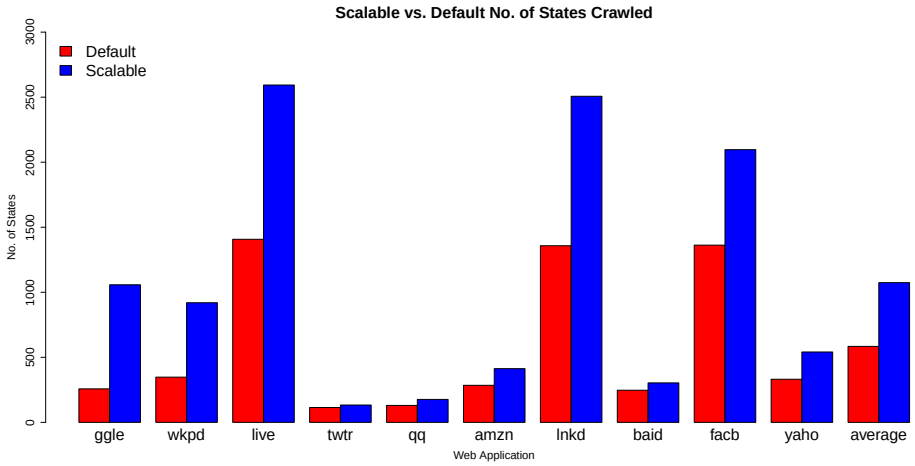

Figure 6.2: Scalable vs. default no. of states crawled: in all cases the proposed version overtakes the default crawler in terms of the number of states crawled given a limited amount of heap memory.

RQ2.C: Time Performance As presented in table 6.6, the default crawler performs more efficiently than the proposed solution. This is, in fact, no surprise to our expectations. As discussed in 6.5.2, the time performance trade offs prepares for the fact that introducing a slower component such as secondary memory to the crawling process can slow down the system. Moreover, compared to the scalability achieved by the solution, the time overhead is not too intolerable. Hence, the answer to question is: although

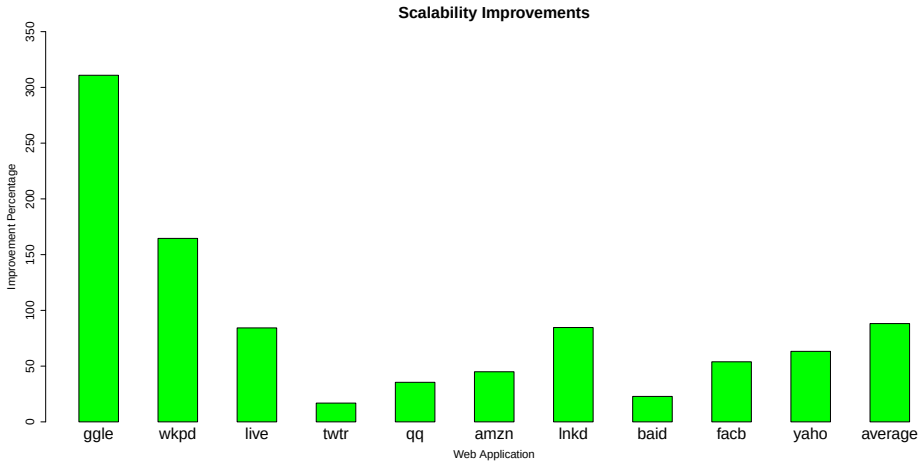

Figure 6.3: Scalability improvements: the percentage of increase in the number of states crawled.

the proposed solution outperforms the default crawler in two cases, and in six out of ten cases the time overhead is less than 10%, on average the time performance of the default crawling technique sounds more promising.

The proposed solution imposes a time overhead which is 18.20% on average and if Facebook.com is excluded from the results, the rest of the experiments show a time overhead of 6.13%.

6.7 Further Memory Analysis

As mentioned earlier we performed a comprehensive memory analysis on both the default crawler and the enhanced version. We performed this analysis to achieve more insight on the way memory is utilized in the heap so as to crawl a greater number of states by means of improving the memory performance.

The initial goal of this memory analysis was to discover the inefficiencies of the proposed version so as to achieve more scalability. This lead to finding two memory leak points and patching them. However, after finding the memory leak points and achieving the scalability, we continued the analysis in more depth so it can be utilized in future memory optimizations.

Table 6.5: Scalability experiments results: the proposed version outperforms the default crawler in terms of the number of states crawled.

ID	Default States	Scalable States	Improvement (%)
ggle	257.4	1057.6	310.88
wkpd	347.6	919.6	164.56
live	1407.4	2593.4	84.27
twtr	114	133.2	16.84
qq	130.2	176.4	35.48
amzn	284.8	412.8	44.94
lnkd	1357.8	2506.8	84.62
baid	247	303.4	22.83
facb	1362.4	2096.4	53.88
yaho	331.4	541	63.25
average	584	1074.06	88.16

We have considered two main approaches to compare the efficiency of memory consumption of the two versions of the crawling algorithm. First, we can limit the amount of memory available to both crawlers and measure the number of states crawled before consuming all memory. The other option is to limit the number of states and measure the amount of memory consumed in the crawling. Both of these methods have some advantages and disadvantages.

As the aim of this work was to increase the number of states crawled given a limited amount of memory (e.g. one Gigabyte), measuring the number of states is a more intuitive approach for carrying out the experiments. However, measuring the memory gives us more insight on what internally is taking place and helps us understand if the memory is used efficiently. Moreover, measuring the memory consumption can pinpoint pointers for finding inefficient parts of the crawler and fixing them. Hence we use both methods to achieve a deeper insight over the memory consumption characteristics of the crawlers.

In order to compare the scalability of the two versions of the crawler we started off by carrying out an experiment with the specifications listed in table 6.7.

