Concurrent Multi-browser Crawling of Ajax-based Web Applications

Stefan Lenselink

Concurrent Multi-browser Crawling of Ajax-based Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Stefan Lenselink
born in Gouda, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Concurrent Multi-browser Crawling of Ajax-based Web Applications

Author: Stefan Lenselink
Student id: 1288369
Email: S.R.Lenselink@student.tudelft.nl

Abstract

Web applications are becoming more and more the replacement of desktop applications, using Ajax as their main technology to achieve user-friendly and interactive user interfaces. The best known example is the email application GMail, other examples include Google Docs, Google Maps, Facebook, Yahoo Mail and Twitter. These technological advantages brings some number of challenges when crawling those applications. The state of the user interface can not be accessed directly as with original web applications. This disables the usage of traditional web crawlers, which uses static analysis to find new pages. A dynamic approach, using a real browser of explore the user interface, was proven to be very successful. The dynamic approach did not scale well to large application, when applications grew the limits of this approach were reached. In this thesis we first perform a measurement to find the exact limitations of using this dynamic approach, knowing the limitations we propose a technique to improve the scalability of Ajax web crawlers. We use the model of web application to apply a dynamic partition function, which we use to analyse the different parts of an Ajax application concurrently using multiple browsers. We implemented our approach in the open source testing tool Crawljax. We describe three case-studies evaluating the effectiveness of our implemented performance improvements. The results are very promising, showing expected performance improvements, and enabled us to use Crawljax successfully in large production environments.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Mesbah, Faculty EEMCS, TU Delft
Company supervisor:	ir. A. Schwartz, Hostnet Amsterdam
Committee Member:	Dr. M. Pinzger, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Iosup, Faculty EEMCS, TU Delft

Preface

When I first heard about the therm Ajax, back in 2005, my first reaction was *"What else is new?"*. I was not impressed by the term as I was already using most of the techniques, I once stated *"This Ajax-buzz is only a hype, it will be over soon!"*. Now my study career has come to an end, I would like to withdraw this statement. Ajax has proven not to be a hype and it has earned its place in the software engineering world. When I first visited Arie van Deursen in the spring of 2009 I wanted to do a masters project in the testing area. Arie introduced me to Ali Mesbah, together we started our journey to optimise the performance, and general improve Crawljax.

There are many people I would like to thank making this thesis possible. First, I want to thank Ali for all his help and support, I learned alot working together with him. He also teached alot about the scientific and research world. Secondly I want to thank Arie for supporting our initial trip to Google London, and not to forget my intern-ship at Google London. This brings me to thanking all the employees at Google who helped me during my intern-ship in London, but especially Mike Davis for being a very good host during my time there.

I want to thank my lovely girlfriend Marie-Louise for supporting me the past four years, especially the last few months when I was nearly never home. Finally, I want to thank my parents supporting me all those years.

Stefan Lenselink
Delft, the Netherlands
December 3, 2010

Contents

List of Figures

Chapter 1

Introduction

Web applications are becoming more and more the replacement of desktop applications. In this chapter we introduce the techniques that support this change, and we give an outline of this thesis. The first section presents Ajax, the major new technique and architectural change for web applications over the past years. Section 1.2 discusses and explains the operation of a crawler. Section 1.3 presents the combination of a crawler and the Ajax technique. Section 1.4 discusses the possible performance and scalability issues when using an Ajax crawler. Section 1.5 introduces the goal and the research questions for this thesis.

1.1 Ajax

Ajax (Asynchronous JavaScript and XML) [22] is one of the most rising and promising techniques in the web application development area of the past few years. Capturing the traditional multi-page application into a single page increases the responsive and interactive experience of a user. Users do not have to "click-and-wait" any more, and the page does not have to be re-rendered again every time an interaction takes place, i.e., only small sub-sets of the page need to get updated. With these new dynamic applications a new term has emerged, Web 2.0 [46], which is used to mark the changes in web applications in facilitating communication, information sharing, interoperability, and collaboration. The term Web 2.0 is used to denote Ajax applications but is also, and more commonly, used to denote user-generated content.

The addition of the responsiveness brought by the Ajax technique makes it possible to operate applications on a web server and inside a browser as if they are desktop applications. Currently the web application market is becoming increasingly dominant and there are operating systems designed around them such as the Chrome OS from Google [49] and the WebOS from Palm [1]. This shows the importance of web applications as a replacement of ordinary applications.

Ajax is a clever combination of using the client-side JavaScript engine to update small parts of the Document Object Model (DOM) with information retrieved by asynchronous server communication. By using Ajax technology developers can create applications in which the page does not have to be re-rendered again every time an interaction has taken place; only small sub-sets of the page need to get updated. Therefore the users experience a very fast responsive application inside the web-browser. The

application is available everywhere the user connects to the internet, and is accessible with every browser. This eliminates the main disadvantages of having to install a full blown desktop application on a computer with a certain amount of computational capacity and the troubles of sharing files with people or other locations. This makes the use of cloud computing interesting. Cloud computing is the term used to describe the trend in the computing world in moving away from desktop applications to on-line services [25].

Although the web applications are not a new phenomena, the use of Ajax techniques are. These new techniques also require a good quality of service, of which testing is an important aspect. In the next section we investigate the use of a crawler as a testing technique.

1.2 Crawling

For normal desktop applications all kinds of techniques are developed in order to ensure a certain quality of service. Techniques like, unit-testing, regression-testing, integration-testing and functional-testing [20]. Testing enables software companies to reduce the number of bugs when releasing a new version. The web application market does not have many reliable testing tools that allow software companies to guarantee quality of service.

To be able to test traditional web applications a crawler can be used [18], for example to detect broken links or detect errors displayed on pages.

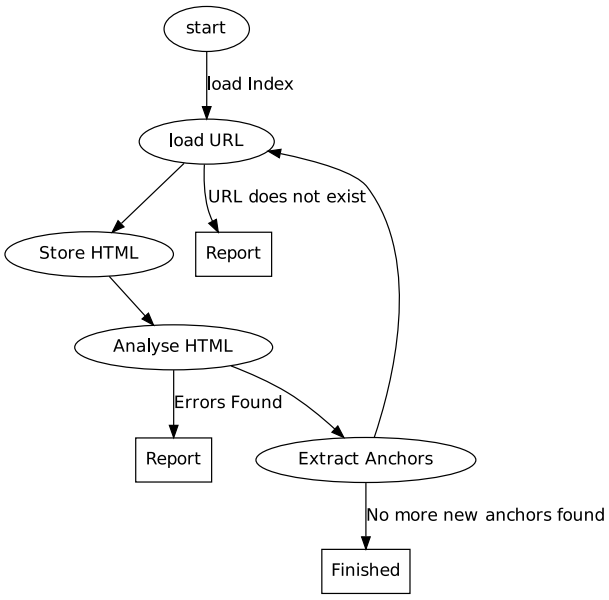


Figure 1.1: Operations of a crawler

Figure 1.1 shows the operations of a crawler for traditional web applications. It

consists of going to the *index* page, store the page HTML source code, analyse the stored HTML source code and extract all the anchor tags. After the anchor tags have been extracted the corresponding URL's (Uniform Resource Locator) can be used to "restart" the system for further exploration. The stored HTML files can be examined for failures. If an anchor tag in one of the stored files does not exist it can be reported. Also the contents of the stored HMTL page can be analysed to see if errors are displayed to the user. Every state or page of a traditional web application has a directly accessible URL, which can be use to directly request the contents of that web page.

1.3 Crawling Ajax

The new Ajax technology does not include the property of having a unique URL representing a unique state in an application [58]. Due to the lack of an external reachable unique state, a state reached by URL, crawlers are not able to access the full content of an Ajax application without the use of a pre-programmed JavaScript engine [54]. This problem of not having a reachable state by URL occurs when crawling[41] and testing an Ajax application[42].

To circumvent this problem Mesbah et. al. proposed to crawl Ajax applications by "Inferring User Interface State Changes"[41]. Their technique focuses around a state machine which stores the "actions" a user executes on a web-page inside a real browser, starting from the root, the *index* state, and following traces down to the final state of a certain path. These states are discovered through searching the current DOM-tree for possible elements at which events can be fired. These events include for example the `onClick`, `onMouseOver` or `onMouseOut`, and firing the events on all the possible candidate elements may result in a new states. The result of an event, the DOM-tree, is compared with the DOM-tree from before the execution. If the DOM-tree is changed a new state of the application is added to the state machine linked to its predecessor state. The edge between the two states represents the element and event combination that results in the new state originating from the previous state. By storing the combination of an element and an event, the crawler is able to repeat the flow of actions, which result in the given state. By using this information it is possible to bring an Ajax application to a given state, and this makes an Ajax application state aware by adding an external indexing shell. In the next section we investigate the performance and scalability of using such an Ajax crawler.

1.4 Performance and scalability

Until recently there was no testing framework available that could perform an automatic full sized test of an Ajax application. To fulfil this need Crawljax[1] was developed by Mesbah et al. [41, 42, 53]. One of the main features and usage of Crawljax is to be able to perform regression testing before the release of a new version of an Ajax application. As software evolves, regression testing is applied to modified versions of the software to provide confidence that the changed parts behave as intended and that the changes do not introduce unexpected faults, also known as regression faults. Using a crawler in an experimental setup has proven to be very successful [41, 42, 53].

[1] `http://www.crawljax.com`

Compared to traditional web crawlers Crawljax has more work to explore. In traditional web crawlers only anchor tags can result in a new state, but when Ajax is used nearly all elements can result in a new state, because almost every element can have an event listener attached. This increases the number of elements that needs to be examined. States in traditional web applications have a unique URL, and this URL can be used to store which state has already been explored. When using the Ajax technique this is not possible, and every element must be examined. If an application contains many states the number of candidate element that needs to be examined can become so large that the crawling can not be performed within a given time limit.

To determine if a state found is already explored the state needs to be compared with all the previous found states. This last operation, searching for a clone-state, is performed in memory. When the size of the DOM-tree increases, or the number of states in an application increases, Crawljax might run out of memory.

1.5 Goal and Research Questions

In this thesis we investigate ways of optimizing the crawling performance. To that end our research questions can be formulated as follows:

RQ1 What is the current performance of Crawljax? We try to find the current limits of Crawljax by conducting performance measurements to find the memory usage, maximum number of states, and runtime performance. Once this is known we can continue with the next research question.

RQ2 How can we improve the performance? We try to find solutions to increase the limits we found while investigating the first research question.

RQ3 How effective are the proposed improvements? Once the performance improvements are described and implemented we measure again the performance to analyse the proposed improvements.

We start this thesis with related work in chapter 2 to give a background in the supporting research fields. Following the related work, crawling Ajax sequentially is introduced and explained in chapter 3. In chapter 4 the limitation for the Ajax crawlers, in terms of memory usage, maximum number of states, and runtime usage is investigated. When the limits are known, we first propose the back-tracking optimisation in chapter 5, and as a second optimisation we propose a concurrent approach in chapter 6. The implementation details of this approach are discussed in chapter 7; this implementation is evaluated in chapter 8. Chapter 9 discusses changes proposed and implemented in this thesis, while chapter 10 concludes this thesis by finding the answers to the research questions.

Chapter 2

Related Work

In this chapter the work related to crawlers, scalability of crawlers and using them concurrently is discussed. The first section discusses the state space explosion problem. Section 2.2 discusses the state space explosion problem within the finite state machines, and section 2.3 discusses the state space explosion problem within the model checking world. Section 2.4 discusses state space reduction techniques. Section 2.5 discusses the concurrent state space exploration techniques. Section 2.6 discusses the operation, use and challenges of parallel crawlers. Section 2.7 discusses the use of a distributed environment to solve concurrent state space exploration algorithms.

2.1 State space explosion

To prove the correctness of a system with respect to a formal specification two main methods are known; theorem proving and state space methods. Theorem proving is based on formulating mathematical formula's to model the correctness of claims for a system. A theorem is proved either manually or with the help of a theorem proving tool.

The state space method performs an automatic analysis and verification based on the behaviour of a system [55]. The state space method consists of a structure, which contains all the states that a system can reach and all the possible transactions a system can make between the states.

The features of the state space method are:

- Automatic analysis of a system; an analysis can be written for one state and applied to all the states. The whole system is analysed, not only the part where explicit analyses are written for.

- Examining the behaviour of a system; instead of only being able to examine a single state, it is also possible to examine the system as a whole.

- Different kind of verification and analysis questions possible; while passing all the states inside a state exploration algorithm different kind of verifications or analysis can be executed.

These key features made researchers put in a lot of effort over the last few decades to reduce the state spaces [55]. The larger the state space is, the more time and memory it consumes to verify a system, this is known as the state explosion problem [34].

Within the current scientific world the state explosion problem is also known as state space explosion. In the field of informatics researchers are facing this problem with the model checking [11, 28], and in section 2.3 we investigate model checking further. In the next section we investigate the state space explosion problem in finite state machines.

2.2 Finite state machine

Finite state machines are used to describe the behaviour of a system by recording the transitions from one state to another state. This method is mostly used in verifying software systems or software protocols [27].

The state machine used inside a crawler is not a fully specified state machine, but an incomplete specified state machine. A completely specified state machine is a state machine where every transaction results in a unique new state [56]. When examining an Ajax application it is possible to have multiple transactions resulting in the same state, e.g., two different links can result in the same page. This observation leads to the fact that the state machine used is an incomplete specified state machine. The minimal version of a completely specified state machine can be found in polynomial time [32]. Incomplete specified state machine, are proved to be NP-complete [48] in terms of finding the minimal state machine. This means that there is no algorithm known that minimises an incomplete specified state machine in polynomial time. In the next section we investigate the state space explosion problem within the model checking world.

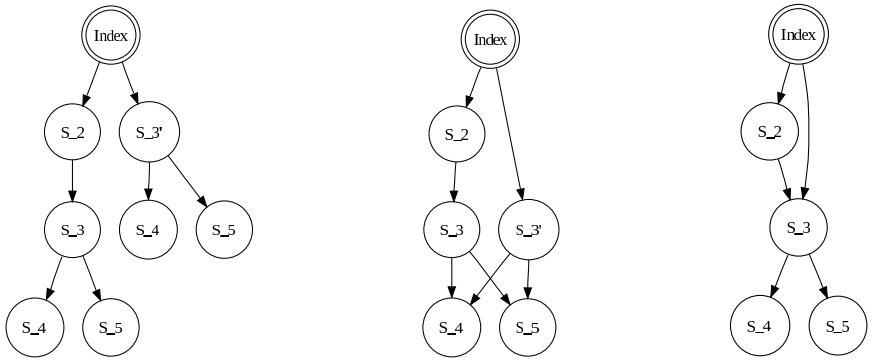
2.3 Model checking

Model checking is a method that checks if the model of a system meets the pre-described constraint; it is mostly used for hardware (circuits) and protocol checking [10, 11]. Within the model checking domain the state space reduction methods can be divided into two main groups: reduction methods that either require user input or operate automatically. Within the model checking domain there are two different kind of model checking techniques specified; *explicit model checking*[9, 11, 16] and *symbolic model checking*[9, 11, 16]. The symbolic model checking method operates on models which are specified by using boolean algebra. The statements made about the system are specified as propositions. The explicit model checkers use (most of the time) their own programming language in which model behaviour and validity is specified. For those techniques a lot of optimisations have been developed [5, 11, 29]. In the next section those optimisations are investigated further.

2.4 State space reduction techniques

For a state space exploration system reduction of the state space explosion problem [55] can be separated into two tasks: Reducing the memory usage and reducing the

total runtime [47]. Both tasks can be achieved by reducing the number of states in total. As the number of states drops the memory required to store all the states reduces and the runtime required to process all the states decreases as well. For the state space exploration system to function correctly, the answer received from the reduced system must be equal to the answer received from the full system [51].



(a) Full state machine (b) No identical states (c) No similar states

Figure 2.1: State Machine Optimisation

Figure 2.1a shows the full state space, in figure 2.1b the states which are exactly the same are removed. Figure 2.1c shows an abstract example of the similarity between two states which can be regarded as the same. Roest et al. developed a method for detecting similar states based on oracles which is presented in [53]. When the number of states can not be reduced any further, the focus moves towards the optimisation of the runtime. This can be achieved using an alternative state exploration method like Sweepline [8, 19, 36], Heuristics [24] or using a concurrent approach. In the next section the concurrent approach is investigated in more detail.

2.5 Concurrent state space exploration

Most modern workstations sold today contain a multi-core processor. Instead of increasing the speed of a single processor, producers have focused on the development of multi-core processors [30]. Multi-core processors provide parallelism using shared memory, and by applying multi-threading it is possible to make use of all the cores simultaneously. While introducing multi-core processors, chip makers also switched to 64-bit word size, allowing 16.8 million terabytes of memory to be addressed by a processor. The memory requirement is not a major issue with multi-core processors.

The performance of a concurrent approach is determined by the quality of the partition algorithm [21, 40]. This algorithm tries to divide the full state space into as many subsets as there are processors participating in the concurrent operation. Each processor in the computation owns one of the state subsets and is responsible for the states inside this subset. Typically a processor starts with one state in the subset and tries to find and explore successors of the initial state, storing them if a successor is found.

Mutual exclusion locks can prevent that different processors perform redundant work by exploring the same parts of the state graph . This process continues (recursively) until the overall queue is empty. If the queue is empty and every processor has finished his work, the algorithm is finished.

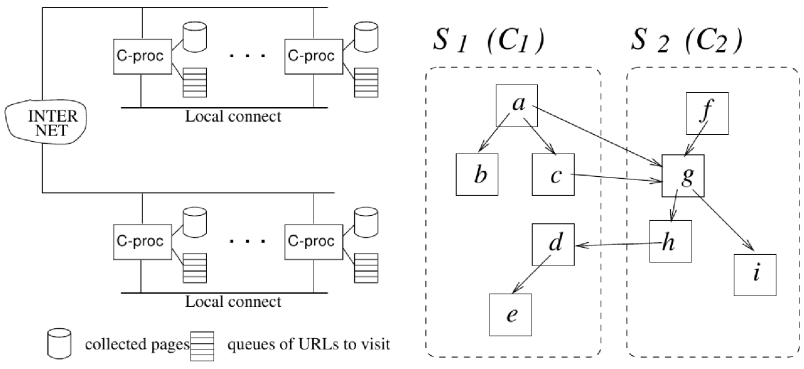
The partitioning can be static or dynamic. Lerda et al. [40] and Garavel et al. [21] both use a static partitioning algorithm. Dynamic partitioning can be used to assign work from one processor to a less busy processor, whereby the dynamic partition function controls the equal termination of all the processors.