What we measured here was the number of states crawled given the limited amount of memory. None of the crawlers exceeded the time limit. In fact, the time limit was deliberately set to 10 hours as by experience

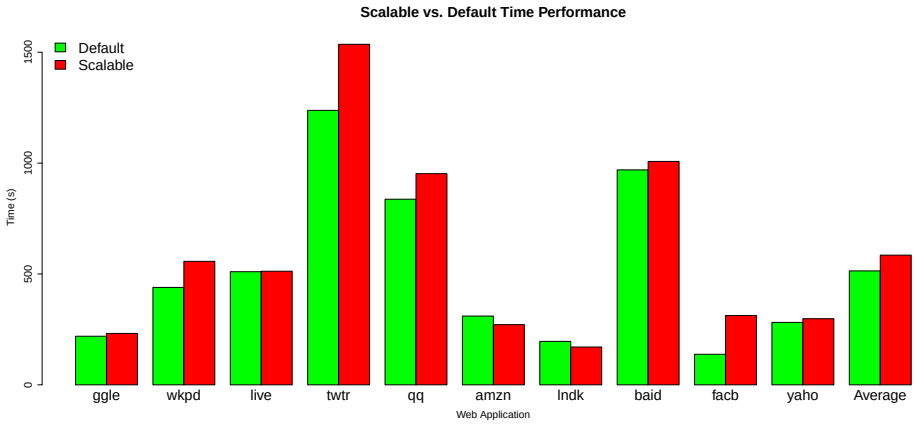

Figure 6.4: Scalable vs. default time performance: the proposed version overtakes the default crawler only in two cases out of ten and in six cases time overhead is less than 10 %. However, on average the time overhead is 18.20%.

we knew that this amount of time is enough for both crawlers to exceed the memory limit. The results however, to our surprise, showed that the proposed solution did not outperform the default crawler in terms of the number of states crawled. The default version crawled 533 states while the proposed solution crawled only 378 states.

Availing from the second approach, we carried out another experiment by profiling the crawlers using VisualVM profiling tool. We set up a second experiment with specifications outlined in table 6.8.

The result of the experiment, however, shows that the default version is consuming less memory than the enhanced version which was surprising for us. The memory consumption in the default version shown in figure 6.5 reaches a maximum of 78 Megabytes while the enhanced crawler exceeds a memory consumption of 197 Megabytes which is shown in figure 6.6. As a result, we repeated the same experiment but changed the object of the experiment, by crawling www.yahoo.com instead of the previous URL. However, the results are not still promising. We observe, as shown in the figures 6.7 and 6.8, a spike at the early stages of crawling for both crawlers. Afterward, non of the crawlers shows significant improvement over the other version.

By observing the memory consumption graphs of VisualVM and the

Table 6.6: Time performance experiments results: the default crawler performs more efficiently on most of the cases. However, in two cases the proposed solution overtakes the default crawler and in six out of ten cases the time overhead is less than 10%.

ID	Default Time (s)	Scalable Time (s)	Overhead (%)
ggle	219.2	231.8	5.75
wkpd	439.4	557	26.76
live	510.2	512.4	0.43
twtr	1237.8	1535.8	24.07
qq	837.2	952.4	13.76
amzn	310.2	271.4	-12.51
lndk	196	170.6	-12.96
baid	969.4	1007.6	3.94
facb	137.8	312.6	126.85
yaho	281.6	298.2	5.89
Average	513.88	584.98	18.20
Avg Excl. facb	555.67	615.24	6.13

log output of the crawlers, we come to conclusion that we need to revise the experiments. First, observing considerable spikes at the beginning of the crawling proves that 100 state is not a justifiable limit for the number of states because the memory required for the storage of the states is not considerable enough to make the maximum happen in the end of the crawling rather than the beginning. In addition, the graphs produced by VisualVM do not distinguish between old generation and young generation. Hence we considered utilizing a more powerful tool providing us with more information about the memory utilization.

In fact, automatic garbage collection in Java, which cleans the unused memory in a relatively complicated way, makes memory measurement experiments more cumbersome. The main reason is that, the frequency of garbage collection is not dependent solely on having unused memory to be cleaned, i.e. the frequency is not constant. The frequency is also dependent on the amount of memory available to be allocated. The less memory available in the heap the more frequent garbage collections occur.

In addition, there are two types of garbage collection, major, and minor garbage collections, each of which affects the used memory measurements

Table 6.7: Initial memory experiment specifications

Aspect	Specification
URL	www.yahoo.com
Maximum Number of States	Unlimited
Maximum Depth	10
Maximum Time	10 Hour
Maximum Memory	1.5 Gigabyte

Table 6.8: Second memory experiment specifications

URL	www.google.com
Maximum Number of States	100
Maximum Depth	10
Maximum Time	10 Hours
Maximum Memory	1.5 Gigabytes
Profiling Tool	VisualVM

differently. There is almost always a difference between the amount of memory allocated and the amount of memory that we expect to be allocated based on our assumptions about the algorithm of a Java program. Hence, measuring memory in specific points of times, while not having confidence whether the garbage collection is performed or not is not the best available indicator of memory utilization.

In the default crawler, as states' data is detained in the memory till the end of the crawling, we expect this data is not garbage collected, neither by major collection nor by minor collection. On the other hand, when running the proposed crawler, we expect the states data be garbage collected once in a while as all references to this data are gone after saving the state data in the database. As such, we need the young generation and old generation measurements to understand if the objects related to the state in the memory is garbage collected or not.

Hence, instead of comparing memory at specific points of time, we need to have old and young generation data, and observe their trends. What we expect to observe is that, the states information is freed in the enhanced crawler by comparing the trends of the old generation in two crawlers. So

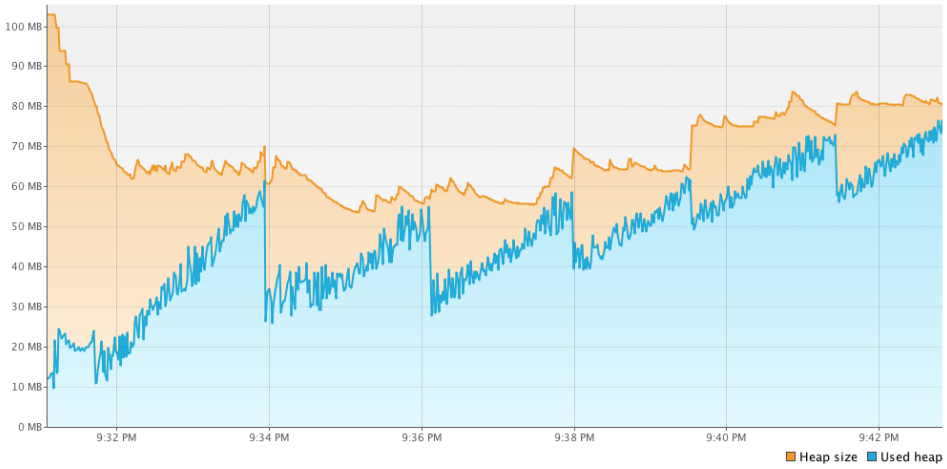

Figure 6.5: Memory consumption of the default version: the experiment is carried out on google.com and the memory consumption is demonstrated.

we profile both crawlers with Yourkit Java profiler to achieved a better understanding of the memory consumption in both crawlers.