The main problem that remains to be solved is that the algorithm performs a non depth-first search (breadth-first) [40]. To be able to execute a repeatable experiment the behaviour of the system must be predictable. Executing the same experiment multiple times must result in a predictable flow of events. Depth first search execution is predictable while the breadth-first execution is not predictable. By adding synchronisation the processors can be forced to execute a depth-first search. In doing so each processor will be forced to wait for a reply from the destination processor whenever it sends a message, this algorithm does not take advantage of parallel processing [21, 40].

Results reported by Holzmann et al. [30] vary from application to application and from test to test. On average 50% speed increase is found using a dual-core computer.

2.6 Parallel Crawlers

Due to enormous size of the Web, it is often imperative to run a parallel crawler. A single-process crawler simply cannot achieve the required download rate in certain cases [7]. Figure 2.2a shows the general architecture of a parallel crawler, a single crawler is represented by `C-proc`. Every crawler has its own data-store to store its retrieved pages, and a queue of URL's to visit.



(a) Architecture of a parallel crawler (b) Parallel crawling two sites

Figure 2.2: Operation of a parallel crawler[7]

Every `C-proc` functions as a stand-alone crawler as described in section 1.2. For

parallel crawlers the work may be distributed either on the same local network or at geographically distant locations. In their work [7], Cho2002 et al. distinguish two types of distributed crawlers:

- *Intra-site parallel crawler*. All the crawlers run on the same local network, and communicate through a high speed interconnect such as a LAN.
- *Distributed crawler*. Crawlers operate on geographically distant location, connected by and communicate via the Internet.

Figure 2.2b shows two sites, $S1$ and $S2$, which are crawled by crawler $C1$ and crawler $C2$. As can be seen from the figure the two sites link to each other, in $S1$ there are two links going from a and c to g. From $S2$ there is one link going from h to d. In order to avoid creating an overlap, both crawlers need to coordinate which page is downloaded by which crawler. In [2, 4, 7] the following methods are noted:

- *Independent*. Every crawler may follow and download all links it finds, there is no coordination. This can create an overlap with the other crawlers.
- *Dynamic assignment*. There is one central coordinator to divide the work over the crawlers. The coordinator uses a partition function to divide the work, and distribute it over the crawlers.
- *Static assignment*. Every crawler is able to determine if a link needs to be explored or if the link is explored by another crawler. This is used by Boldi2004 et al. presented in [4].

In the next section we investigate a programming model to operate concurrent state space exploration systems and parallel crawlers in a distributed environment.

2.7 MapReduce

The MapReduce programming model for processing and generating large data sets was introduced by Dean et al. [13]. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The user of the MapReduce library only programs his computation, the `map` and `reduce` functions. The library takes care of the messy details of the parallelism, fault-tolerance, data distribution and load balancing.

The input data is split in M pieces by the MapReduce library of typically 16 to 64 megabytes (MB) per piece. The `Map` function programmed by the user is started with a key/value set parsed from the input piece. The `Map` function processes the key/value pair and produces a new intermediate key/value pair. These key/value pairs are provided to the `Reduce` function programmed by the user, and the `Reduce` function processes the given key/value pairs into a single output file.

Figure 2.3 shows the main execution of the MapReduce algorithm. From the left the input data is processed through the `Map` functions through the `Reduce` functions into the output files. During the operation of a MapReduce cycle a special user process is active, called the Master. The master keeps track of the processing, determines when all the `Map` operations are finished and the `Reduce` phase must be started.

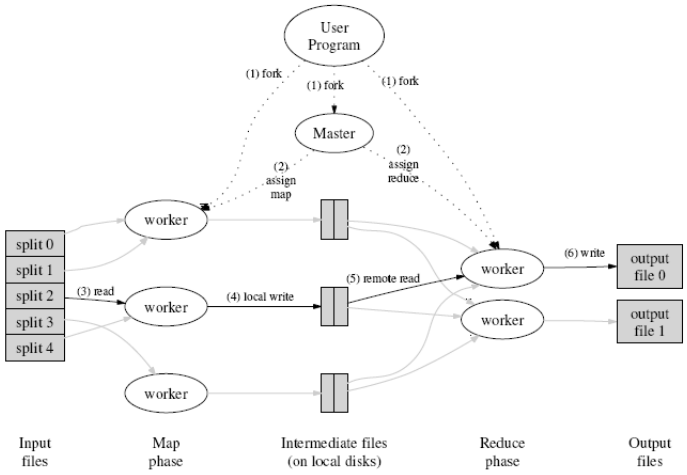


Figure 2.3: MapReduce execution overview as originally appeared in [13]

The main advantage of MapReduce is that programmers can be oblivious to parallelism and distribution [38]. The MapReduce paradigm automatically enforces a developer to write code which does not have to communicate between threads. This is also the main problem concerning the applicability on web crawlers. One of the main building blocks of the MapReduce algorithm is the fact that the individual `Map` or `Reduce` instances are free of any synchronisation and of any shared memory. This also implies no communication with other worker nodes by other media like message passing.

MapReduce is mostly designed for data processing of large amounts of data (over multiple terabytes) [13, 38, 52]. State exploration applications are not data processing applications. At the start there is no data available and during operation more data is discovered. To use the MapReduce structure with state exploration applications the application needs to be altered in a way to support MapReduce operations. To operate for example within Crawljax one approach would be to regard the candidate elements that need to be examined for a given state as the input data. In the `Map` function the candidate elements can be explored and as a result are new states as intermediate keys. The `Reduce` function can merge all the new found states into a single output graph. One of main problems with this setup would be the handling of new discovered candidate elements. These must somehow feeded back into the system; as suggested by [38] the problem could be decomposed by multiple MapReduce computations.

In this chapter we have discussed state space exploration algorithms, finite state machines and model checking reduction techniques. The use of concurrent reduction techniques and the applicability of MapReduce on those are investigated. In the next chapter we introduce the challenges and ideas behind crawling Ajax applications and we introduce the original crawling algorithm.

Chapter 3

Crawling Ajax Sequentially

In this chapter we introduce how Ajax applications can be crawled using a sequential approach. The first section introduces the general challenges of crawling Ajax applications, and we provide some solutions to these problems. Section 3.2 defines the abstract idea behind crawling Ajax applications, and some of the technical terms are introduced. Section 3.3 defines the algorithm used to sequentially crawl Ajax applications.

3.1 Challenges of crawling Ajax

The Ajax technology heavily relies on the client-side JavaScript engine. The JavaScript engine operates between the browser and the web server, using the XMLHTTPRequest to communicate with the server. The JavaScript engine is able to execute events originating from elements, these can be `onClick`, `onMouseOver` or `onMouseOut` events. Using the `XMLHTTPRequest` the JavaScript engine is able to send a small request to the server and with the received result the DOM-tree can be updated.

A traditional crawler only has to extract all the external anchor tags to find new states. Using the Ajax technique every element with an event handler can cause a state change. The traditional anchor tags are replaced by events on the client-side JavaScript engine.

When an event is executed on an element it is difficult to determine if the event is finished. This is because browser API's do not provide any method of detecting when the effects on the page of issuing a particular event have finished [45]. The easiest solution to this problem would be to issue a timeout after every event. This solution has the usual drawbacks: if the specified timeout is too short an incorrect DOM-comparison can be performed. If a too long timeout is used the delay introduced is unnecessary.

A state in an Ajax application does not have a unique URL, like a state in a traditional web application. This excludes the possibility of directly accessing a state, and a state can only be accessed using its chain of events leading to that state.

The operation of going back is not trivial because an Ajax application changes the DOM-tree during its execution. Those DOM-changes are not registered with the browser history-engine by default. Executing the back-operation will result in loading the last loaded page which is not the previous state. When a special history framework

is used, like for example jQuery History Plugin[1] or Really Simple History[2], which registers itself to the back-button of the browser, the back-operation will result in the previous state.

Now we know the challenges of crawling Ajax. In the next section we introduce the idea behind crawling Ajax proposed by Mesbah et al. in [41].

3.2 Idea behind crawling Ajax

The highly dynamic aspect of the Ajax applications, and the dependency on the client-side JavaScript engine discussed in the previous section makes it clear that a static analysis, an analysis by requesting URL's to retrieve the static HTML source, is not possible. Instead a *dynamic analysis* approach is applied, in which actual events are fired on all relevant elements inside a `real` browser. From these events a *state-flow graph* is inferred, which represents the user interface states and possible transitions between them.

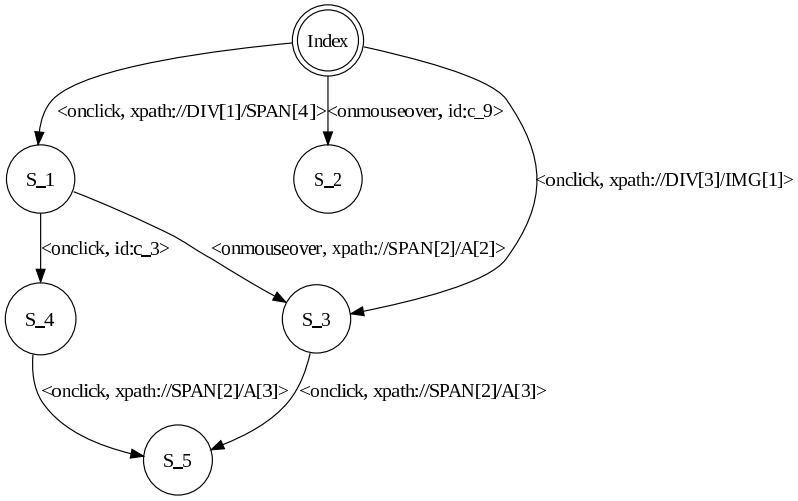


Figure 3.1: The state-flow graph visualisation

Figure 3.1 shows an example state-flow graph of a simple AJAX site. It shows a total of six states (nodes) and seven events (edges). On the edges there are labels with an identification (either via its ID-attribute or via an XPath expression) of the clickable. From the *index* state three candidates result in the new states; S_1, S_2 and S_3.

The state-flow graph is used to access a state, whereby the shortest path from the index to the state that needs to be accessed is determined using Dijkstra's shortest path algorithm[15]. The chain of events found along this path is executed to reach the destination state. To solve the problem to see if an event is finished a static wait-timeout is used after every event executed. The state-flow graph can also be used

[1] `http://tkyk.github.com/jquery-history-plugin/`
[2] `http://code.google.com/p/reallysimplehistory/`

to generate a mirror or execute test-cases based on that state. The state-flow graph could be interpreted as a model of the Ajax application. The generated model can be used to check if the system functions according to its specifications. More precise, the invariants specified check if the generated model is conform the specifications of the system [42].

If no back-operation framework is used, the operation of going back to the previous state consists of reloading the *index* page and following all the events from the state-flow graph leading to the previous state. This process can be optimised by using Dijkstra's shortest path algorithm[15].

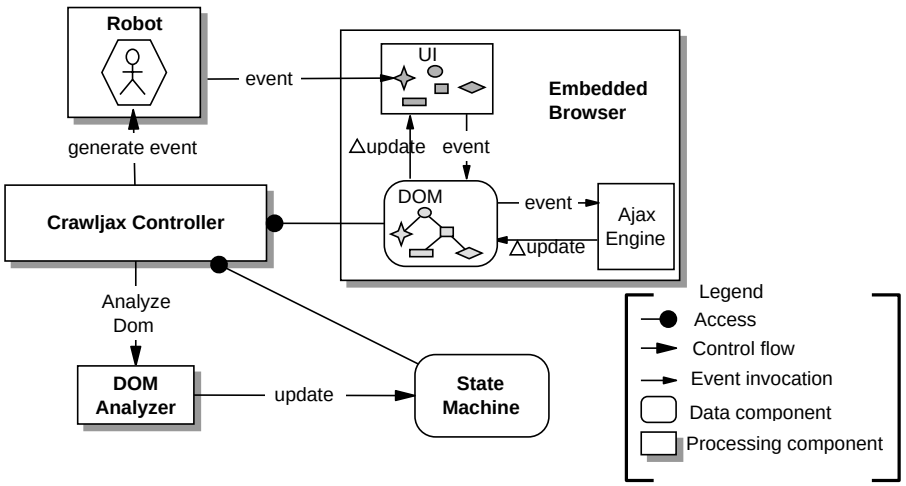


Figure 3.2: Processing view of the crawling architecture, as introduced in [41]

Figure 3.2 shows the processing view of the crawling architecture. It features the following entities:

- EmbeddedBrowser. The `real` browser, in which the user interface is inferred to trigger state changes. The browser used must be capable of executing JavaScript, and to process all other techniques used by Ajax applications.
- Robot: A robot is used to click on elements or fill-in form elements.
- StateMachine. The finite state machine maintains the state-flow graph, is responsible to add new edges and states and maintains a pointer to the current and previous states.
- Controller. The controller maintains the robot-instance and is able to instruct the robot to execute actions. It also has access to the current DOM-tree to feed the DOM-changes to the DOM Analyser.
- DOM Analyser. The comparator is used to detect whether the DOM-tree is changed after an event or is used to compare DOM-trees when searching for duplicate-states. Oracle comparators can be used to compare the targeted differences from the DOM-trees by first stripping the differences, defined through, e.g., regular expression or an XPath expression, and then comparing the stripped DOM-trees using a simple string comparison[53].

The components and techniques to crawl Ajax applications are now introduced, and in the next section the algorithm used for crawling Ajax applications sequentially is introduced.

3.3 Original crawling algorithm

Algorithm 1 shows the original algorithm as first published by Mesbah et al. [41]. This algorithm consists of two procedures; the start procedure (line 1) and the main crawl procedure (line 7).

Algorithm 1: Crawling AJAX

```
   input : URL, tags, browserType
 1 Procedure MAIN()
 2 begin
 3     global browser ← INITEMBEDDEDBROWSER(URL, browserType)
 4     global robot ← INITROBOT()
 5     global sm ← INITSTATEMACHINE()
 6     CRAWL(null)
 7 end
 8 Procedure CRAWL(State ps)
 9 begin
10     cs ← sm.GETCURRENTSTATE()
11     Δupdate ← DIFF(ps, cs)
12     f ← ANALYSEFORMS(Δupdate)
13     SetC ← GETCANDIDATECLICKABLES(Δupdate, tags, f)
14     for c ∈ C do
15         robot.ENTERFORMVALUES(c)
16         robot.FIREEVENT(c)
17         dom ← browser.GETDOM()
18         if STATECHANGED(cs.GETDOM(), dom) then
19             xe ← GETXPATHEXPR(c)
20             ns ← sm.ADDSTATE(dom)
21             sm.ADDEDGE(cs, ns, EVENT(c, xe))
22             sm.CHANGETOSTATE(ns)
23             if STATEALLOWEDTOBECRAWLED(ns) then
24                 CRAWL(cs)
25             sm.CHANGETOSTATE(cs)
26             if browser.history.CANGOBACK() then
27                 browser.history.GOBACK()
28             else
                   // We need to back-track by going to the initial state.
29                 browser.RELOAD()
30                 ListE ← sm.GETPATHTO(cs)
31                 for e ∈ E do
32                     re ← RESOLVEELEMENT(e)
33                     robot.ENTERFORMVALUES(re)
34                     robot.FIREEVENT(re)
35 end
```

In the start procedure the system is initialised; the browser, robot and the state machine are initialised (line 3 to 5). After the initialisation the crawl procedure is

started with an empty state (line 6); full crawling is started at the *index* state.

When the crawl procedure is invoked, the current state is first retrieved from the state machine (line 10). This current state is compared against the previous state (line 11). The Δ-update received is used to extract a list of all the candidate elements that needs to be examined (line 13).

A for-loop is executed over the candidate elements list to process every candidate. For every candidate the crawler first fills the form fields (line 15) and after that fires the event for that candidate (line 16). The DOM-analyser determines if the state is changed (line 18). If no state change is detected the for-loop continues with the next candidate element. If there is a state change according to the DOM-analyser, a new state is created, and it is added to the state-flow graph contained in the state machine (line 20). In order to recognise an already met state, a *hashcode* is computed for each DOM-tree state, that is used to compare every new state against the list of already visited states on the state-flow graph. If an identical or similar state is recognised in the state machine, that state is used, or otherwise a new state is created. On every state change, clone-state or new-state, an edge is added to the state-flow graph using the state machine (line 21). The current state of the state machine is changed to the new found state (line 22). If no limiting conditions are exceeded, like reaching the maximum depth, maximum runtime or maximum number of states, the crawl procedure is recursively called (line 24) to find new possible states at the detected new state.

When the recursive crawling is finished the current state pointer in the state machine is changed to point to the previous state (line 25). If an AJAX application uses a back-operation framework and thereby supports browser history (line 26) then for changing the state in the browser the built-in history back functionality to move backwards (line 27) is used. If the back-operation of the browser can not be used the *index* state will be reloaded (line 29), and the path to the current state (line 30) will be retrieved. For every event on the retrieved path, first the element is located on the page, then the corresponding form-values are filled and finally the event is fired (line 32 to 34). When the loop is finished the crawler will be "reloaded" into the previous state, so it has gone one state back.

To locate an element XPath has been adopted along with the attributes to provide a better, more reliable, and persistent element identification mechanism. For each clickable, the XPath expression of that element is reverse-engineered, which gives its exact location on the DOM-tree (line 19). This expression is saved in the state machine (line 21) and it is used to persistently find the element after a page-reload (line 32).

In this chapter we introduced the challenges, the ideas and the original algorithm for crawling Ajax applications as proposed by Mesbah et al. [41, 42, 53]. In the next chapter we are going to measure the performance of the original crawling algorithm to analyse the performance bottlenecks. In chapter 6 we extend the ideas and the crawling algorithm introduced in this chapter to support concurrent crawling.

Chapter 4

Measuring Performance

In this chapter we investigate the limits of the current implementation of Crawljax. In the first section we introduce the tools that are used to generate the testing infrastructure and to measure the performance. In section 4.2 we introduce the experimental applications that have been created to perform our measurements. Section 4.3 presents the results that are found during our measurements and the performance of Crawljax is analysed.

4.1 Measurement tools

To find the limitations of Crawljax, a specialised plugin called the Benchmark Plugin[1] has been written. This plugin is executed on every new found state and on every reload of a state. It records different measurement values, including the number of states, the number of edges, the runtime, the memory usage, CPU usage, number of revisited states, and number of revisited edges. It is executed when a new state is found and on every step of the back-tracking procedure. The plugin maintains two lists: an absolute list and a Δ-list. The absolute list is the total measurement from the start and the Δ-list contains the differences between the previous execution of the plugin. For example the runtime field in the absolute list is the runtime measured from the start of the crawling, while the runtime value in the Δ-list is the time taken between two measurements. The plugin is split into two separate parts; one part that contains the GUI, and a second part that performs the measurement. For the plugin to function correctly the use of the GUI part is optional, while the recording part is needed to actually record the benchmark data. The recording can be stored in files for post-processing. The GUI can be used as an interactive monitoring tool.

To perform our measurements we needed an Ajax application to execute Crawljax. All the exsisting Ajax applications used in previous research[41, 42, 53] where not flexible enough. We could not specify the number of states, edges or the contents of the Ajax applications. So we deceided to generate experimental Ajax applications, and therefore we created an application called TSG (Test Set Generator). The purpose of the tool is to generate a simple Ajax application. The generated application uses the jQuery framework[2] to perform its Ajax operations.

[1] `http://crawljax.com/plugins/plugin-benchmark/`

[2] `http://jquery.com`

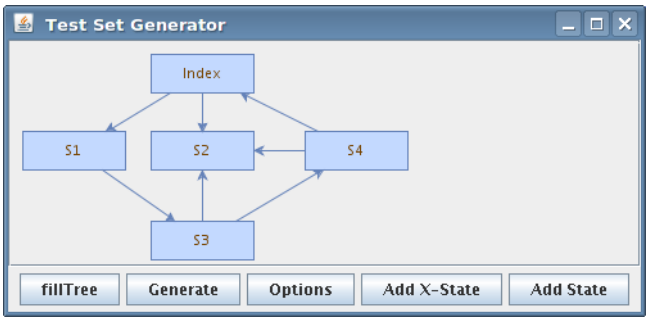


Figure 4.1: TSG GUI

Figure 4.1 shows the main GUI of TSG. The main GUI consists of a graph drawing tool, the JGraphX[3] graph drawing component is used as the framework to visualise the GUI. In the GUI states can be added and edges between those states can be drawn. There is a button for adding a single state or multiple states as a `flat-tree` underneath the last selected state. TSG also has an option to fill a full balanced graph for a given depth and a specified number of states.

A state drawn in TSG will result in a HTML file, an edge in TSG will result in an anchor tag inside the originating state HTML source code. The event connected to this anchor tag will result in the load of the HTML source code of the target state. Using this principle we are able to transform the graph visualised by TSG into an Ajax application.

In TSG an option has been added to fill the HTML source with a place holder text, Lorem ipsum[4], until the resulting HTML source code will reach a predefined size. By doing this we can control the size of the resulting DOM-tree when loading the states in Crawljax.

In the next section we use TSG to create our "experimental" Ajax applications.

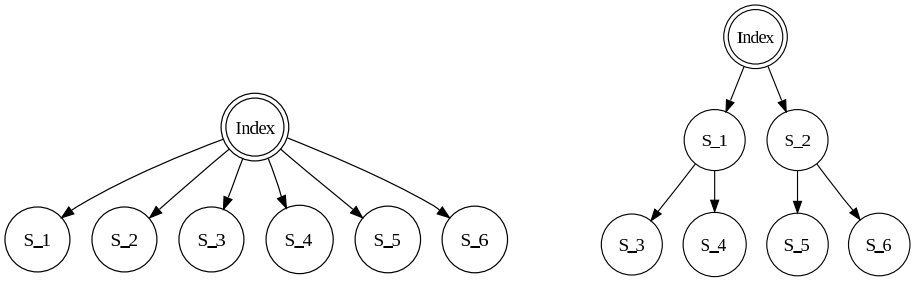
4.2 Experimental applications

For our initial benchmarks we created several "experimental" Ajax applications. Two main categories are defined: Flat state tree, further referred as `flat-tree`, and full balanced tree, further referred as `full-tree`.

In the `flat-tree` systems every state, except the *index* state is a final state, the *index* state holds all the edges available in the system as can be seen in figure 4.2a. The `full-tree` systems have more depth and the *index* state only contains the edges to the first layer of states and from those states the system continues.

We chose to create 200 states in addition to the index state. For the `flat-tree` systems this means that the index holds 200 edges resulting in 200 anchor tags to examine. For the `full-tree` systems we added two extra parameters: depth and nodes. The depth is the number of levels of states that exist underneath the *index* state. Figure 4.2b has a depth of 2; this will result in the following state-flow at its deepest point: *index* $\rightarrow$ *S_1* $\rightarrow$ *S_3*. In this example *S_3* is a final state at level 2. The node parameter

[3] `http://jgraph.com/jgraph.html`
[4] `http://lipsum.com`



(a) `flat-tree` (b) `full-tree`

Figure 4.2: Example experimental applications

defines the number of outgoing edges that a state has, in figure 4.2b this is 2. For our experiments we chose to use a depth of 3 and for the number of nodes we chose 6.

For every resulting state in our experiments we chose 0 KB, 128 KB, 256 KB, 384 KB, 512 KB, 768 KB and 1 MB as size for our placeholder text. The 0 KB case will give us our base-line experiment, where the resulting system will only have the necessary anchor tags in the DOM-tree to operate correctly. Measurements performed on this system will tell use how much runtime and memory every system will use as a minimum.

In the next section we present the results of Crawljax runs on our generated Ajax applications.

4.3 Measurement results

In this section we try to find the maximum number of states Crawljax can handle. This is done using the "experimental" Ajax applications described in the previous section.

Table 4.1: Performance results of the `flat-tree` based systems with 200 states

Size	Runtime	Memory usage
0 KB	6 min	4 MB
128 KB	6 min 35 sec	104 MB
256 KB	7 min 45 sec	204 MB
384 KB	9 min 11 sec	304 MB
512 KB	11 min 13 sec	404 MB
768 KB	16 min 7 sec	604 MB
1 MB	22 min 10 sec	804 MB

Table 4.1 shows the result data we measured using our Benchmarking Plugin while running the `flat-tree` based Ajax applications. The first column shows the size of extra content we added to the DOM-tree, the second column displays the runtime, while the last column displays the total memory usage of the state-flow graph. The

first row, where we added 0 KB of extra content, is the base measurement. The runtime used in this experiment is the minimum runtime for the Ajax application without any added content. This runtime is also at least used in all other experiments. When the size of the DOM-tree increases the total memory increases linearly. If the DOM-size is doubled from 128 KB to 256 KB the total memory usage is doubled as well, this also holds for example for the increase from 512 KB to 1 MB.

The runtime does not increase linear when the DOM-tree size increases linear, the runtime increases slightly exponential. When for example the DOM-tree size doubles from 128 KB to 256 KB, the added runtime is tripled. Remember that the base measurement, when the size of the DOM-tree is not increased, takes 6 minutes. The extra runtime when running with 128 KB is 35 seconds, while the extra runtime when running with 256 KB is 1 minute and 45 seconds. This is longer than the excepted 1 minute and 10 seconds. When the DOM-tree size is increased from 512 KB to 1 MB the extra runtime is more than three times increased.

Table 4.2: Performance results of the `full-tree` based systems with 200 states

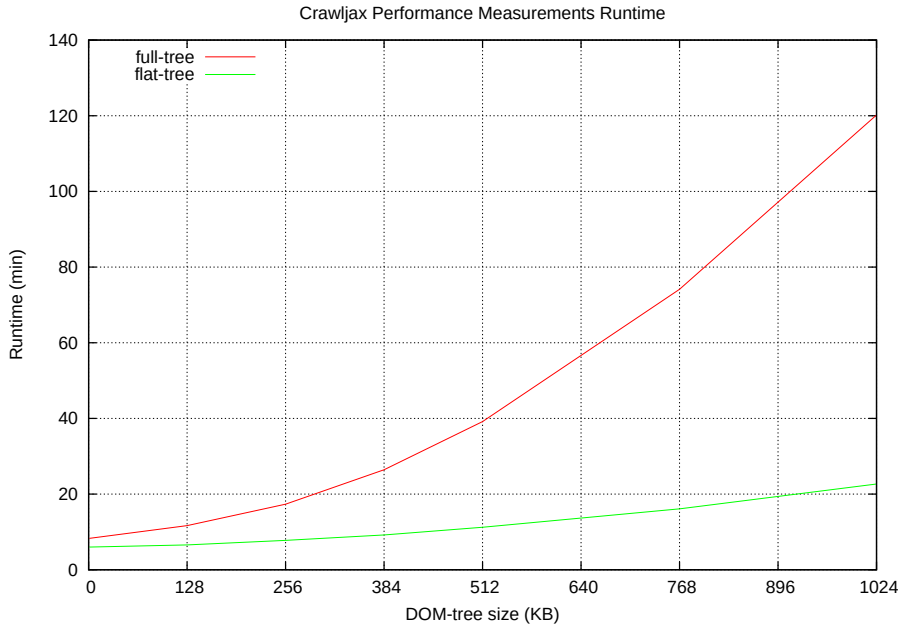
Size	Runtime	Memory usage
0 KB	8 min 17 sec	5 MB
128 KB	11 min 40 sec	114 MB
256 KB	17 min 19 sec	223 MB
384 KB	26 min 25 sec	330 MB
512 KB	39 min 9 sec	441 MB
768 KB	74 min 8 sec	654 MB
1 MB	120 min 11 sec	871 MB

Table 4.2 shows the result data we measured while running the `full-tree` based Ajax applications. It features the same columns as table 4.1 and we used the same stepping in the DOM-size. The memory usage in table 4.2 shows nearly the same trend as in the `flat-tree` based Ajax applications. Everytime the DOM-tree is doubled in size the total memory usage also is roughly also doubled.

The runtime increases much more than compared with the `flat-tree` based applications. This is due to back-tracking. In the `flat-tree` application the back-tracking only consists of reloading the index state, while in the `full-tree` application the back-tracking must be performed much more often to examine all the specified levels, because our experiment uses a depth of three levels.

Figure 4.3 shows the performance of Crawljax in a graph for both the runtime (4.3a) and the total memory useage (4.3b). The x-axis displays the size of the DOM-tree in KB, and in figure 4.3a the y-axis displays the runtime in minutes, and in figure 4.3b the y-axis displays the total memory usage in MB. The figures both show two graphs: one for the `flat-tree` based Ajax applications and one for the `full-tree` based Ajax applications.

The line for the `full-tree` based Ajax applications shows a slight exponential growth. Increasing the size of the DOM-tree of an Ajax application, increases the runtime at a slight exponential rate. If the time permitted to execute Crawljax is sufficiently large every application, regardless of the number of states, can be crawled. Although the time increases exponentially Crawljax was able to process 200 states of



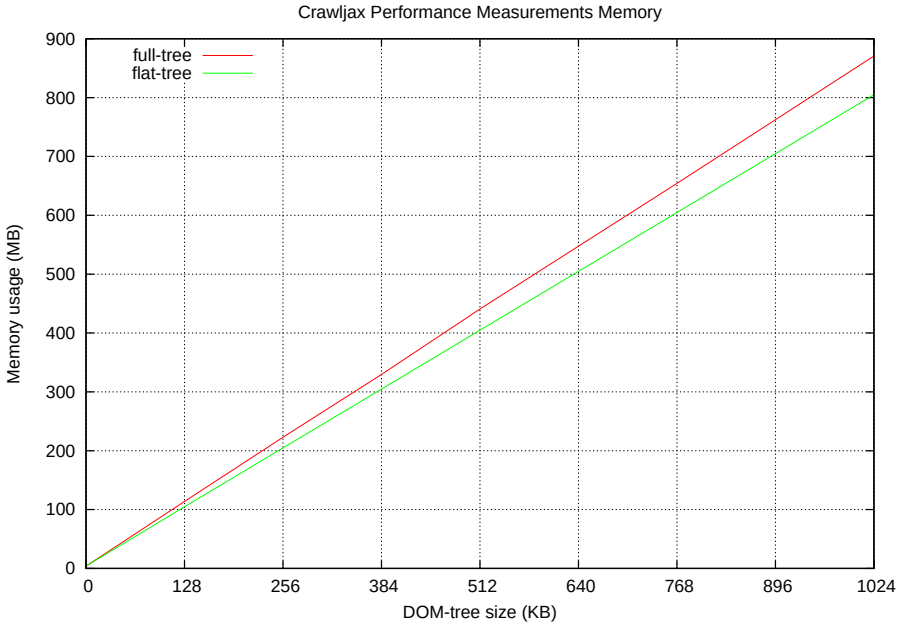
(a) Runtime results

Figure 4.3: Performance of Crawljax, varying the DOM-tree size

1 MB each in about two hours.

The limiting factor for the number of states examined is the amount of memory in a workstation. As can be seen from table 4.1 and 4.2 the amount of memory on a per state basis is about 4 times the size of the DOM-tree. Imagine a workstation holding 16 GB of memory. When the DOM-tree is 1 MB the maximum number of states is defined by $\frac{16384\ MB}{4 \times 1\ MB} = 4096$ states. It is most likely that enterprise applications will not have 4096 states but less, and for Crawljax to be useful on enterprise Ajax applications we therefore focus on solutions to optimise the runtime. Memory optimisations will be shortly discussed in chapter 10.

In the next chapter we discuss our first proposed optimisation solution, which is optimising the back-tracking operation. In chapter 6 we introduce our main runtime optimisation, concurrent crawling, for which we will give the implementation details in chapter 7 and evaluate the optimisation results obtained in chapter 8.



(b) Memory results

Figure 4.3: Performance of Crawljax, varying the DOM-tree size

Chapter 5

Back-tracking Optimisation

This chapter discusses the problem, that we encountered during our initial evaluation of the scalability of Crawljax, called the back-tracking problem. The first section presents the original design of the back-tracking operation. Section 5.2 proposes an optimisation that could be made. Section 5.3 evaluates the proposed solution, and section 5.4 summarises and concludes the proposed optimisation.

5.1 Original back-tracking method

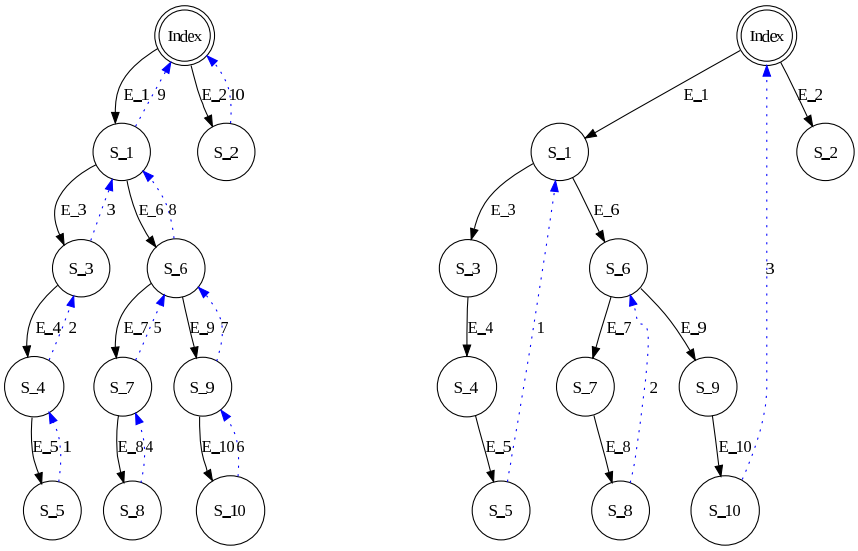
While crawling Ajax applications the states are evaluated in a depth-first manner [41]. First the crawler descents to the depth-most location. When the crawler reaches the depth-most location and it does not find any new states at that level it returns one level up. From there it continues to search further for new states not examined yet.

Returning one level up is done by reloading the *index* page and following all the events which lead to that state. Figure 5.1a shows an example of states found and processed. As the first action, the crawler starts at the *index* state and discovers two candidates which might lead to two states. These candidates will in the end result in states *S_1* and *S_2*. When the crawler is finished searching for candidates, at the *index* state, it continues a level deeper by executing the first found candidate, which results in *S_1*. This process is repeated (*S_1* $\rightarrow$ *S_3* $\rightarrow$ *S_4* $\rightarrow$ *S_5*) until the crawler is at the lowest level (*S_5*). When finished it returns to the previous state, and this is done by loading the *index* page and navigating to *S_4* by using the sequence of *E_1* $\rightarrow$ *E_3* $\rightarrow$ *E_4*. The different back-tracking steps are shown by the blue arrows in figure 5.1a. The process of returning to the previous state is repeated until the crawler is in a state where there is work left to examine. From *S_4* it goes back to *S_3* and than to *S_1* and at *S_1* there is work left because *S_6* is not examined yet.

In the next section we investigate the possibility to optimise the back-tracking.