First of all, we found two leak points in the proposed crawler. In fact what held us from achieving more scalability was that as the states were being added to the the stateflow graph one by one, they were also being referenced in a another sub-component of the crawler that was responsible for managing the paths of the crawling. By fixing leak points in those parts we achieved the scalable results presented in the scalability experiment results.

Utilizing Yourkit profiler, we took snapshots of the memory usage of the default crawler and discovered the information summarized as follows:

- State machine method in which the new states are created (newStateFor) amounts for 52 percent of the memory allocated to the objects.
- Inspecting the new DOM amounts for 45 percent of the memory allocated to objects.
- 99 percent of the objects created by the newStateFor are state objects.
- Objects allocate by the inspectNewDOM method are mostly DOM related objects such as HTML element, HTML anchor element, HTML script element etc.

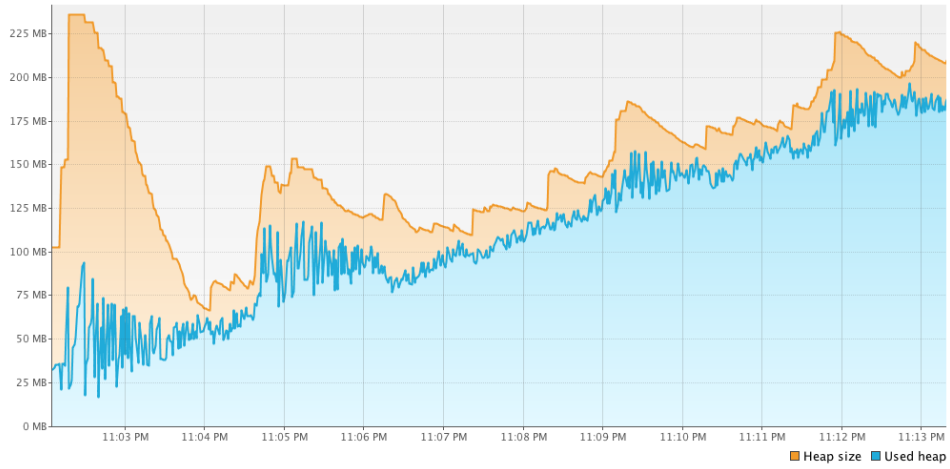

Figure 6.6: Memory consumption of the proposed version: the experiment is carried out on Google.com and the memory consumption is demonstrated.

We conducted the same analysis on the snapshot taken from the scalable version and the information is summarized in the following:

- State machine method in which the new states are created (newStateFor) amounts for 52 percent of the memory allocated to the objects.
- Inspecting the new DOM amounts for 64 percent of the memory allocated to objects.
- Similar to the default version, Objects allocated by the inspectNewDOM method are mostly DOM related objects such as HTML element, HTML anchor element, HTML script element etc.
- The method follow(), which brings the browser from index to the state that must be explored, allocates 24 percent of the memory to objects mostly used for IO.
- States object retain only 2 percent of the memory
- The candidate actions amount for 69 percent of the memory

We performed seven different memory inefficiency inspections on the both crawlers to diagnose potential memory inefficiencies.

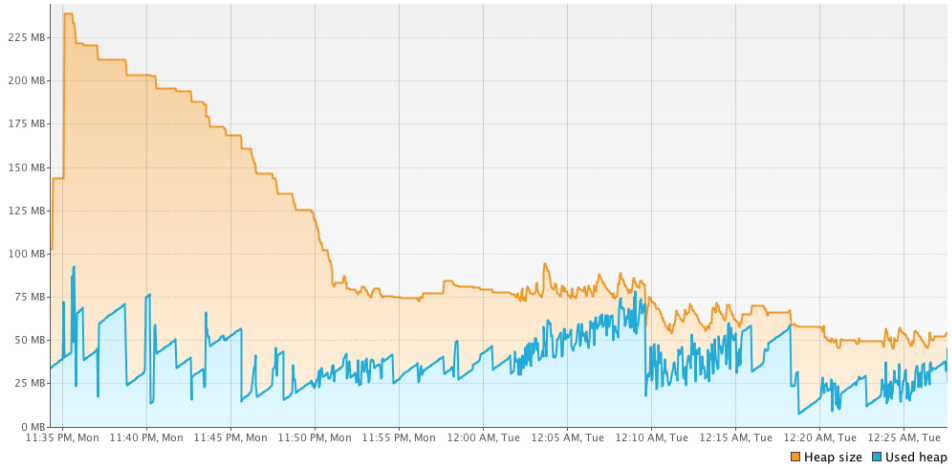

Figure 6.7: Memory consumption of the default version: the experiment is carried out on Yahoo.com and the memory consumption is demonstrated.

The plot of memory utilization by two versions are provided in figures 6.9 and 6.10

As it was expected the long generation for the default crawler is ascending because states are not freed and retained in the memory. However, for the proposed solution the freeing of states objects makes the graph descending at times.

As presented in the tables and figures of this chapter, the proposed solution works correctly in crawling all object Web applications. In addition, the proposed solution improves the scalability of crawling process by 88.16%. This improvement is achieved on a time overhead cost. The proposed crawler imposes a time overhead which is 18.20% on average and if Facebook.com is not considered, the rest of the experiments suggests a time overhead of 6.13%.

Having illustrated our improvement ideas on the scalability of the crawler and presented the the results of the experiments, we revisit the questions and the answers obtained for them in this work in the next chapter.