5.2 Optimisation

Looking at our example of figure 5.1a, when returning from *S_5* to *S_4* there are no unexamined candidate elements left. The crawler returns back to level *S_4* without doing anything useful. The same happens at *S_3*. The only useful operation performed is changing the current state in the state machine. As shown in figure 5.1a the states are



(a) Original version

(b) Optimised version

Figure 5.1: Traversing the state space. The black edges, prefixed E_, denote the events resulting in a new state. The blue edges represent the back-tracking steps

changed back from *S_5* to *S_1* by visiting *S_4* and *S_3* without executing any functional work. This behaviour could also be captured by the blue arrow 1, as shown in figure 5.1b, which goes directly from *S_5* to *S_1* skipping *S_4* and *S_3*. To be able to implement this behaviour the path must be stored together with the candidate clickable and the algorithm needs to be changed not to return to the previous state during recursive crawling but return to the "next-to-examine" state.

Applying this philosophy on our example would result in executing only 3 times a `back` operation instead of 10 times. This way we revisit only 6 states again to return to the correct state instead of 27 states, which is 4.5 times less states that need to be accessed. These improvements are only applicable in certain circumstances. When the state-flow graph is close to a flat tree without much depth the improvement is nearly zero because back-tracking will not happen so often or is not that deep.

In the next section we investigate the performance of the proposed optimisation.

5.3 Evaluation of the optimisation

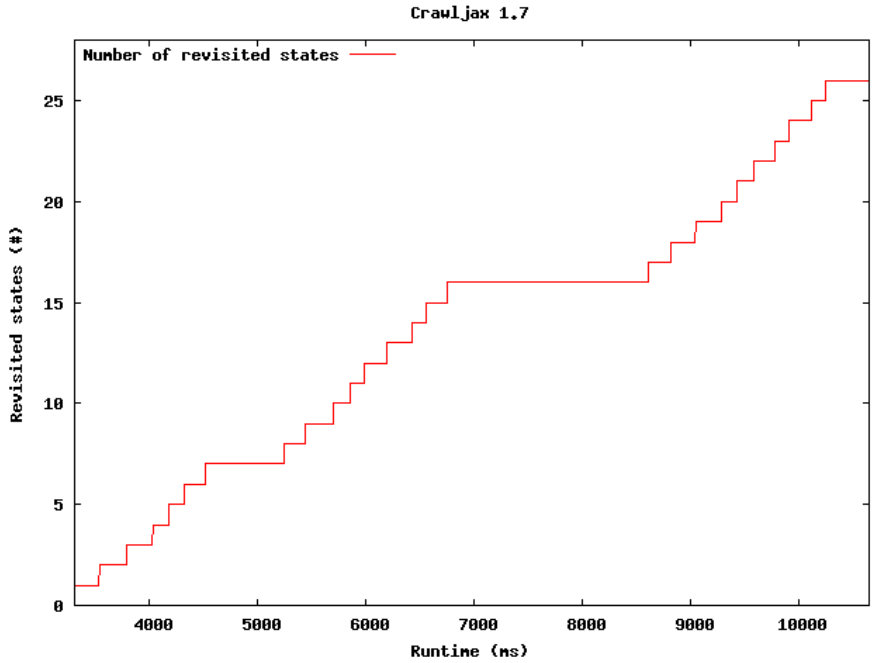
We have implemented this optimisation algorithm in Crawljax 1.8 version to measure the performance gain. Table 5.1 shows the comparison of the results between two versions of Crawljax. The 1.7 version of Crawljax contains the non-optimised version of the back-tracking operation, the 1.8 version of Crawljax contains the optimised version of the back-tracking operation.

Figure 5.2 shows two graphs containing the number of revisited states as a function of time; for the 1.7 and 1.8 version of Crawljax. The data for the graphs is captured

Table 5.1: Comparison of back-tracking implementation

	Crawljax 1.7	**Crawljax 1.8**
Runtime (ms)	25134	16142
Number of Revisited States	27	6
Number of Back-tracks	10	3

by running a simple Ajax application as shown in figure 5.1. In both graphs there is a period of time where there is no increment in the number of revisited states. For the 1.7 version this happens around 5000 milliseconds and between 7000 and 8000 milliseconds. In version 1.8 the same happens between 4000 and 5000 milliseconds and between 6000 and 7000 milliseconds. During those periods new states are found and no back-tracking is active, but before and after such a phase the back-tracking operation is active. Figure 5.2a clearly shows that there are much more states revisited between the two phases in comparison to figure 5.2b.



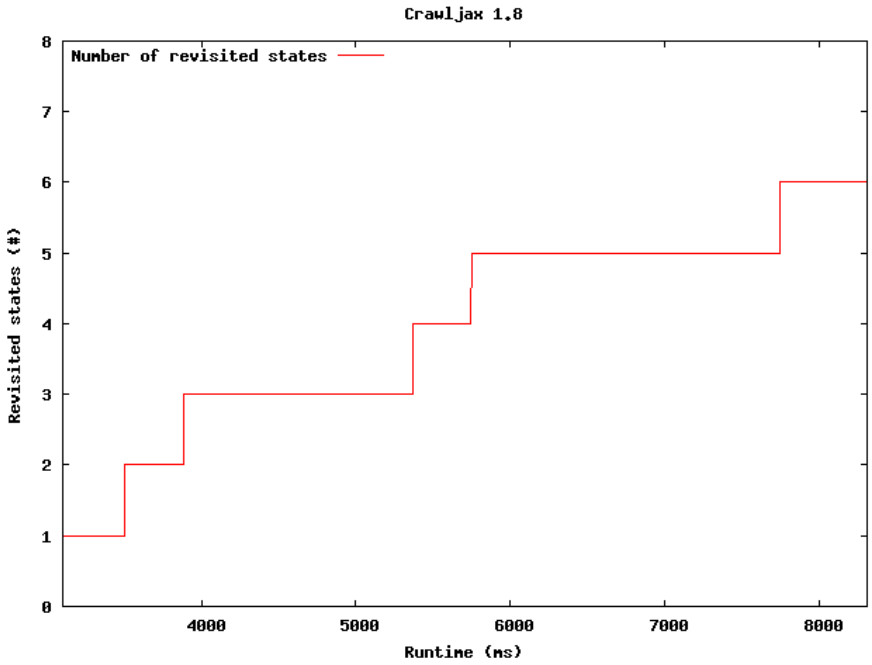
(a) Crawljax 1.7

Figure 5.2: Comparison between 1.7 and 1.8 versions with regard to back-tracking

In the next section we summarise and conclude the implemented optimisation.

5.4 Performance gain

In order to determine if an Ajax application is fully loaded, a waiting period can be executed after every event and page (re)load. This value is by default around 500 milliseconds. The test system displayed in figure 5.1 denotes 11 states. In the 1.7 version,



(b) Crawljax 1.8

Figure 5.2: Comparison between 1.7 and 1.8 versions with regard to back-tracking

according to table 5.1, there are 27 revisited states. That brings the total wait-time to 11 $states + 27$ $revisits = 38$ wait periods. For the 1.8 version, according to table 5.1, there are 6 revisit states, which gives a total wait-time of 11 $states + 6$ $revisits = 17$ wait periods. Figure 5.3 shows the total runtime for the 1.7 and the 1.8 version (left y-axis) as a function of the waiting period shown on the x-axis. The right y-axis shows the performance improvement in percentage. The theoretical performance improvement is $(1 - (17/38)) * 100 \simeq 55\%$. Figure 5.3 shows that the improvement graph is approaching 50% but does not reach it. Not reaching the theoretical performance improvement is caused by the fact that the 1.8 version contains various other updates, which could influence the runtime in a negative fashion.

The speedup gets significant when the wait-time increases. The speedup is, at the typical 500 ms wait-time, about 35%. This speedup is dependent on the structure of an Ajax application. When the state-flow graph of an application results in an *index* state with many edges resulting in states which are final states, the speedup will not be the same as for the system shown in figure 5.1. The typical Ajax application has a non-flat state model and we believe that the back-tracking optimisation will provide a certain degree of performance improvement. Using more processing units concurrently we could even further increase the crawling speed and that is what we will be presenting in the next chapter.

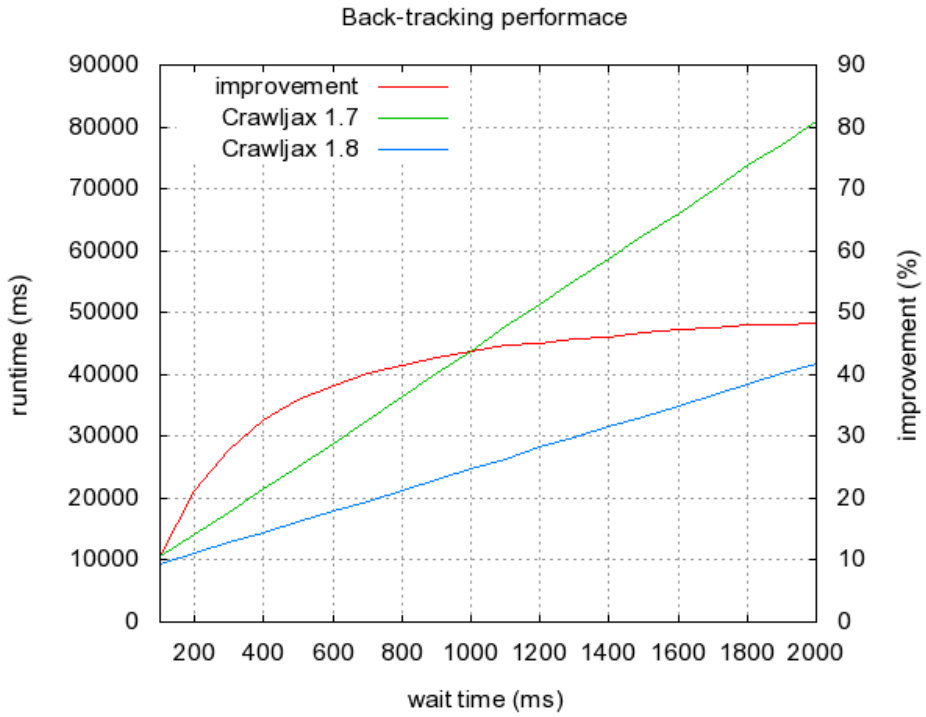


Figure 5.3: Improvement of the back-tracking operation between version 1.7 and 1.8

Chapter 6

Crawling Ajax Concurrently

In this chapter we propose our solution to reduce the state space explosion encountered when crawling Ajax applications. In section 6.1 we determine if the state space explosion, that occurs while crawling Ajax applications, is a memory or a runtime problem. Section 6.2 presents our solution based on using multiple threads and browsers. This section also presents the changes that need to be made to the processing view introduced in section 3.2. In section 6.3 we propose a partition function to be used to distribute the work over the participating threads. Section 6.4 presents the changes that need to be made to the sequential crawling algorithm as introduced in section 3.3.

6.1 Memory versus Runtime optimisation

When running a state space exploration algorithm one of the problems that can be encountered is the state explosion problem [55]. The state explosion problem can be separated in two distinct parts; a *memory* part and a *runtime* part. When the state space algorithm does not fit within the available amount of memory of a workstation, there is a memory problem. When the algorithm does not complete within a given amount of time there is a runtime problem. The runtime problem is relative to the time limit given. The maximum amount of memory, which can be addressed in the workstations today, is 16.8 million terabytes of memory [30]. Due to this theoretical maximum amount of memory we expect that memory problems eventually will disappear due to further hardware development. As discussed in section 4.3 when having a DOM-tree size of 1 MB, a workstation containing 16 GB of memory can handle 4096 states, which we assume is enough to handle enterprise web applications. This assumption only leaves the runtime problems to be solved. New processors introduced are only growing in the number of cores and not any more in the raw core speed, putting a halt on speeding up single threaded single core applications [6, 26, 30]. In this work, we focus on finding ways of reducing the runtime of the crawling process. The next section introduces our solution to the runtime problem.

6.2 Multi-threaded, multi-browser crawling

To perform runtime optimisation the back-tracking has been optimised first, this optimisation is discussed in chapter 5. This optimisation is already a first step, and by

using concurrent computation we can make another optimisation step. Computer vendors today are not increasing the speed of their processors but are only increasing the number of cores [6, 26, 30]. This enforces developers to develop concurrent applications, instead of conventional sequential applications, to take advantage of the multiple cores.

As mentioned in chapter 3 the current web crawler uses only one browser and one main thread. This results in the utilisation of only one core at most and leaving the other cores unused. To utilise these unused core(s) we propose our solution: Multi-threaded, multi-browser crawling. By using multiple threads the work can be executed concurrently on all the cores of the CPU. *Two people can divide the work of cleaning the dinner dishes fairly effectively: one person washes while the other dries. If several more people show up, it is not obvious how they can help without getting in the way or significantly restructuring the division of labour[23].* The same situation exists while crawling, the controller and the browser make a couple in this case. The controller waits for the browser to get ready and the browser waits for the controller for new commands. Adding more threads to the controller while only using one browser will result in blocking operations waiting for the browser to get ready. To solve this problem multiple browsers will be used, the thread-browser combination is the same combination as the washer-dryer combination from the example above. The original controller will be split into a new controller and multiple crawlers. Figure 6.1 shows the new structure of the processing view of concurrent crawling. The controller will be the single main thread monitoring the total crawl procedure. While a single crawler is responsible for doing all operations concerning the crawling procedure and executing the operations in the browser, the controller has no knowledge of the current browser and also has lost the connection to the state machine.

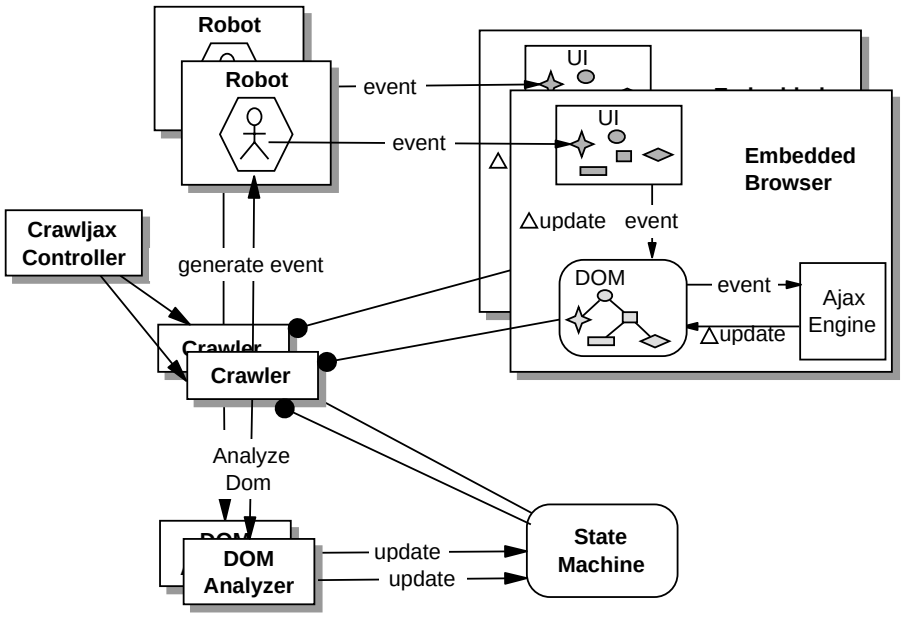


Figure 6.1: Processing view of the concurrent crawling architecture with two threads

Compared with figure 3.2 the new architecture can have multiple crawler instances

running from the same controller; the controller is only responsible for setting-up, starting and finishing the system. The total crawling work is divided over multiple crawlers, which all have their own robot and browser combination.

All the crawlers share the same state machine. The state machine makes sure every crawler can read the states, and through their DOM analyzer update the state machine. The state machine makes sure that no duplicate work is done, clone-states are detected and new states are added.

The operation of discovering new states based on executing events on the current state can be executed in parallel. Since the time needed to retrieve a new state is dominated by the response and the transfer time, the page retrieval and rendering task by the browser takes more time than the analysis of the DOM-tree. It is more efficient to execute the crawling in parallel than doing it sequential [12]. Based on our measurements performed in chapter 4, waiting for the browser takes most of the total runtime. When a request is sent to the browser, the request uses a network connection to communicate with the browser. This introduces some network-traffic. Every request inside the browser might result in a server-side request causing some time to process. To determine if the DOM-tree is changed the DOM-tree is retrieved from the browser, and compared inside the crawler using dynamic analysis. All those operations, sending requests, waiting for the network-traffic and comparing DOM-trees, take some time to execute. During those waiting times other processes can execute their work.

To use the multiple cores and divide the work over the crawlers a partition function must be designed. In the next section we design a partition function to be used with Ajax crawlers. Only when the partition function has been designed the crawling algorithm can be altered to operate concurrently.

6.3 Partition function

As noted in section 2.5, the performance of a concurrent approach is determined by the quality of the partition function [21, 40]. To operate a concurrent state space exploration algorithm, a partition function is necessary that divides the state space over the participating nodes. Let S be the number of states and N the number of processors participating. A partition function can be described as $\pi : S \rightarrow \{1, ..., N\}$, mapping any of the global states to one of the processors participating [40]. A partition function can be static or dynamic. With a static partition function the division of work is known before executing the code. When a dynamic partition function is used, the decision which processor will execute a given state is made at runtime.

6.3.1 Static partition function

Algorithm 2 shows an example application which uses a static partition function. The application first starts with a *Set* of nodes (N) and reads-in its *work* from a file (line 4). When the application processes all the work that needs to be done, the work is divided over the participating nodes using the *partition* function.

The usage of a static partition function ensures that every node, on average, has $work/N$ work elements to process. When every work element takes the same amount of time to process, the work load will be shared equally. If the amount of time to process a single work element varies from element to element, long idle times can

Algorithm 2: Example of a static partition function

```
   input : nodes
 1 Procedure MAIN()
 2 begin
 3 |   Set N ← INITNODES()
 4 |   Set work ← READWORKFROMFILE()
 5 |   for w ∈ work do
 6 |   |   Node n ← PARTITION(w)
 7 |   |   SENDWORKTONODE(w,n)
 8 end
 9 Procedure PARTITION(Work w)
10 begin
11 |   return (GETLINENR(w) mod N)
12 end
```

occur on processors [37]. The use of a static partitioning algorithm has a big advantage: it reduces the message overhead when searching for a node, which handles a given state because it is known in advance [40]. A hash function can be used, for example a bit forwarding hash function [37], which maps a state onto a processor based on the hash of the current DOM.

6.3.2 Dynamic partition function

A dynamic partition function divides the work over processors dynamically during runtime. For example; choose the processor with the smallest work queue to send work to. Dynamic partition functions are often known as on-the-fly partition functions. An example of a dynamic partition function is shown in algorithm 3. Based on the system described in algorithm 2 the partition function can be replaced by this dynamic version. In this dynamic partition function the work is sent to a processor which has the least amount of work to do.

Algorithm 3: Example of a dynamic partition function

```
   input : nodes
 1 Procedure PARTITION(Work w)
 2 begin
 3 |   return (NODEWITHLEASTAMOUNTOFWORK())
 4 end
```

The advantage of such a solution is that, on average, every node is equally busy. It can happen that some node has done much more work than another node, because of an imbalanced distribution of the work by the partition function. Another problem resulting from a dynamic partition function is that a node does not know directly which node is processing a certain state. To find that node, the current node must send messages to all other nodes to find out, causing some overhead [37].

6.3.3 Partition function for concurrent crawling

To crawl Ajax applications concurrently a partition function must be designed that divides all the work over the available processors. This either can be a static or a

dynamic partition function. When a clustered network of workstations is used, a static partition function will give the best results, because it is known in advance which processor is going to handle which states. When a shared memory solution is used, a dynamic partition function will be the best solution. Using such a solution, the workload can be balanced over the threads participating in the computation.

The state-flow graph of an Ajax application is built dynamically during exploration of the Ajax application. Due to this dynamic nature a dynamic partition function is chosen. The task of our dynamic partition function is to distribute the work equally over all the participating nodes.

Before the partition function can be designed we first need to understand what the definition of *work* is. While crawling an Ajax application *work* can be defined as: Bringing the browser back into a given state and explore the first un-explored candidate state from that state. Our proposed solution works as follows: After the discovery of a new state and if there are still un-explored candidate states left in the previous state, that state is sent to another processor for further exploration. The processor chosen will be the processor with the least amount of work left.

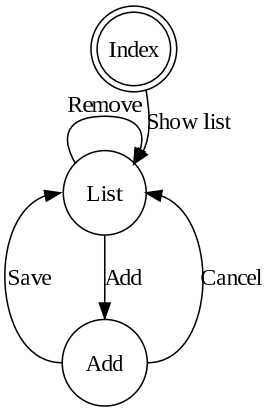


Figure 6.2: Example application `Todo`

Figure 6.2 shows a simple example application called `Todo`. When the application is visited it shows an introduction *index* page, on that page there is a link (*Show list*) to the *List* page. On the *List* page there is the list of items To-Do. On this page a new item can be added by pressing the *Add* button. This will result in a new page, which features an input box and two buttons, *Save*, and *Cancel*. Both those buttons brings the application back to the *List* page. The added todo can be removed by pressing the *Remove* button.

Crawling the example todo application concurrently would be to first load the *index* state and find all target clickables, which in this case is only the *Show list* link. Click on the target clickable and see if the DOM-tree is changed, when the *Show list* link is pressed the list of todo's becomes visible; the DOM-tree is changed. The crawler has made the transaction from the *index* state to the *List* state. In this

state the crawler finds two candidates, the *Remove* button and the *Add* button. Lets assume the *Remove* button is examined first, when executed the DOM-tree changes because a todo item will be removed. The current crawler (crawler_1) will continue to explore this possible new found state, but the *Add* button from the *List* state is not yet examined. To examine this candidate clickable we start a new crawler (crawler_2). The *work* of the crawler will be to go back to the *List* state by reloading the *index* state and clicking on the *Show list* link. When crawler_2 is in the *List* state it fires the *Add* button. Crawler_2 will eventually start a new crawler_3 because crawler_2 detects two candidate elements both resulting in a DOM-tree change.

When the crawling of this example application is finished, 3 crawl paths will be examined by 3 different crawlers:

1. $Index \rightarrow List \rightarrow Add \rightarrow (Save)List$
2. $Index \rightarrow List \rightarrow Add \rightarrow (Cancel)List$
3. $Index \rightarrow List \rightarrow (Remove)List$

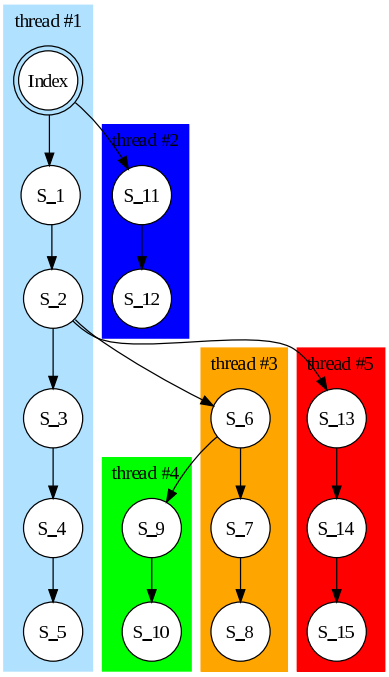


Figure 6.3: Partition function for concurrent crawling, having 5 partitions

Figure 6.3 shows the application of the partition function for concurrent crawling on another example application. In the index state (*index*) two candidate states are found: *S_1* and *S_11*. The initial thread continues with state *S_1*, *S_2*, *S_3*, *S_4* and finishes in *S_5* (Depth-first [41]), and a new thread is simultaneously branched off that

will have to explore state *S_11*. This new thread first reloads the *index* and after that goes into *S_11*. In state *S_2* and *S_6* this same branching also happens, resulting in a total of 5 threads. These threads and their corresponding partitions are coloured red, blue, lightblue, orange and green.

Also a dynamic partition can be created that keeps the crawling procedure in a Depth-First manner. This can be achieved by maintaining two queues; one with free processors and one with states to process. When a new state is found it is placed in front of the queue. When a processor becomes available it takes the head of the queue and continues from there. So the queue behaves as an FiLo (First in Last out) queue, often also known as a Stack.

Now that the partition function has been introduced, the original sequential crawling algorithm can be changed into a concurrent version.

6.4 Concurrent crawling algorithm

Based on the processing view as defined in figure 6.1 and the partition function, we can adapt the sequential crawling algorithm 1 introduced in section 3.3. The proposed changes include:

CH1 Splitting the `controller` into a new `controller` and multiple `crawlers`.

CH2 Moving the `robot` and `browser` from the `controller` to the `crawler`.

CH3 Implement the new designed partition function.

Algorithm 4 shows the new concurrent crawling algorithm, in which we performed changes **CH1**, **CH2**, and **CH3**. We started the change to the crawling algorithm with **CH1**, the extraction of the `crawler` from the `controller`. This resulted in a new procedure called *RUN* (line 7 to 18). The *RUN* procedure takes a *State* and an *EventPath* as arguments. The *State* specified is the state in which an candidate element must be examined. To reach this state the *RUN* procedure uses the *EventPath* to back-track into the instructed state state (line 12 to 16). The *RUN* procedure can be executed concurrently, it is started at the begin of earch crawl path. For every thread shown in figure 6.3 the *RUN* procedure is executed, e.g., the red coloured thread #5 is explored by executing *RUN* with *S_2* as *State* and $index \rightarrow S\_1 \rightarrow S\_2$ as *EventPath*.

After **CH1** was completed we continued with **CH2**, the `robot` and `browser` are moved to the *RUN* procedure because every concurrently running thread needs access to its own browser (line 10 to 11). To be able to share `browsers` between consecutive invocations of the *RUN* procedure we introduce a `browserPool` (line 4). This pool manages the available browsers, on every invocation of the *RUN* procedure a `browser` is requested (line 10).

The final change we applied, **CH3**, was the addition of the *PARTITION* procedure (line 43 to 46). This procedure is executed with a *State* as argument. For every not yet examined clickable (line 46) a new `crawler` will be created with the given *State* and the *ExactPath* to that state as arguments (line 47). Once the crawler is created, it is distributed to the first available executing thread (line 48). As multiple crawlers can explore the same state at the same time they must mark the candidate clickable as

Algorithm 4: Concurrent AJAX Crawling

```
   input : URL, tags, browserType, nrOfBrowsers
 1 Procedure MAIN()
 2 begin
 3     global sm ← INITSTATEMACHINE()
 4     global browserPool ← INITBROWSERPOOL(nrOfBrowsers, browserType)
 5     crawler ← CRAWLER()
 6     crawler.RUN(null,null)
 7 end
 8 Procedure RUN(State s, EventPath ep)
 9 begin
10     local browser ← browserPool.GETEMBEDDEDBROWSER()
11     local robot ← INITROBOT()
12     browser.GOTO(URL)
13     for e ∈ ep do
14         re ← RESOLVEELEMENT(e)
15         robot.ENTERFORMVALUES(re)
16         robot.FIREEVENT(re)
17     CRAWL(s)
18 end
19 Procedure CRAWL(State ps)
20 begin
21     cs ← sm.GETCURRENTSTATE()
22     Δupdate ← DIFF(ps, cs)
23     f ← ANALYSEFORMS(Δupdate)
24     Set C ← GETCANDIDATECLICKABLES(Δupdate, tags, f)
25     for c ∈ C do
26         SYNCH(c)
27         begin
28             if cs.NOTEXAMINED(c) then
29                 robot.ENTERFORMVALUES(c)
30                 robot.FIREEVENT(c)
31                 cs.EXAMINED(c)
32                 dom ← browser.GETDOM()
33                 if STATECHANGED(cs.GETDOM(), dom) then
34                     xe ← GETXPATHEXPR(c)
35                     ns ← sm.ADDSTATE(dom)
36                     sm.ADDEDGE(cs, ns, EVENT(c, xe))
37                     sm.CHANGETOSTATE(ns)
38                     PARTITION(cs)
39                     if STATEALLOWEDTOBECRAWLED(ns) then
40                         CRAWL(cs)
41                     sm.CHANGETOSTATE(cs)
42         end
43 end
44 Procedure PARTITION(State cs)
45 begin
46     while SIZEOF(cs.NOTEXAMINEDCLICKABLES()) > 0 do
47         crawler ← CRAWLER(cs, GETEXACTPATH())
48         DISTRIBUTEPARTITION(crawler)
49 end
```

examined (line 31), and only process a candidate element which is not yet processed (line 28). To prevent multiple crawlers to accidentally perform the same operation the body of the foreach loop is protected by a synchronised block (line 26 to 42).

The sequential crawling algorithm has been altered into a concurrent version; the crawler is separated from the controller, the back-tracking is executed first and the notion of un-explored candidate elements is introduced. In the next chapter the changes are implemented, and the concurrent crawling algorithm is applied to the research implementation tool called Crawljax.

Chapter 7

Implementation

In this chapter we present the implementation of the concurrent crawling algorithm, as defined in algorithm 4 and explained in chapter 6. We refer to this new implementation as Crawljax 2.0. The first section gives an introduction on the development of Crawljax during our research. Section 7.3 introduces the notion of executable tasks and reflects on the work performed to get the tasks executed and working correctly. Section 7.4 gives details about the way we handle the creation and administration of multiple browsers. Section 7.5 introduces the techniques used to achieve synchronisation and prevent race-conditions inside Crawljax. In the final section 7.6 we present how we are able to be backward compatible with the previous versions of Crawljax.

7.1 Crawljax 2.0

We started our research with the 1.7 version of Crawljax. This was the version in which we detected the back-tracking problem (chapter 5), and the improved back-tracking algorithm was introduced in, and released as the 1.8 version. The 1.8 version was also the version in which we started our work on concurrent crawling. The 2.0 version of Crawljax contains the concurrent implementation and all its configuration options.

To implement the concurrent crawling in the 2.0 version, the following steps were needed:

- To crawl concurrently, executable tasks must be defined, separated from the original core and executed using a task-execution framework.
- All the threads must be able to use multiple browsers, request them and release the browsers again.
- The 2.0 version of Crawljax the crawling order must be the same as in the previous versions, also the state-flow graph also must be exactly the same when using only one thread. To achieve those goals the code must be synchronised.

To achieve those steps, we first introduced a new term and datastructure which are described in the next section.

7.2 CrawlAction

In the 1.9 version a new data structure was introduced; *CrawlAction*. In the 1.8 version two for-loops where implemented; one looping over all the candidate elements found in a state, the second loop, nested in the first loop, loops over all the possible events like; mouse clicks, form submits and mouse hovers. For the single threaded version this is sufficient. When multi-threading was introduced this became too complicated and a new data structure was introduced called *CrawlAction*. This data structure is the combination of a candidate element and an event. Objects of this type could be used to denote what event must be performed on which candidate element.

In the next section we define Tasks to be used to execute a single *CrawlAction*.

7.3 Tasks

There are two approaches possible in implementing multi-threaded applications. The first approach is **Task-based** and the second one is **Actor-based** [39]. Task-based threads are threads which execute a single method or an entire session. Actor-based threads are threads which operate more autonomously and are most of the time found in daemon like operations. In Crawljax 2.0 we decided to use a task-based approach. This was done because the partition function shown in figure 6.3 results in a single instruction; back-track to the given state and start crawling from that state. Everytime a new branch is detected the partition function defines a new task. Before the Task-based approach can be implemented the code must be separated into clear tasks. When tasks have been defined, the original code will be split-up and integrated into the different already existing architectural components.

7.3.1 Defining a Task

Within Crawljax 1.8 there was no clear code partition that could be considered as Task-based nor Actor-based [23]. The new processing view, figure 6.1, provides an idea about a part that could be transformed into a Task-based multi-threaded application. The crawler will be refactored to be able to execute a given task.

In section 6.3.3 a partition function was designed to separate the state space into partitions and distribute those over the participating nodes. Figure 6.3 shows the crawl paths of an example state-flow graph, where a partition can be defined as a crawl path which needs to be explored. Every unique crawl path from the resulting state-flow graph will be discovered by a unique crawler. The crawler first back-tracks into the state where it needs to execute the corresponding *CrawlAction*. As every *CrawlAction* can result in a state change; for every *CrawlAction* a new crawler will be created.

To summarise, the task of the crawler is to:

1. Back-track into its start-state.
2. Execute its assigned *CrawlAction*.
3. Continue with the crawl-procedure.

The crawler class is designed as an implementation of a *Runable* class. This enables the execution of crawlers within threads. The executed method of a *Runable* class

is its *run* method. This method is executed by the executing thread. All the initialisation, back-tracking, execution of its assigned *CrawlAction* and the further crawling is performed from that *run* method.

Now that a task has been defined as executing a crawler by running its *run* method, the actual work can be extracted from the original controller and merged into the crawler.

7.3.2 Splitting tasks

The crawler needs a browser to perform its crawling, in Crawljax 1.8 those browsers were started and controlled by the controller. Keeping the controller responsible for controlling the browsers, while moving all the crawling operations to the crawler would make Crawljax architectural complex. When in this case a crawler wants to execute an operation on a browser the request must be placed to the controller, while the controller must keep track on which browser and which request must be executed.

In algorithm 4 the *browser* and *robot* variables are moved from the **global** scope to the **local** scope of the *RUN* procedure. In Crawljax this is done by moving the browser from the controller into the crawler class. This move revealed all the dependencies of the controller on a single browser instance. All those dependencies were moved from the controller to the crawler to resolve them. When there are no outbound dependencies on shared variables inside the crawler, the crawler and its corresponding browsers objects can be executed in parallel.

All the operations, which are executed on a single instance variable or operations which are mutual exclusive operations are moved or maintained inside the controller. Every crawler has access to the controller to execute the shared operations. This makes the controller responsible for managing the synchronisation, leaving the crawler unaware of synchronisation issues.

As the crawler is extracted from the controller the crawlers can now be executed, and this is done by using threads to run the *Runable* crawler.

7.3.3 Executing tasks

Threads are a mechanism by which tasks can run asynchronously, whereby tasks are abstract, logical, discrete units of work. Dividing the work of an application into tasks simplifies program organisation, facilitates error recovery by providing natural transaction boundaries, and promotes concurrency by providing a natural structure for the parallelisation of work. Ideally, tasks are independent activities; work that does not depend on the state, result or side effects of other tasks [23]. The separation of the controller and crawler discussed above makes it possible to execute a task: running the crawler.

There are two possible solutions to run these tasks; as a thread-per-task approach or using the pool threads approach [23, 39]. Executing tasks in pool threads has a number of advantages over the thread-per-task approach. Re-using an existing thread instead of creating a new one reduces thread creation and teardown costs over multiple requests. By properly tuning the size of the thread pool enough threads can be available to keep all the processors busy. By running all the threads at once the application could run out of memory or crash due to competition for resources among the threads.

A pool threads approach is implemented using the *ThreadPoolExecutor* class supplied by the default Java JVM from version 1.5 onwards. The *ThreadPoolExecutor* performs all the operations regarding the creation of threads, and making sure the maximum number of threads is running.

The *ThreadPoolExecutor* provides the following protected over-ridable methods designed for extension:

- `beforeExecute()`, is called before the execution of each task.
- `afterExecute()`, is called after execution of each task.
- `terminated()`, can be overridden to perform any special processing that needs to be done once the *ThreadPoolExecutor* has fully terminated.

The `beforeExecute` and `afterExecute` methods can be used to manipulate the execution environment; for example, re-initialising *ThreadLocals*, gathering statistics, or adding log entries.

To be able to extend the *ThreadPoolExecutor* operations a new class is added, the `CrawlerExecutor`. In this class the `beforExecute` extension point is used to prepare the logging. The `afterExecute` is used to determine the termination of the Crawling process. This is the case when there are no crawlers running anymore and there is no more work left in the queue.

The `CrawlerExecutor` is also extended with a `waitForTermination` call. This enables the *CrawljaxController* to halt until crawling is finished. The blocking operation is implemented using a `wait` on the **this**-object of the `CrawlerExecutor`, and when the `terminated` is executed a notify message is sent to the **this**-object to unlock the *CrawljaxController*. By using this system the controller can wait for the executor to be finished before continuing.