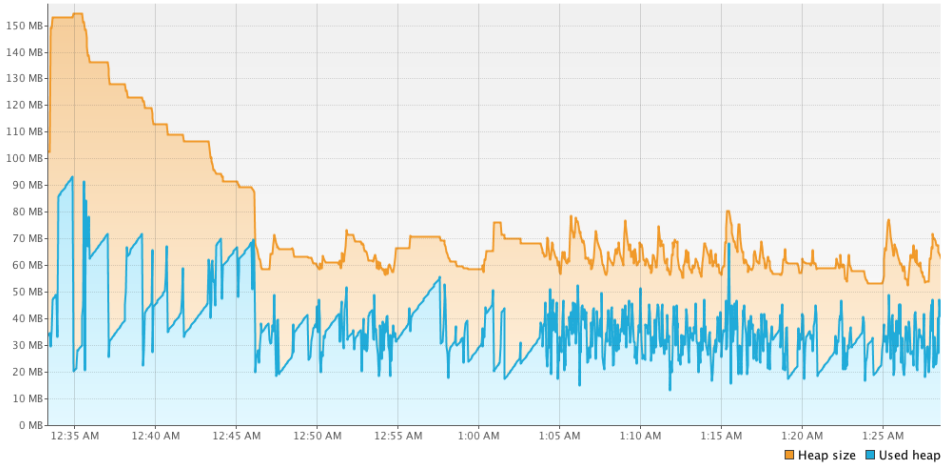

Figure 6.8: Memory consumption of the proposed version: the experiment is carried out on Yahoo.com and the memory consumption is demonstrated.

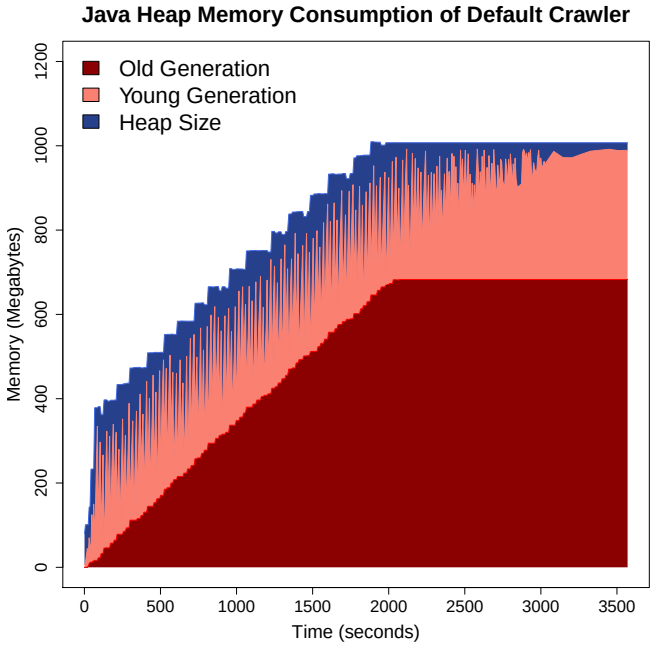

Figure 6.9: Default memory consumption: the lowest curve (in red) shows old generation allocation. On top of that, the young generation (in salmon) is presented. The highest curve (in blue) shows the maximum amount of memory available to be allocated, i.e. Java heap size. The important observation here is that the old generation allocation never drops as all the states are detained in memory. In this experiment Java maximum heap size is set to 1 gigabyte and 330 states are crawled.

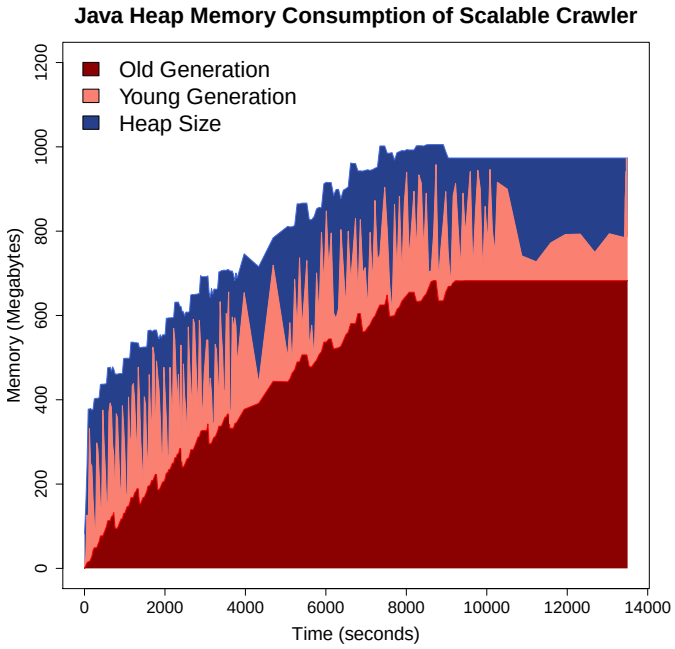

Figure 6.10: Improved memory consumption: the lowest curve (in red) shows old generation allocation. On top of that young generation is presented (in salmon). The highest curve (in blue) shows the maximum Java heap memory available to be allocated. One of the results of our work is that the old generation allocation occasionally drops, i.e. states data is freed. This enables the crawler to discover 542 states using 1 gigabyte of memory.

Chapter 7

Discussion

Having presented the proposed enhancements and their evaluations in the last two chapters, in this chapter we revisit the research questions and discuss the findings in more details. We review the findings of the state transition management work first and then continue the section with dissertating the answers to the research questions of the scalability work. Furthermore, we discuss the "threats to validity" of the work as well. Finally the ideas emerged from this thesis for the direction of future work finalizes this chapter.

7.1 State Transition Management

Time performance improvement was the target of the enhancements we put forward for enhancing the state transition management of the crawling process. In order to achieve the improvement, we brought forth and developed three alternative methods for managing the transition of states in the crawling process. The results of evaluation of these methods, revised incrementally one after another, are discussed in this section.

7.1.1 RQ1.A: Time Performance of the Proxy-Based Solution

The goal of the experiments on the proxy-based solutions was to examine to what extent this method is able to improve the time efficiency of state transition management. As it is shown in figure 5.5, the time overhead the proposed enhancement imposes on the crawling process is significantly greater than the time improvement it brings about. In eight cases out of ten, the default crawling process performs better than the proposed proxy based. Moreover, in seven cases out of of the aforementioned eight cases, the default crawler consumes significantly less time than the proposed solution. The prospered solution overtakes the default crawling process only in two cases and the performance difference in these two cases are not as considerable compared to difference in the cases where it is overtaken by the default one.

All in all, as it is shown in the average bar, the default version performs better than the proposed one. Having performed this experiment, we now have a better insight on the effects of utilizing a proxy for time optimization, in the crawling process. In fact, the findings of this part of the work helped guide the rest of the study to a great extent. For example, we had planned to form two other projects for which proxy utilization would have been a pivotal issue.