The `CrawlerExecutor` also takes care of keeping all the processors busy. When a new crawler is added via its `execute(`*Work*`)` operation and there is a free thread available the crawler is started on that thread. If there is no free thread available the work is stored in a *BlockingQueue* data structure for later scheduling. In algorithm 4 this operation of keeping all the processors busy and scheduling work is expressed by a single call, *distributePartition*.

The `CrawlerExecutor` only knows how to execute tasks when they are entered into its queues. To start the crawling of an Ajax application an initial-crawler must be entered into the `CrawlerExecutor`.

7.3.4 Initialise crawling

To initialise the crawling procedure, Crawljax opens the index page to search for clickables [41]. After the index state, the corresponding DOM-tree, the state machine, state-flow graph and crawl session can be created. This all is done by a special crawler called the initial-crawler.

The initial-crawler extends the normal crawler execution by performing some extra operations before its normal processing. Algorithm 5 shows the operations performed before the normal procedure starts. This *RUN* procedure overwrites the *RUN* procedure defined in algorithm 4. The inital-crawler first goes to the index page to build

Algorithm 5: Tasks of the initial-crawler

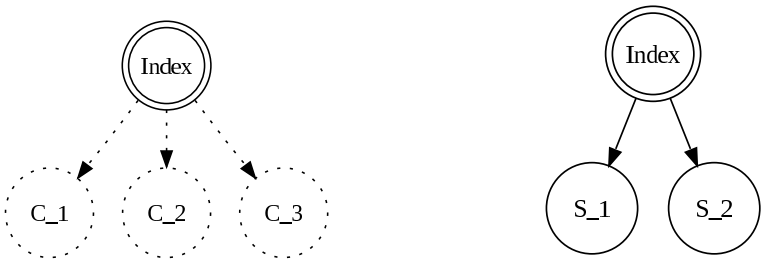
```
   input : crawler
 1 Procedure RUN()
 2 begin
 3 |   goToInitialURL()
 4 |   i ← State(index, getDom())
 5 |   g ← Graph(i)
 6 |   sm ← StateMachine(g, i)
 7 |   s ← Session(sm)
 8 |   Crawl(i)
 9 end
```

the *index* state (line 3 to 4). When the *index* state is found the new state-flow graph can be created. The state-flow graph can not be created earlier because it needs the *index* state (line 5). When the state-flow graph is available the *StateMachine* and *Session* can be created (line 6 to 7). Those three Objects all depends on the *index* state. When everything is setup the *Crawl* procedure from algorithm 4 is executed continuing with the normal crawling procedure. The initial-crawler behaves as a normal crawler and collaborates in the crawling process. It will be the first crawler to detect possible clickables and it will instantiate new crawlers to assist in the crawling.

The crawler is able to initialise itself, distribute its tasks over the participating nodes and crawl an Ajax application. The last thing that needs to be done is to prevent the crawler from executing duplicate or too much work.

7.3.5 Preventing duplicate work

As discussed in section 7.3.1, for every *CrawlAction* a new crawler is added to the `CrawlerExecutor`.



(a) Candidate elements

(b) Final state-flow graph

Figure 7.1: System progress from candidate elements to states

Figure 7.1 shows the crawling of a sample system. In figure 7.1a the candidate elements found from the *Index* state are shown as dotted circles C_1, C_2 and C_3. After the crawling is finished the resulting state-flow graph is shown in 7.1b. Lets assume that candidate C_1 results in state S_1, candidate C_3 results in state S_2, and

candidate *C*_2 did not result in a state change. After the *Index* state the following crawlers and theire work where added to the `CrawlerExecutor`:

- Crawler_1; Reload *Index*, click on *C*_1 and continue crawling.
- Crawler_2; Reload *Index*, click on *C*_2 and continue crawling.
- Crawler_3; Reload *Index*, click on *C*_3 and continue crawling.

When running with one thread crawler_1 will discover state *S*_1 and stop crawling because it can not find any more candidate elements. After crawler_1, crawler_2 will start crawling by clicking on *C*_2, and *C*_2 will not result in a DOM-tree change. There are two possible scenarios for crawler_2; one is to terminate at this point and let crawler_3 continue. The second option is to let crawler_2 continue with candidate *C*_3. This is possible because the DOM-tree is not changed so crawler_2 is still in the *Index* state and *C*_3 is unexplored. When crawler_2 examines *C*_3 it first removes crawler_3 from the queue.

Running with only one thread this optimisation operates correctly but when running with multiple threads this can lead to errors. When for example crawler_2 decides to examine *C*_3 and wants to remove crawler_3 from the queue, but crawler_3 was already started both crawler_2 and crawler_3 can end up examining *C*_3. To prevent these kind of problems some measures are taken. Inside each state, the state maintains four lists:

- **candidateActions**. This is the full list of *CrawlActions* that needs to be examined. This list is built when the state is explored for the first time.
- **registerdCandidateActions**. This list contains the combination of a crawler and a CrawlAction. Basically it denotes which crawler is registered to execute which CrawlAction. When a new crawler is registered a CrawlAction from the **candateActions** list is transferred into the **registerdCandidateActions** list together with the new registered crawler. The CrawlAction will be removed from the **candateActions** list when transferred.
- **workInProgressCandidateActions**. This list denotes which crawler is executing which CrawlAction at the moment.
- **registeredCrawlers**. This is the complete list of registered crawlers for this state which are not yet started.

When a crawler requests a (new) *CrawlAction*, the following checks are made:

1. If already a CrawlAction is registered for the requesting crawler return that *CrawlAction*.
2. Are there any unregistered CrawlActions left in the candidateActions list return the first non-registered CrawlAction
3. Find the last registered crawler, remove it from the `CrawlerExecutor` waiting queue. Remove the registration for the crawler from the waiting queue and return the *CrawlAction* to the requesting crawler.

During the last check work can be deleted, and it could theoretical be possible that the crawler is just started executing. So before a crawler starts back-tracking it checks if the work it is about to execute is not already executed or in process of execution by another crawler. If the work is stolen during back-tracking, the crawler whose work is stolen will request a new *CrawlAction*.

This whole procedure ensures that every *CrawlAction* is executed only once, and every crawler is used as efficient as possible. The next section describes how a crawler gets hold of a browser.

7.4 Multiple-browsers

We are now able to execute crawlers as tasks concurrently using the `CrawlerExecutor` framework while not performing any duplicate work. A crawler operates on one single browser and it is dependent on the state of that browser [41]. During the life time of a single crawler it acquires a single browser instance. After the crawler is finished crawling it releases its acquired browser. This results in the creation of a new browser for every crawler executed. To optimise this operation a new structure is introduced: the *BrowserPool*. When a crawler is finished the browser instance can be re-used again. All started browsers are kept in a pool of browsers to be re-used. This reduces start-up and shut-down costs to only once. The *BrowserPool* is the main manager of all running browsers. It can be queried for a browser instance, and when a crawler is finished working with a browser it must release the used browser by telling the *BrowserPool* that it has finished. The *initEmbeddedBrowser* and *initRobot* procedures from algorithm 4 are implemented inside the *BrowserPool*.

The first approach was to let every crawler request a browser instance when needed, and return a free instance or create a new browser instance if there are no browsers available. This approach was an on-demand approach, which kept the crawler waiting for the instantiation time of the browser, taking approximately 4 seconds. We optimised this approach by sequentially starting all other browsers inside the *BrowserPool* after the first browser was requested, as explained in the next subsection.

7.4.1 Browser Booting

Figure 7.2 shows the procedure used to request a browser inside the *BrowserPool*. The requesting procedure is built around two central data components: the list of available browsers and the list of taken browsers. If a browser is distributed to a crawler, the browser is removed from the list of available browsers and added to the list of taken browsers. If a browser allocation is released the browser is transferred from the taken-list to the available-list. The *BrowserPool* is started with both lists initialised empty.

When a browser is requested it is checked first if the booting procedure is active. If the booting procedure is not in use and there is no browser available a new browser instance is created. If there is a browser available that instance is returned and removed from the list. If at the beginning the browser booting is enabled and the browser-pool is not fully booted, i.e., the configured number of browsers are not yet created, the booting procedure is started if it is not already running. The booting of the browsers is a loop over the number of browser instances that have to be created and every new browser instance is added to the available list. The available list is a standard

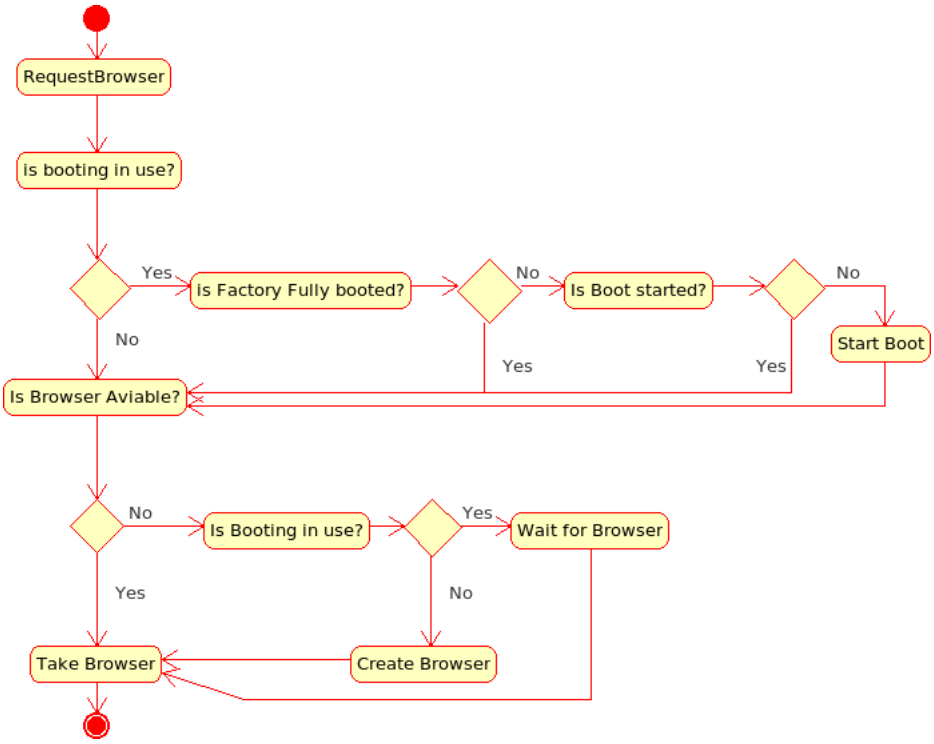


Figure 7.2: Browser booting procedure

Java *BlockingQueue*. A queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. The *BlockingQueue* also supports a fairness policy for ordering waiting producer and consumer threads, granting threads access in FIFO order. This enables the first thread requesting a browser to receive the first created browser.

After implementing the system as shown in figure 7.2 the total boot-up times of all the browsers is still the sum of starting the individual browsers, but Crawljax can already start crawling after the first browser instance is created. When more threadable work is found, the browsers can be used as they come available. This procedure is by default enabled in the configuration but can also be disabled; falling back to the on-demand way of requesting browsers.

The browser booting code can be regarded as an Actor-based implementation [23, 39]. Once the booting is started it operates as a stand-alone daemon, adding browsers to a queue to be used when the browsers are started. The daemon is started once the first request for a browser comes in, it finishes when it is ready starting the number of configured browsers.

The operations on the *BrowserPool* are executed by all the participating threads. The order of those operations is undefined, and because there is only one instance of the *BrowserPool* the state of the *BrowserPool* must be guarded using synchronisation. In the next section the different concepts of synchronisation used are explained.

7.5 Synchronisation

In algorithm 4 most of the code is protected by the *SYNCH* block. This is done to prevent multiple concurrently running threads to make changes to shared memory. To implement the *SYNCH* block we first need to understand synchronisation and after that determine what part of Crawljax is shared between the threads.

Sometimes statements as `counter++;` might look as a single operation but are instead three separate operations. First the current value of `counter` is read from memory, secondly the value from memory is increased with one and as the last operation the value is written back to memory. The correct, thread-safe, operation would have been `synchronised(counter){ counter++ }`. With this statement the lock of counter must first be retrieved (`synchronised(counter)`). When the lock is granted the code from thread 1 continues executing and thread 1 will be the only thread currently allowed to access `counter`. When thread 1 has finished, the lock will automatically be released.

When classes contains operations (logical or physical) that are not atomic, race conditions can occur. Leading to two distinct kind of storage conflicts:

- **Read/Write conflicts.** Thread 1 reads a value, "a", needed for its computation. Directly after thread 1 has read value "a" thread 2 writes a new value into "a". Thread 1 now is performing calculations on old, incorrect data. The calculation performed by thread 1, does not reflect the actual state of the system. This is also called an "inconsistent read" [17].

- **Write/Write conflicts.** Two different threads try to write the same field at the same time, one of the write actions is lost and the result is impossible to predict. This is also called a "lost update" [17].

Race conditions occur when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime environment, i.e., the right answer relies on lucky timing.

Imagine two students decided to meet up at the exit of the college building at the end of a college. The first student, student-a, arrives at the main exit and does not find the other student, student-b. At that moment student-a remembers there are two exits in the building (main exit and the side exit), after some period he moves to the side exit. At the side exit student-a does not find student-b either. There are a few scenarios possible:

- Student-b has had a delay and was not at either exits.

- Student-b arrived at the main exit after student-a left the main exit.

- Student-b was at the side exit and did not find student-a and went to look for student-a at the main exit.

Unless both students have agreed on a "recovering" protocol they could be walking back and forth forever trying to find each other. The desired outcome of the process of finding each other depends on the relative timing of events (how long does a student wait at an exit, etc). This problem can be solved by allowing only one student to walk at a time (using locking / synchronisation).

It is easier to design a program with concurrency in mind than to retrofit concurrency later [14]. The first thing to determine when retrofitting concurrency in existing applications is to determine the shared states. These shared objects can be found by following all the references accessed from within a thread. The only references that need to be followed are the references, which are shared between all possible running threads. When all accessed shared classes are either immutable or thread-safe and all created objects during life time of a thread are side effect free (synchronised usage of non-final static members) the whole application is said to be thread-safe.

Crawljax 2.0 uses shared memory to store the state-flow graph, i.e., the state-flow graph is shared with every thread. The shared memory must be guarded so that it is not violated by multiple threads accessing the data concurrently. Therefore the shared data classes must be thread-safe. A class is considered thread-safe if it behaves correctly when accessed from multiple threads, i.e., the calling code nor the runtime environment performs any synchronisation or other coordination [23]. For a class to be thread-safe operations on the class must operate atomic, i.e., the operation must execute as a single, indivisible operation.

To protect against race conditions, and enable atomicity, there are two structures available: synchronisation and locking. The internal synchronisation mechanism of Java uses a built-in lock available in every object, the intrinsic lock. When writing synchronised functions the internal lock of the object is used, but when the state of an application is composed of multiple objects the intrinsic lock can not be used to guard that state. A separate special locking object needs to be instantiated. The intrinsic lock of that object can be used to guard the composed state. Adding synchronised blocks to prevent un-synchronised changes introduces overhead costs and can result in large waiting times. The overhead can be avoided for objects known as immutable [50]. An object is immutable if its state is never changed after initialisation [3, 50]. The initialisation of the state of the immutable object is done in the constructor of the object. After the object has been created the state can never be changed, i.e., all the non-static member fields of the class are defined final.

When a shared data component needs to store a variable, which is related to only one thread, *ThreadLocals* can be used. Each thread holds an implicit reference to its copy of a thread-local variable. The thread-local is implemented as a HashTable using the thread as the key. As long as the thread is alive the *ThreadLocal* instance is accessible. After a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist). In the *BrowserPool* the current browser belonging to a thread is stored, and when a plugin wants to access the current browser a call is made to the *BrowserPool* to retrieve the browser used by the current Thread. Using the *ThreadLocal* mechanism prevents the use of synchronised code.

To find all the shared code, we followed all the dependencies on other objects accessed from the crawler, which are not created from within the crawler. When a new crawler is created a single object, the *Controller*, is stored as a reference. We used the Eclipse[1] Java IDE, and temporary removed the reference to the the *Controller*. All the compilation errors reveiled the operations performed on the *Controller* that need to be checked. Every access to the *Controller* is potentially not thread-safe, and we traced

[1] `http://www.eclipse.org/`

every method execution on the *Controller* object. For every method execution it was determined if the method was thread-safe or which part needed to be synchronised.

The Java 6 Platform[2] supplies a collection package that implements collections designed for use in multi-threaded contexts: The concurrent collection. All implementations offered, offer concurrent access without any synchronisation. These implementations can be used without any worry about thread interference. Most of the classes implement an efficient "wait-free" algorithm based on the work presented by Michael et al. [43]. Using those collections offers the ability to access the data concurrently without any synchronisation. In Crawljax we implemented these collections in the *BrowserPool* to keep track of the free and taken browsers. For the list of free browsers we used the *ArrayBlockingQueue* to be able to keep the crawlers halted when they requested a browser or when there was no browser available yet. The other location where the concurrent collections were used is in the state to store the *CrawlAction* together with the crawlers.

Knowing which part of the software is thread-safe, we needed to document that information.

7.5.1 Documenting thread safety

To document the thread safety in classes and their fields; the following annotations are designed for the Java programming language [23]:

- @ThreadSafe. Documents the class as thread-safe.
- @NotThreadSafe, class is not documented as thread-safe or class is found to be not thread-safe and document it explicitly.
- @GuardedBy(lock). Field or method is guarded by the given lock.
- @Immutable. The documented class is immutable and implies @ThreadSafe.

These annotations can be used to document thread-safety promises and synchronisation policies. There are three types of annotations; **Class**, **Field** and **Method** annotations. The @Immutable, @ThreadSafe and @NotThreadSafe are **Class** annotations while the @GuardedBy(lock) annotation is only used to document a **Field** or a **Method**.