7.1.2 RQ1.B: Time Performance of the First Plugin-Based Solution

Similar to the RQ1.a, the goal here was to assess the time performance but for the second solution we proposed. Having revised this solution based on the lessons learned from the development and evaluation of the previous solution, we assessed the time performance of the solution by the same set of experiments as that of the proxy-based solution.

As it is shown in the bar-chart of figure 5.6, the first plugin-based solution, improves the time performance of the state transition management to a great extant. The proposed solution overtakes the default version in 6 cases and in the rest of the cases the difference is not very significant. The proposed solution, on average, improves the time consumption of the transition management process by 253.34%.

7.1.3 RQ1.C: Time Performance of the Second Plugin-Based Solution

The time performance of this version of the crawler was assessed in the very same way as its two predecessors. As it is illustrated in figure 5.7, the third solution improves the time performance of the state transition management of the process. Although this version of the crawling process was theoretically more improved than the second solution, the results show that this version, on average, improves the time performance of the transition management process by 197.48% which is less than the second solution. However, this version also significantly overtakes the default version.

7.2 Scalability

In the scalability part of this work, we developed the idea of creating an alternative solution for storage of web application states' data. The alternative solution employs a graph database for storing the data to let the main

memory be available to the rest of the tasks in the crawling process. We devises three research question targeting the correctness memory and time performances of the crawling process. In the process of carrying out the work further questions emerged which we seized the opportunity to examine them.

7.2.1 RQ2.0: Determinism of the Crawler

In the process of designing the experiments for pursuing answers to RQ2.a which targeted the correctness of the proposed solution, we faced the challenge of developing an oracle for examining the correctness of the solution. As we opted to utilize the default solution in shaping the oracle, we had to figure out if the crawler behave deterministically. Hence, we formed RQ2.0 and carrying out the experiments, the result showed that the default crawler is completely deterministic.

7.2.2 RQ2.A: Correctness

Having assured the determinism of the default crawler, we built and harvested a number of object web applications and assessed the correctness of the proposed solution. The oracle used for testing the correctness of the solution was formed over the stateflow graph that the crawler builds out of crawling a Web application. The results of the experiments show that the proposes solution crawled all the object Web applications completely correctly. It discovered the same stateflow graphs as the default crawler.

7.2.3 RQ2.B: Memory Performance

Having obtained full confidence on the correctness of the proposed solution from the experiments results, we continued with the memory performance part of the work. RQ2.b investigates to what extent the proposed crawler increases the number of states crawled. The answer to this question is illustrated in figures 6.2 and 6.3. As it is shown in the figures, the proposes solution is able to outperform the default crawler in all instances and achieves a greater state coverage. The proposed solution, on average, improves the scalability by 88.16%.

The scalability improvements achieved here can potentially be useful if the crawler will be deployed as a cloud computing service (e.g. on Amazon Web Services). Especially, supporting partial locks on the stateflow graph, the proposed crawler can help implement an efficient concurrent deployment of the crawler. On the contrary, the default version of the crawler needs to

lock the whole stateflow graph when it requires to add states or transitions to the data structure.

7.2.4 RQ2.C: Time Performance

In order to improve the scalability of the crawling process, we opted for creating an alternative solution utilizing a graph database. As mentioned in the discussion of the trade-offs, we investigated the effects of the discussed counter-effects. As the results of the experiments show, the default version of the crawler outperforms the proposed solution in most of experiments. However, in two out of ten instances the proposed solution overtakes the default version and in five out of ten cases the time overhead imposed by the proposed solution is less than 10%.

Having provided an overview over the research questions and the answers obtained by the experiments, we move on with discussing the threats to validity concerns.

7.3 Threats to Validity

This threat to validity component sheds more light on some of the concerns we have had during this work. We provide details on how we endeavored to mitigate them. In this section we address internal and external threats to validity of this work.

7.3.1 Internal Validity

In the first part of this work, the state transition management improvements, we encountered a substantial obstacle. The top Alexa Websites are highly complex Web applications and they are not deterministic by any means. In order, to mitigate the determinism problem of the experiments, we decided on running a greater number of experiments on each object and then take average over results. However, after carrying out the experiments, we observed the state transition management part of the crawling process amounts for a tiny portion of the total crawling time. As a result, we strove on finding a method that could compare the two versions of the crawler on the same circumstances. Hence, we put forward the idea of having both versions of the state transition management plugged into the crawler at the same time. This way, when checking whether the application has transitioned to a new state or not, we first use the mutation-summary library to inquire about changes made to the DOM. Afterward, regardless of the report provided by

our agent, we perform the default comparison behavior. Now, if the mutation summary reports of no change and the comparison also reaffirms it, the time spent for the default comparison is regarded as the time that would have been saved if we had plugged in only the proposed solution. We measure the time each of the solutions consumes on performing its tasks and then compare them bases on these times.

7.3.2 External Validity

Regarding the external validity of this work, the most important concern here is how well the conclusion can be generalized. Firstly, the Web applications selected as the objects of the experiments can highly affect the final results. The set of all Web applications on the Internet is too large to find a fairly acceptable representative subset for it. Hence, to mitigate the systematic error, i.e. the bias in selecting object Web applications, we opted to choose the most visited Websites from Alexa top list.

In addition, configurations of the crawler for running the experiments provide us with a wide set of options; for example, the types of HTML elements chosen to be clicked on during the crawl can be pivotal to the final results of the experiments. In order to mitigate this bias, we cared for selecting the configurations that are most natural, general and meaningful for crawling a Web application for testing, comprehension or analysis presupposes.

Having discussed the threats to validity of this work, we move on with outlining the directions this research for future work.

7.4 Future Work

Throughout performing multiple projects that shaped this thesis, we achieved a greater insight on the area of crawling modern Web applications. From the experiences learned in this thesis, we bring forth three ideas to move this work to the next level. These ideas, from a higher level perspective, lie in the areas of improving the time performance and further improving the scalability of the crawling process.

7.4.1 Further Scalability through String Optimization

One of our presumptions about the crawling process was that memory allocated to states' data is the only obstacle in achieving absolute scalability. That is, we assumed if the memory is not used for states storage, we can

endlessly resume crawling. However, from performing the memory analysis, we discovered two important areas of limitation. First, one of the diagnostic tests for the memory consumption of the crawler revealed that a very considerable amount of memory is waisted by having multiple instances of the same string in the memory. Further investigation showed that these strings originate predominantly from the HTML attributes and elements extracted form the DOM-tree. Possible solutions that we can suggest is to employ string interning techniques. The second area of memory optimization follows in the next section.