Class annotations are relatively un-intrusive and are beneficial to both users and maintainers. Users can see immediately whether a class is thread-safe, and maintainers can see immediately whether thread-safety guarantees must be preserved [23]. These class annotations are part of the public documentation while the field and method annotations are not part of the public documentation and will only be used and read by the maintainers.

@GuardedBy(lock) documents that a field or method can only be accessed when the specified lock is held. The following list of possible arguments to @GuardedBy is commonly used:

[2] `http://java.sun.com/javase/6/`

- @GuardedBy("this"). Meaning the synchronisation is done on the **this** object (on methods this implies the synchronised keyword, on fields this implies the synchronized(this).. construction).
- @GuardedBy("fieldName"). The fieldName object is used for synchronisation or locking.
- @GuardedBy("ClassName.fieldName"). Like @GuardedBy("fieldName"), but referencing a lock object held in a static field of another class.
- @GuardedBy("methodName()"). The lock object that is returned by calling the named method.
- @GuardedBy("ClassName.class"). The class literal object for the named class.

7.6 Backwards compatibility

One of the key requirements of the new multi-threaded design was to maintain backward compatibility with the single threaded version of Crawljax. In the original version of Crawljax the states are processed in a Depth-First manner [41]. This behaviour must be maintained when running the concurrent version with only one thread. In section 6.3.3 we introduced the notion of maintaining two queues one with the processors and one with the work. The queue with processors is maintained by the *CrawlerExecutor*.

When crawling concurrently, a queue of tasks is used by the `CrawlerExecutor` to keep track of the unexplored work. When using a queue implementation, the crawling will result in a breath-first search. This is because in every state the list of found *CrawlActions* is added at the end of the queue. So the deepest state is at the bottom of the queue while the first unexplored state is at the top of the queue. When processing the queue the state space is explored "level-by-level".

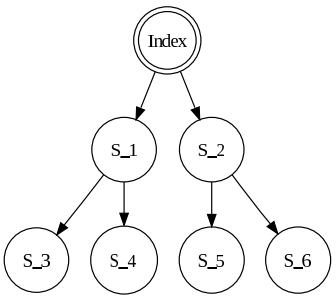


Figure 7.3: Example state-flow graph showing 7 states

Figure 7.3 shows an example state-flow graph, and when exploring this state space using depth-first traversal the order in which the states are found will become: *index*, *S*_1, *S*_3, *S*_4, *S*_2, *S*_5 and *S*_6. If a breath-first search is performed the ordering would be: *index*, *S*_1, *S*_2, *S*_3, *S*_4, *S*_5 and *S*_6.

The default work-queue implementation used inside the `CrawlerExecutor` is the *LinkedBlockingQueue* which is a queue which orders elements FIFO (first-in-first-out). The head of the queue is the element that has been on the queue the longest time. The tail of the queue is the element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue [23].

Table 7.1: State of the work-queue after every CrawlAction for both a Queue and Stack implementation

Queue	**Stack**
index: [S_1, S_2]	index: [S_1, S_2]
S_1: [S_2, S_3, S_4]	S_1: [S_3, S_4, S_2]
S_3: [S_2, S_4]	S_3: [S_4, S_2]
S_2: [S_4, S_5, S_6]	S_4: [S_2]
S_5: [S_4, S_6]	S_2: [S_5, S_6]
S_4: [S_6]	S_5: [S_6]
S_6: [⊘]	S_6: [⊘]

Table 7.1 shows the development of the work-queue over time where, it displays the contents of the work-queue after the found *CrawlActions* are added. When there is a search for *CrawlActions* inside a state a temporary queue is created to store the found *CrawlActions*. Once the exploration is finished the complete contents of the temporary queue is added to the work-queue. As an optimisation the crawler continues with the *CrawlActions* stored in the current state if one of those resulted in a new state the new state will be examined. This procedure prevents a real breath-first processing of the state space. To cope with this problem a custom queue implementation has been made, that is based on a stack implementation. A stack is LIFO (last-in-first-out) and results in preserving the depth-first behaviour. This is also visible in table 7.1, where the queue implementation results in the order: *index*, *S_1*, *S_3*, *S_2*, *S_5*, *S_4* and *S_6*. While the stack implementation in table 7.1 shows the ordering: *index*, *S_1*, *S_3*, *S_4*, *S_2*, *S_5* and *S_6* which is depth-first. Using a stack instead of a queue results in a reproducible state ordering while exploring the state space. This only is true when running with only one thread active; when running with multiple threads the order of states is undefined. The state-flow graph will be identical but the ordering in which states are found different on every crawl execution.

The 2.0 version of Crawljax includes tasks (crawlers) which can be executed individually by the *ThreadPoolExecutor* framework. The *BrowserPool* is used to manage the multiple browser instances. Every crawl operation is executed in a thread-safe mode using synchronisation. When running with one thread Crawljax 2.0 crawls in a backwards compatible mode. In the next chapter we investigate if the changes that are proposed and implemented result in improvements.

Chapter 8

Evaluation

In this chapter we perform an evaluation of our concurrent crawler as proposed in chapter 6 and the implementation described in the previous chapter. In the first section we repeat our previous performed experiment from chapter 4 to see the improvement results on performance using a concurrent approach. In section 8.2 we will present the results of a case-study performed at Google London on their Adsense[1] software. Section 8.3 presents the results of a case-study performed at Hostnet[2] on their customer portal Mijn Hostnet[3].

In this chapter we also try to find the answer for our last research question: How effective are the proposed improvements? Besides this research question we formulate the following evaluation questions:

EQ1 Has our concurrent implementation made Crawljax more scalable?

EQ2 Is the site-coverage of Crawljax the same when using the concurrent implementation?

In this chapter we investigate if the changes made Crawljax approach the theoretical improvement. The theoretical improvement can be defined by the following formula: $T_{min} = \frac{1}{n} \times T_1$. This formula calculates the minimal runtime given by T_{min}, for a given number of threads (n) and the runtime while running sequentially (T_1). Beside this theoretical improvement we also define the speedup; $speedup = \frac{time_{seq}}{time_{par}}$ the $time_{seq}$ is the time used while running sequentially, and $time_{par}$ denotes the time used running concurrently for a given number of threads.

8.1 Measuring performance

In chapter 4 we performed a measurement to analyse the performance of Crawljax. In this section we repeat our experiments, but we use our concurrent implementation. The same experimental Ajax applications are used, but the number of threads will be increased.

[1] `http://www.google.com/adsense`
[2] `http://www.hostnet.nl`
[3] `http://mijn.hostnet.nl`

We use a workstation equipped with two Intel Xeon E5345 CPU's, which contains 4 cores each resulting in a total of 8 cores that can be used. The workstation contains 16 GB of memory that can be used, and the applications we examined do not require a web server. The only application active will be Crawljax and the browsers used. The same configuration and the default wait-timeout is used as those of chapter 4.

8.1.1 Result

The measurements are split the same way as is done in chapter 4. We analyse the performance of the full-tree and the flat-tree based systems. In this section we focus on the full-tree based systems. In section 4.3 we concluded that those systems were causing the biggest runtime problems because of their back-tracking. When analysing we only focused on the runtime aspect, which is the property we are trying to improve.

Table 8.1 shows the results of using the concurrent Crawljax version on the full-tree based systems. The first column shows the size of the experimental system, while in the following columns the runtime for an increasing number of threads is shown.

Table 8.1: Performance results of concurrently crawling the full-tree based systems.

Size	#1	#2	#4	#8
0 KB	8 min 17 sec	4 min 10 sec	2 min 10 sec	1 min 15 sec
128 KB	11 min 40 sec	6 min 2 sec	2 min 52 sec	1 min 47 sec
256 KB	17 min 19 sec	9 min 5 sec	4 min 27 sec	2 min 37 sec
384 KB	26 min 25 sec	13 min 5 sec	7 min 2 sec	3 min 56 sec
512 KB	39 min 9 sec	19 min 11 sec	9 min 48 sec	5 min 44 sec
768 KB	74 min 8 sec	37 min 11 sec	18 min 54 sec	10 min 41 sec
1 MB	120 min 11 sec	60 min 18 sec	30 min 47 sec	16 min 47 sec

The table shows for all experimental Ajax applications, that the runtime is reduced by nearly a factor of 2 when the number of threads is doubled.

Figure 8.1 shows the size of the DOM-trees on the x-axis in KB, and the runtime on the y-axis in minutes. The different graphs represent the number of threads used.

Figure 8.2 shows the number of threads used on the x-axis and the runtime expressed in minutes on the y-axis. Both plots (8.2a and 8.2b) show the theoretical improvement that can be obtained plotted as a second line. As the workstation used holds 8 cores the theoretical improvement function follows the measured data until 8 threads where it starts to deviate. The theoretical improvement continues, unaware of the limited number of cores present in the machine, while the measured data stays constant (8.2a) or gets even a little worse (8.2b).

8.1.2 Conclusion

Our measurements show an almost linear increase in speed when doubling the number of threads until the number of cores available is reached. The measured optimisation follows the theoretical optimisation very closely. When the number of cores available is reached (at 8 threads) the improvement increase disappears and becomes zero. When the number of threads is increased beyond the number of cores the runtime increases slightly again, which is due to context-switches and synchronised execution of the Java code.

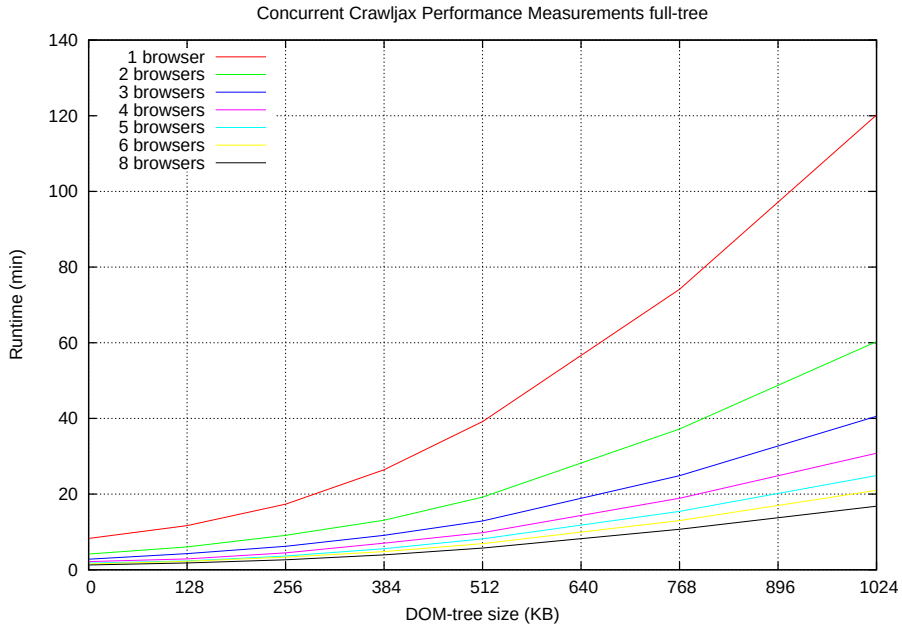
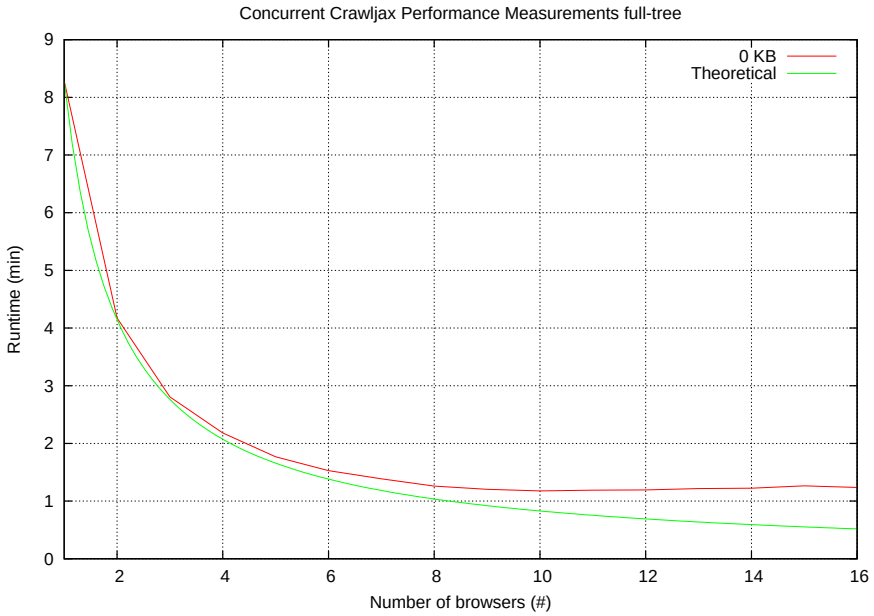


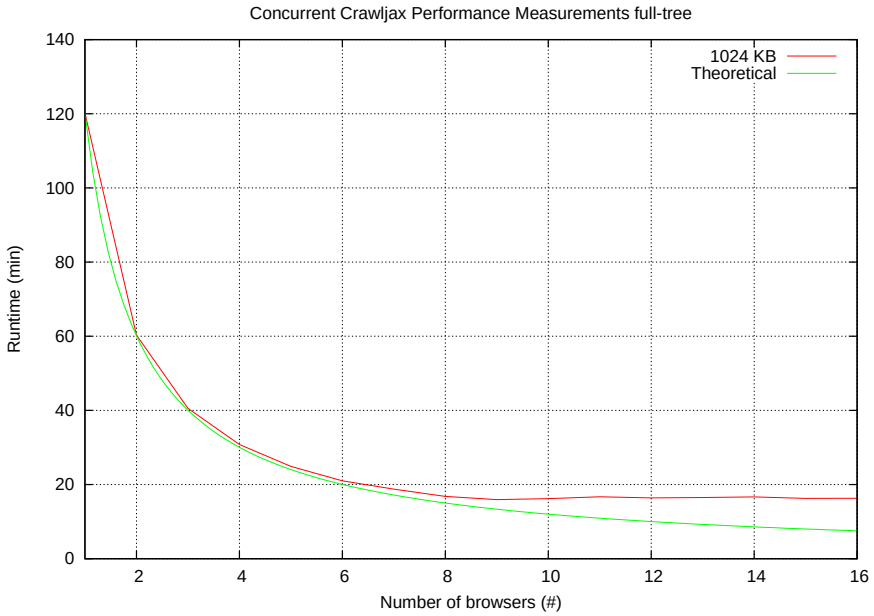
Figure 8.1: Runtime for full-tree based applications with increasing DOM-size and different number of threads

Crawljax has become more scalable till a certain level, in this case until 8 threads are used. Enabling 9 threads or more will not lead to an improvement. We have seen the same performance improvements when the size of DOM-tree increases. Also every run we performed during this evaluation we found the same number of states and edges, so we can conclude that the site coverage when running Crawljax concurrent is the same as running sequentially.

The improvement is nearly as high as theoretical possible when running our experimental Ajax applications. In those applications we increased the size of the DOM-tree with a place-holder text but it is not a *real* Ajax application. To investigate if the concurrent approach really has the same potential for enterprise applications as shown for the experimental Ajax applications, we conducted a number of experiment in industrial settings.



(a) 0-KB



(b) 1024-KB

Figure 8.2: Runtime for full-tree based applications with increasing number of threads

8.2 Google Adsense

Our first business case is *Google Adsense*, a free online Ajax application developed by *Google* that empowers online publishers to earn revenue by displaying relevant ads on their online content. The Adsense interface is purely built using GWT (Google Web Toolkit)[4] components and is written in Java. I performed a 3 months internship at Google, and during this time I had the opportunity to conduct a case-study to investigate the changes applied to Crawljax.

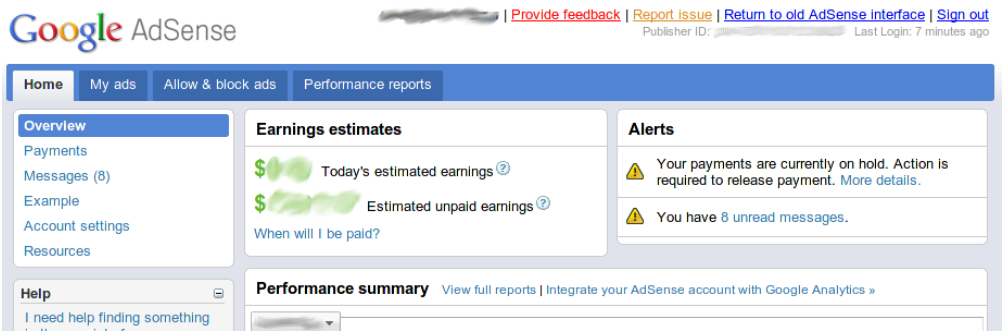


Figure 8.3: Index page of *Adsense*

Figure 8.3 shows the index page of *Adsense*. On the top there are four main tabs, `Home`, `My ads`, `Allow & block ads`, and `Performance reports`. On the top-left side a box is present, the `left-menu`, that holds the anchors for the current selected tab; in figure 8.3 the `Home`-tab is selected. Underneath the `left-menu` box there is a box holding links to help related pages. To the right side of the `left-menu` and the `help-menu` box the main contents of this page can be found. The contents of a page is loaded by an Ajax call into a `div`, and this `div` has the `contents`-id.

To derive our experimental data we used the *Google* infrastructure. This infrastructure offers the possibility to run our experiments on a local workstation or on a distributed cluster.

8.2.1 Infrastructure

The local workstation used for running our experiments is equipped with an Intel Core 2 Quad Q6600 processor and holds 4 GB of memory. The distributed cluster did not have a known specification, the number of cores differs between different clusters. Newer clusters were equipped with 6 or 8 core CPU's, while older clusters had 2 or 4 cores. The amount of memory was also not specified, but the job-distributor ensured a minimum of 2 GB memory per job as a minimum. The distributor also ensured a minimum of one core, but allowed utilising more cores when these are not busy. The cluster is shared between all development teams, so our experiment data was gathered while other teams were also utilising the cluster. To prevent starvation on the distributed cluster a maximum runtime was specified of one hour, and if a test ran for a larger amount of time the test was killed.