7.4.2 Further Scalability through Candidate Elements Optimization

Performing the memory analysis, we discovered that storage of candidate clickable elements is the number one cause for consuming up the memory resources. As such, we think the next area which is worth investigating to improve the scalability, is developing alternative methods for storage of candidate clickables. This can be carried out either by utilizing a graph database, or changing the order of storage of the elements. That is, it is possible to change the logic so there is no need for storing all the candidate clickables of all the stored states in the memory.

7.4.3 Scalability Improvement by Text Compression Techniques

As string objects containing HTML elements consume a significant amount of memory in crawling Web applications, we believe that utilizing text compression techniques has the potential to improve the scalability of the crawler. Especially, large DOM strings of the same application may share a considerable portion of content which brings about memory optimization opportunities.

7.4.4 Improving Time Performance Targeting

RQ1 targeted the time performance of the state transition management of the crawling process. As presented in the result, the improvements achieved in the second and the third solutions were very significant. In addition, the lessons learned from the proxy-based solution were also very instructional in guiding the directions of the rest of the project, specially which directions not to pursue. However, we also learned from the second part of the project, during the memory analysis, that utilizing a tool such as YourKit profiler,

we might be able to find more significant areas for optimization. That is, we are not sure, but it is possible that among manifold areas of optimization, pinpointed first by our insight over the crawling process, a preliminary comparison over the significance of the selected areas, results in more significant achievements.

Chapter 8

Related Work

Web crawlers have been a major area of research [22–34] since the invention of the Web. Optimizing Web crawlers, as a consequence, has been an attractive research topic too [35–37]. Here we discuss a number of studies in the literature focusing on optimizing the performance of different Web crawlers. In addition, as we performed a comparison on graph databases, we enumerate some other studies comparing graph databases too.

8.1 Optimizing the Crawling Process

Jourdan et al. propose a strategy for efficient crawling of modern Web applications (which they call rich Internet applications) [38]. The proposed method creates a model for predicting the behavior of modern Web applications, and based on this model produces an execution plan that is optimized for finding new states in the shortest amount of time. Afterward the model is updated based on the discovered states if the new states are not compatible with the previous model. The authors compare their techniques with other modern Web crawlers and depth-first and breadth-first crawling strategies.

In another study [39], Jourdan et al. improve their previous work [38] by utilizing statistics gathered during a crawl session to predict optimized strategies for selecting the next part of the application to be crawled. The authors compare the performance of their proposed strategy with breadth-first strategy, depth-first strategy and their previous crawling strategy.

In 1999, Heydon et al. [35] investigated the inefficiencies in their Java Web crawler called Mercator. In particular they found that they have to refactor their crawler to make their code more aligned with regards to the characteristics of Java Runtime Environment such as the mechanism memory is allocated and deallocated in Java Heap and excessive synchronization. They investigated the synchronization, memory management and other aspects of the Java and achieved performance improvement in the crawler. In another paper [36], they explain the Web crawler implemented in Java, by enumerating the components of web crawlers and explaining the alternatives and trade-offs of different approaches. They compare the performance

of their crawler with that of Google^{TM}in terms of rate of downloading documents, the volume of data downloaded, and HTTP request sent.

Another study in the area of optimizing the crawling process in web applications was done by Edwards et al. [37] at IBM research center. This study focuses on optimizing the Webfountain web crawler. The Webfountain is an incremental web crawler, i.e. the crawler updates the versions of the websites it has already crawled when they are updated.

Edwards' paper has a target that is relatively different from the aim of this thesis. They aim to optimize various competing criteria of crawling Web, such as freshness, in Webfountain. They propose an adaptive model that optimizes managing the URL queue and internal variables of crawling in Webfountain. They utilize simulation and computational models to evaluate their work, but hope to evaluate it on real data once Webfountain is operational.

8.2 Graph Database Applications

In this thesis we compared many graph databases to select the graph database most suited to our applications. Graph databases have been either the target of research or been utilized in many research projects [40–48]. We found research focusing on comparing different aspects of graph databases [49–52]. A paper which is to some extent similar to our comparison is performed on the comparison of models of graph databases; in this paper, Angels et al. [53] investigate the area of modeling graph databases in a survey. They focus on the various aspects of modeling such as query languages, data structures, and integrity constraints. We have availed of these studies, in performing a comparison of our own, for choosing the best database suited to our application.

Chapter 9

Conclusion

Aiming at improving time consumption and scalability of crawling modern Web applications, we proposed and evaluated our enhancements ideas in this thesis. Availing of efficient DOM monitoring techniques, we focused on providing three alternative solutions for reducing time consumption of the "state transition management" sub-task of the crawling process. The main idea for improving the time consumption was to bypass unnecessary steps of the state transition management by efficient tracking of alterations made to the DOM. In addition, utilizing a graph database for storage and retrieval of dynamic Web states, we aimed at increasing the number of states crawled by freeing the memory detained by states during a crawling session.

We used Crawljax, a modern Web crawler, as the platform for actualizing our ideas into concrete implementations so as to evaluate them. As such, we reverse engineered Crawljax to gain insight on the default crawling process and applying our ideas in terms of enhancements to Crawljax.

In order to improve the time consumption of the state transition management, we developed three alternative solutions that were revised incrementally one after another. In all three solutions, an agent for tracking DOM mutations was developed to be installed on browser side of Web applications. The first solution, utilized a proxy for installing the monitoring agent and a Web server for hosting the agent. The proxy was replaced by Crawljax plugins in the second solution but the Web server was retained for hosting the agent. Finally, in the third solution, agent installation was carried out by Crawljax plugins in the same manner as the second solution but the Web server was eliminated from the design and the agent was incorporated in the core of the solution.

The crawling process in the crawler captures dynamic state changes and navigational paths between the states to form a finite-state machine model called stateflow graph. Aiming at reducing main memory consumption to increase the number of states crawled, we proposed the idea of providing an alternative solution for storing the stateflow graph data structure in a graph database on secondary memory. We conducted a comparison over graph databases in which Neo4j emerged to be the graph database most suited

for our application. We implemented a scalable solution utilizing the graph database to free memory from states' data and provide the rest of crawling tasks with freed memory to continue the crawling to cover a greater number of states.

The proposed solutions were evaluated by comparing them with the default crawler. Five rounds of experiments were carried out on ten Web applications selected form Alex most visited websites. In experiments on state transition management, maximum number of states was set to 10 and the time consumptions of the default crawling process and each enhanced version were measured simultaneously. In the state coverage improvement, the scalable crawler output (stateflow graph) was compared with the output of the default version to ascertain the correctness of the proposed solution. The maximum amount of Java heap memory was set to one Gigabyte and the number of states crawled was measured to evaluate the scalability of the proposed solution. To assess the scalable solution in terms of time consumption, maximum number of states was set to 100 and the total time consumptions of the scalable solution and the default crawler were measured.

The results of the experiments showed that proxy-based solution was not able to reduce the time consumption of the "state transition management". Having learned that proxy imposed a significant time overhead on the crawling process, we changed the direction of the thesis. The second solution, revised based on the results of the proxy-based solution, was able to improve the time performance of the state transition management , on average 253.34%; that means time consumption ratio of default to proposed is 3.53. For the third solution, the ratio of time consumption of default version to time consumption of proposed solution is 2.9748 which means 197.48% improvement in time performance.

Comparing stateflow graphs created by scalable and default crawlers upon crawling deterministic web applications, we observed the scalable solution crawled correctly in the exact same manner as the default crawler. The proposed solution improved the scalability by increasing the number of states crawled, on average, by 88.16%. However, this scalability improvement was achieved costing a time overhead of 18.20%.

A memory analysis was conducted both on the scalable solution and the default solution. The main outcomes of this memory analysis were finding two memory leak points in the scalable solution, resolving which led to the achieved state coverage improvements; and identifying scalability obstacles to bring forward optimization opportunities for future work.

Finally, as a measure to improve reproducibility of the experiments, in addition to including the object Web applications in the thesis, the imple-

mentation of the scalable solution is accessible as open source software at https://github.com/saltlab/crawljax-graphdb.

Bibliography

[1] J. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.

[2] "Oracle java micro edition embedded client customization guide," 2012. `http://docs.oracle.com/javame/config/cdc/cdc-opt-impl/ojmeec/1.1/custom/html/tuning.htm`, [retrieved: Sep. 25, 2013].

[3] T. Berners-Lee, L. Masinter, M. McCahill, *et al.*, "Uniform resource locators (url)," 1994.

[4] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," vol. 6, p. 3, ACM, 2012.

[5] D. Goodman, M. Morrison, and B. Eich, *Javascript® bible*. John Wiley & Sons, Inc., 2007.

[6] E. Ecma, "262: Ecmascript language specification," *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr*, 1999.

[7] D. Raggett, A. Le Hors, I. Jacobs, *et al.*, "Html 4.01 specification," *W3C recommendation*, vol. 24, 1999.

[8] "world wide web consortium." `http://www.w3.org/`, [retrieved: Sep. 23, 2013].

[9] "The webkit open source project." `http://www.webkit.org/`, [retrieved: Sep. 10, 2013].

[10] "Introducing flockdb." `https://blog.twitter.com/2010/introducing-flockdb`, [retrieved: Apr. 16, 2013].

[11] "Twitter.com." `https://twitter.com/`, [retrieved: Oct. 7, 2013].

[12] "Seleniumhq browser automation tool." `http://www.seleniumhq.org/`, [retrieved: Aug. 12, 2013].

[13] "Web storage w3c recommendation." `http://www.w3.org/TR/webstorage/`, [retrieved: Sep. 18, 2013].

[14] "Mutation events." `https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Events/Mutation_events?redirectlocale=en-`, [retrieved: Nov. 21, 2013].

[15] R. Weinstein, "Dom mutation events replacement: The story so far-existing points of consensus," Dec. 2011. `http://lists.w3.org/Archives/Public/public-webapps/2011JulSep/0779.html`, [retrieved: Nov. 21, 2013].

[16] "Document object model (dom) level 3 events specification." `http://www.w3.org/TR/DOM-Level-3-Events/#events-mutationevents`, [retrieved: Nov. 21, 2013].

[17] "Mutation-summary javascript library." `https://code.google.com/p/mutation-summary/`, [retrieved: Nov. 22, 2013].

[18] "Mutation observers vs. mutation summary." `https://code.google.com/p/mutation-summary/wiki/DOMMutationObservers`, [retrieved: Nov. 22, 2013].

[19] M. Shapiro, "Structure and encapsulation in distributed systems: the proxy principle," in *Proc. 6th Int. Conf. on Distributed Computing Systems(ICDCS)*, pp. 198–204, May 1986.

[20] "Understanding memory management." `http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html`, [retrieved: Oct. 1, 2013].

[21] G. B. Tim Lindholm, Frank Yellin and A. Buckley, *The Java ® Virtual Machine Specification*. 500 Oracle Parkway, Redwood City, California 94065, U.S.A, java se 7 ed., 2013.

[22] Y. Guo, K. Li, K. Zhang, and G. Zhang, "Board forum crawling: a web crawling method for web forum," in *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pp. 745–748, IEEE Computer Society, 2006.

[23] V. Shkapenyuk and T. Suel, "Design and implementation of a high-performance distributed web crawler," in *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 357–368, IEEE, 2002.

[24] J. Cho and H. Garcia-Molina, "Parallel crawlers," in *Proceedings of the 11th international conference on World Wide Web*, pp. 124–135, ACM, 2002.

[25] C. C. Aggarwal, F. Al-Garawi, and P. S. Yu, "Intelligent crawling on the world wide web with arbitrary predicates," in *Proceedings of the 10th international conference on World Wide Web*, pp. 96–105, ACM, 2001.

[26] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," 2000.

[27] J. L. Wolf, M. S. Squillante, P. Yu, J. Sethuraman, and L. Ozsen, "Optimal crawling strategies for web search engines," in *Proceedings of the 11th international conference on World Wide Web*, pp. 136–147, ACM, 2002.

[28] G. Pant, P. Srinivasan, and F. Menczer, "Crawling the web," in *Web Dynamics*, pp. 153–177, Springer, 2004.

[29] M. Ehrig and A. Maedche, "Ontology-focused crawling of web documents," in *Proceedings of the 2003 ACM symposium on Applied computing*, pp. 1174–1178, ACM, 2003.

[30] S. Chakrabarti, B. E. Dom, and M. H. van den Berg, "System and method for focussed web crawling," July 9 2002. US Patent 6,418,433.

[31] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th international conference on World Wide Web*, pp. 141–150, ACM, 2007.

[32] C. Castillo, "Effective web crawling," in *ACM SIGIR Forum*, vol. 39, pp. 55–56, ACM, 2005.

[33] M. Álvarez, A. Pan, J. Raposo, and J. Hidalgo, "Crawling web pages with support for client-side dynamism," in *Advances in Web-Age Information Management*, pp. 252–262, Springer, 2006.

[34] S. Chakrabarti, M. Van den Berg, and B. Dom, "Focused crawling: a new approach to topic-specific web resource discovery," *Computer Networks*, vol. 31, no. 11, pp. 1623–1640, 1999.

[35] A. Heydon and M. Najork, "Performance limitations of the java core libraries," in *Proceedings of the ACM 1999 conference on Java Grande*, pp. 35–41, ACM, 1999.

[36] A. Heydon and M. Najork, "Mercator: A scalable, extensible web crawler," *World Wide Web*, vol. 2, no. 4, pp. 219–229, 1999.

[37] J. Edwards, K. McCurley, and J. Tomlin, "An adaptive model for optimizing performance of an incremental web crawler," in *Proceedings of the 10th international conference on World Wide Web*, pp. 106–113, ACM, 2001.

[38] K. Benjamin, G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut, *A strategy for efficient crawling of rich internet applications*. Springer, 2011.

[39] M. E. Dincturk, S. Choudhary, G. Von Bochmann, G.-V. Jourdan, and I. V. Onut, "A statistical approach for efficient crawling of rich internet applications," in *Web Engineering*, pp. 362–369, Springer, 2012.

[40] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 966–975, IEEE, 2007.

[41] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1247–1250, ACM, 2008.

[42] M. Graves, E. R. Bergeman, and C. B. Lawrence, "Graph database systems," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 14, no. 6, pp. 737–745, 1995.

[43] B. Iordanov, "Hypergraphdb: a generalized graph database," in *Web-Age Information Management*, pp. 25–36, Springer, 2010.

[44] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: mining maximal frequent subgraphs from graph databases," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 581–586, ACM, 2004.

[45] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference*, p. 42, ACM, 2010.

[46] R. Angles and C. Gutierrez, "Querying rdf data from a graph database perspective," in *The Semantic Web: Research and Applications*, pp. 346–360, Springer, 2005.

[47] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 976–985, IEEE, 2007.

[48] K. Riesen and H. Bunke, "Iam graph database repository for graph based pattern recognition and machine learning," in *Structural, Syntactic, and Statistical Pattern Recognition*, pp. 287–297, Springer, 2008.

[49] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *Social Computing (SocialCom), 2013 International Conference on*, pp. 708–715, IEEE, 2013.

[50] M. Buerli and C. P. S. L. Obispo, "The current state of graph databases," 2012.

[51] A. Popescu, "A comparison of 7 graph databases," 2013. `http://nosql.mypopescu.com/post/40759505554/a-comparison-of-7-graph-databases`, [retrieved: Feb. 2, 2013].

[52] M. C. Alekh Jindal, "Benchmarking graph databases," 2013. `http://istc-bigdata.org/index.php/benchmarking-graph-databases`, [retrieved: Jan. 29, 2013].

[53] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

Appendix A

Deterministic Candidates

In this appendix we list all the object Web applications that we crawled to find deterministic applications. A deterministic was required for forming an oracle for correctness experiments. The list is as following:

http://www.heatcityreview.com/somervillenews.htm
http://www.martinkrenn.net
http://www.kastanova.nl
http://www.al-awda.org
http://thething.it
http://www.engruppo.com
http://www.turbopatents.com
http://www.ece.ubc.ca
http://www.ubc.ca
http://www.cultofmac.com
http://www.python.org
http://metrics.codahale.com
http://www.uss.de
http://www.facebook.com
https://www.google.ca
http://www.bing.com
https://mail.google.com
https://github.com
https://code.google.com/p/guava-libraries
http://www.vogella.com
http://www.cacno.org
http://sedo.co.uk/search/details.php4?domain=thestart.net
http://www.iltasanomat.fi
http://www.fairvote.org
http://www.beehive.nu
http://www.littlewhitedog.com
http://www.provincetown.com
http://www.expedia.ca/?semcid=ni.ask.12908&kword=DEFAULT.-

NNNN.kid&rfrr=Redirect.From.www.expedia.com/Home.htm
http://www.airtoons.com
http://www.loco.pl/pl
http://www.grudge-match.com/current.html
http://www.kastanova.nl
http://www.acces-local.com/wordpress
http://www.sfchronicle.com
http://www.eldritch.com
http://www.eldritch.com
http://ruyguy15.150m.com
http://www.bghs.org
http://www.axis-of-aevil.net
http://www.infoshop.org
http://www.introducingmonday.co.uk
http://www.linuxdevcenter.com/pub/a/linux/2000/06/29/hdparm.html
http://www.twentysevenrecords.com
http://www.hallwalls.org
http://www.justfood.org
http://bshigley.tumblr.com
http://www.ottawacitizen.com/index.html
http://emeraldforestseattle.com/forums/ubbthreads
http://www.cancernews.com/default2.asp
http://www.usablenet.com
http://www.viz.com/naruto
http://www.viz.com/naruto
http://www.needcoffee.com
http://www.libraryplanet.com
http://www.coversproject.com
http://windows.microsoft.com/en-us/windows/support#top-solutions=windows-8
http://buffalo.bisons.milb.com/index.jsp?sid=t422
http://www.b3ta.com
http://antiadvertisingagency.com
http://www.sfbike.org
http://www.grimemonster.com
http://www.threadless.com
https://wilwheaton.net
http://www.hasbrouck.org
http://www.uberbin.net
http://www.gaijinagogo.com

http://lalibertad.com.co/dia/p0.html
http://www.wrightfield.com
http://www.modernhumorist.com
http://www.bcdb.com
http://desktopgaming.com
http://www.metalbite.com
http://nowyoulistentomelittlemissy.blogspot.ca
http://www.cimgf.com
http://www.paulmadonna.com
http://dawnm.com
http://typicalculture.com/wordpress
http://www.vegweb.com
http://www.newdream.org
http://www.isc.org/downloads/BIND/
http://hunyyoung.com
http://home.planet.nl/ mooij321
http://www.antique-hangups.com
http://www.project451.com//
http://lmuwnmd.wpengine.com/wp-signup.php?new=www.techblog.com//
http://c-level.org
http://rprogress.org/index.htm
http://www.math.mcgill.ca