[4] `http://code.google.com/webtoolkit/`

To obtain a repeatable experiment every experiment must be self-contained, and therefore we started a new *Adsense* frontend and database server for the start of every experiment. So every experiment contains a fresh and clean database environment. The test-data is loaded into the database during the initialisation of the *Adsense* frontend, and a different set of data is loaded when the test is executed in the local environment than when the test is executed on the distributed cluster.

In the next section we introduce the different configurations we used to execute our evaluation experiments and what results have been measured.

8.2.2 Experimental Setup

We started our evaluation study with four different Crawljax configurations. We started with the first one and extended the coverage while investigating the web application. Every configuration is an extension of the previous configuration.

MainTabs This configuration is the most simple configuration, by using this configuration we were able to access all the content behind the four main tabs. The configuration was built using the `crawler.click(''a'').withAttribute` option. The attribute to select the clickable `divs` was the `class`-attribute of the main tab-bar buttons, and this configuration resulted in 4 states, 4 edges and 16 examined elements.

Left menu This is an extension of the **MainTabs** by also including all links in the `left-menu` box. Crawljax was instrumented via its `crawler.click(''a'').underXPath` option to select the `left-menu` using its xpath. This configuration resulted in 39 states, 40 edges and 551 examined elements when running local. When running on the distributed cluster it resulted in 20 states, 23 edges and 256 examined elements.

All anchors This is an extension of the **Left menu** by also including all anchor tags which are inside the `contents-div`. This is achieved by using the `crawler.click(''a'').underXPath` configuration option. This resulted in 56 states, 66 edges and 1639 examined elements when running local, and on the distributed cluster it resulted in 48 states, 67 edges and 1016 examined elements.

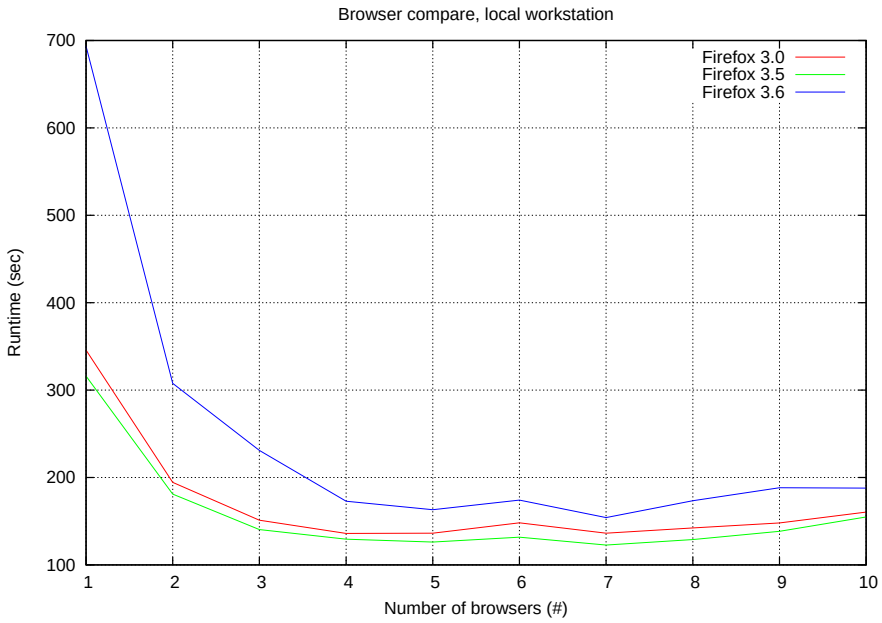
Everything This configuration covers the entire application. It tries to click on all anchor tags, all divs, all table cells and fills in all forms. Using this configuration we were able to cover 121 states, 225 edges and 6468 examined elements when running locally. The **Everything** configuration could not be executed on the distributed cluster because it reached the runtime limit. We excluded the filling of forms in a configuration called **Everything no forms**.

The initial configuration used, **MainTabs**, had a very low coverage of the application, resulting in only 4 states and 4 crawl paths. The optimum number of threads to use would be 4, so this configuration made no sense in analysing because it did not produce any practical results.

To inform the user that the *Adsense* interface is currently loading a loading icon is displayed to the user. This loading icon is displayed on a loading panel, built from a

`div` with the `class`-attribute `loading-panel`. To determine if the interface is ready loading, a query is sent to the DOM-tree to see if the `contents-div` contains text and if there are no `loading`-panels visible to the user. We used this technique instead of using a static timeout to obtain a greater throughput, because when the browser is finished loading it is directly set to work on the next state instead of waiting for the timeout duration.

After defining those four configurations, we conducted a small experiment to determine the browser version to be used in our experiment. Due to the environment used we were restricted to use Linux as our operating system, leaving only Firefox as a candidate to be used as subject browser. The supported versions of Firefox include 3.0, 3.5 and 3.6. Figure 8.4 shows the comparison of the different versions running locally (8.4a) and on the distributed cluster (8.4b).

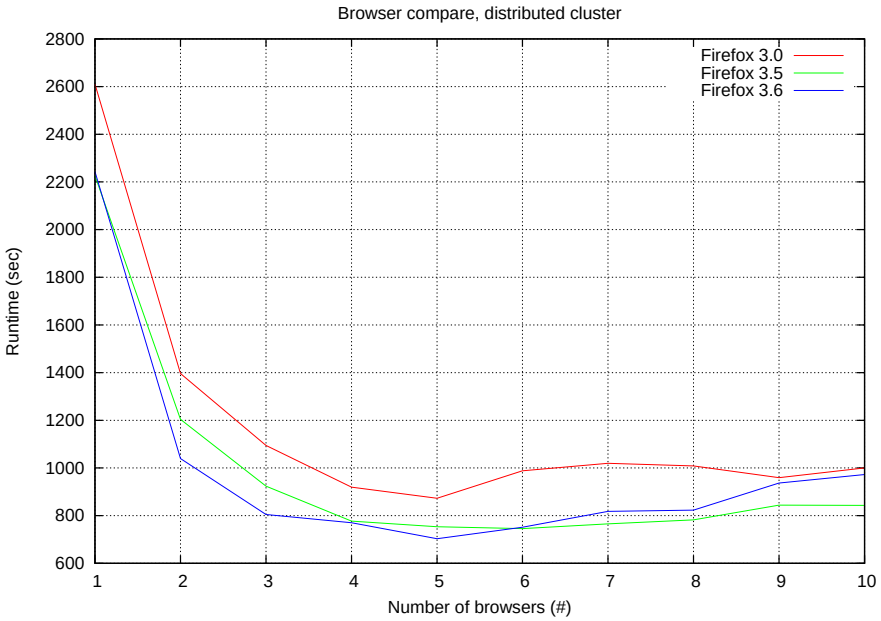


(a) Local

Figure 8.4: Determining the browser version

As figure 8.4 shows the 3.6 version is the slowest browser when running locally, while it can be seen that it is the fastest browser when running on the distributed cluster. Because of this "inconsistent" behaviour of the 3.6 version we chose the 3.5 version of Firefox, that runs the fastest when running locally, and produces more stable measurement data when running on the distributed cluster.

To conduct our experiments we measured the runtime as the time from the start of the crawling of Crawljax until the last browser closes. We also measured the number of states, number of edges and the number of examined elements. These values are retrieved after every successful execution of Crawljax using one of the four configura-



(b) Distributed Cluster

Figure 8.4: Determining the browser version

tions described above. The only parameter changed in the configuration is the number of threads, which is our *independent variable*. The *dependent variable* is the runtime.

The runtime of our experiment is influenced by the other users of the system. To cope with this influence we ran every experiment multiple times and the average of the runtime has been taken. During the local execution of our experiments we turned off all other processes to only have the bare-minimum running, and we repeated our experiment 10 times. On the distributed cluster we were unable to turn off other processes, the resource is shared and to get a reliable average we executed every experiment 300 times.

8.2.3 Results

To examine the data we gathered[5] we used SPSS[6] to perform a statistical analysis and we used R[7] for plotting statistical graphs.

Table 8.2: Descriptive statistics over the dependent variable (runtime) for the independent variable (number of threads used)

minutes

	N	Mean	Std. Deviation	Std. Error	95% Confidence Interval for Mean		Minimum	Maximum
					Lower Bound	Upper Bound		
1	**299**	**17,0613**	**2,31540**	**,13390**	**16,7978**	**17,3249**	**10,17**	**22,15**
2	**299**	**9,9243**	**1,28598**	**,07437**	**9,7779**	**10,0707**	**7,05**	**13,37**
3	**292**	**7,4651**	**1,04823**	**,06134**	**7,3444**	**7,5858**	**5,35**	**10,56**
4	**297**	**6,2338**	**1,03936**	**,06031**	**6,1151**	**6,3525**	**4,74**	**10,32**
5	**294**	**5,9806**	**1,05696**	**,06164**	**5,8593**	**6,1020**	**4,13**	**10,72**
6	**290**	**5,6920**	**,99718**	**,05856**	**5,5767**	**5,8072**	**4,20**	**11,01**
7	**291**	**5,6744**	**1,15004**	**,06742**	**5,5417**	**5,8070**	**3,69**	**10,75**
8	**297**	**5,6404**	**1,11101**	**,06447**	**5,5135**	**5,7673**	**3,78**	**11,12**
9	**295**	**5,5521**	**1,26716**	**,07378**	**5,4069**	**5,6973**	**3,38**	**11,83**
10	**299**	**5,4721**	**1,14160**	**,06602**	**5,3422**	**5,6020**	**2,33**	**10,55**
Total	**2953**	**7,4871**	**3,70372**	**,06816**	**7,3535**	**7,6207**	**2,33**	**22,15**

Table 8.2 show some descriptive statistical numbers from our statistical analysis, it includes the mean, standard deviation, standard error, lower and upper bound, and the maximum and minimum values measured for a given number of threads. These numbers are gathered while running the **Everything no forms** configuration on the distributed cluster. We chose this configuration because it covered the most states and has the highest runtime and investigates the most candidate elements. As can be seen from column two there is no sample which reaches 300 measurements, this is due to problems *Google* faces with their distributed test cluster. The executed Unit-Test sometimes deadlocked without an apparent reason.

To analyse if our measurements are statistical significant we performed an One-way ANOVA test. Before conducting this test we formulated a null hypothesis, added number of threads will not influence the runtime. If the means of all the groups tested by ANOVA are not equal, we can reject the null hypothesis, but we still don't know which of the means differ we solved this problem by performing a post hoc test.

Table 8.3: Test of Homogeneity of Variances

Method	**Statistic**	**df1**	**df2**	**Sig.**
Levene	55,884	9	2943	0.000

The significance value for homogeneity of variances is < 0.05, table 8.3 shows the variances of the groups are significantly different because the significance found is 0.000. Since this is an assumption of ANOVA, we can use the ANOVA method.

[5] http://spci.st.ewi.tudelft.nl/downloads/adsense_concurrent_evaluation.zip
[6] http://www.spss.com/
[7] http://www.r-project.org/

Table 8.4: One-way ANOVA.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	35540.225	9	3948.914	2345.919	0.000
Within Groups	4953.987	2943	1.683		
Total	40494.212	2952			

Table 8.4 shows the the main ANOVA result. The significance value comparing the groups (number of threads) is < 0.05, so we could reject the null hypothesis that added number of threads will not influence the runtime. However, since the variances are significantly different, this might be the wrong answer.

Table 8.5: Robust Tests of Equality of Means.

Method	Statistic	df1	df2	Sig.
Welch	1060,017	9	1198.048	0.000
Brown-Forsythe	2355.528	9	1914.845	0.000

Table 8.5 shows the significance value are both sill < 0.05, so we still reject the null hypothesis. However, these methods does not tell which what number of threads are introduces a significant runtime decrease. To find out we performed a post hoc test, the Games-Howell test.

Table 8.6: Post Hoc Games-Howell multiple comparisons. * indicates the mean difference is significant at the 0.05 level.

	1	2	3	4	5	6	7	8	9	10
1	-	*	*	*	*	*	*	*	*	*
2	*	-	*	*	*	*	*	*	*	*
3	*	*	-	*	*	*	*	*	*	*
4	*	*	*	-		*	*	*	*	*
5	*	*	*		-	*	*	*	*	*
6	*	*	*	*	*	-				
7	*	*	*	*	*		-			
8	*	*	*	*	*			-		
9	*	*	*	*	*				-	
10	*	*	*	*	*					-

Table 8.6 shows the results of the Games-Howell post hoc test, a $*$ means that the difference in runtime is significant. The table shows there is a limit on the number of browsers that can significantly decrease the runtime. This number for our experiments is 5.

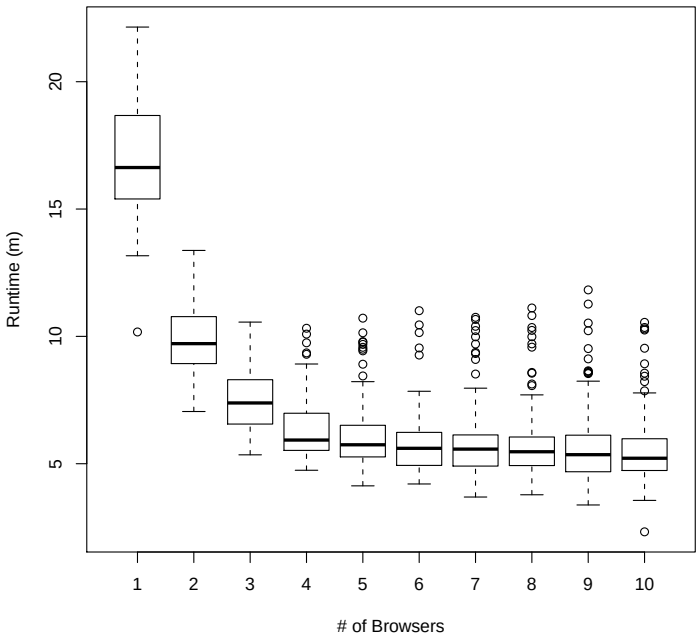


Figure 8.5: Runtime for **Everything no forms** on the distributed cluster

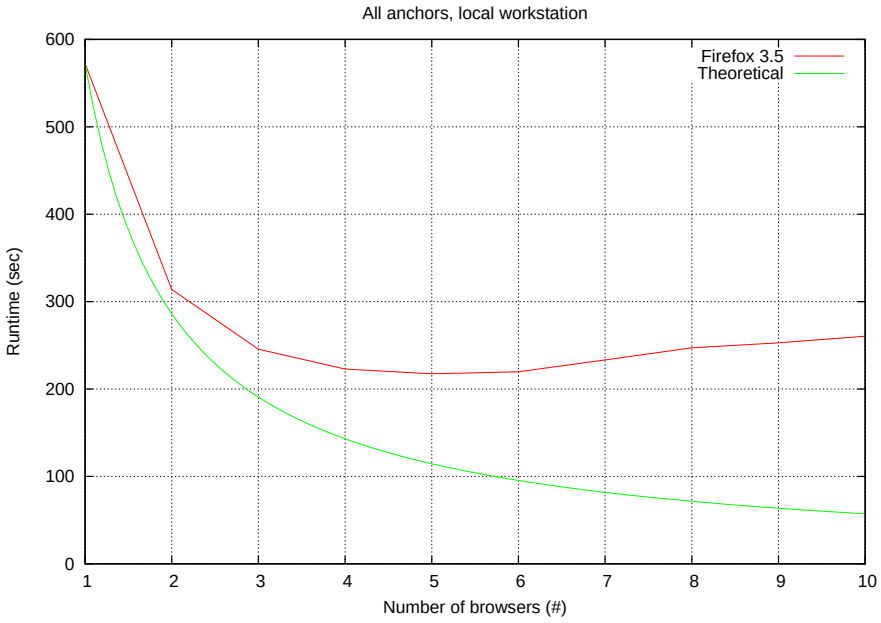
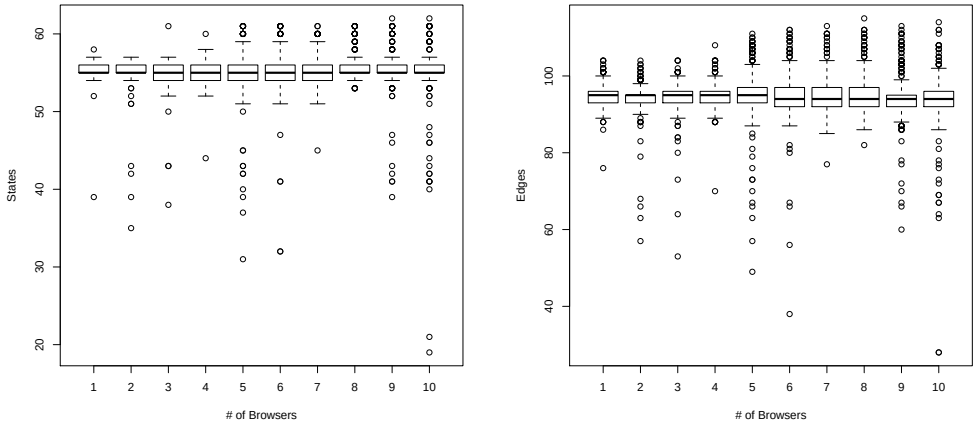


Figure 8.6: Runtime for **All anchors** running locally

Figure 8.5 shows a boxplot of the same data shown in table 8.2. As is also visible from the figure the optimum number of threads is reached when using 5 threads. Adding another thread will not decrease the runtime. Figure 8.6 shows a graph of the runtime when running the **All anchors** on the local workstation. In this graph we also plotted the theoretical improvement; the graph does not follow the theoretical improvement as well as for the experimental Ajax applications. The shortest runtime is found when running with 5 threads. For more than 5 threads the runtime increases because of context-switches and synchronisation inside Crawljax.



(a) Detected states

(b) Detected edges

Figure 8.7: Boxplots of detected states and edges versus the number of threads on the distributed cluster

To analyse the coverage of Crawljax while using the concurrent implementation we measured the number of states and the number of edges detected. Figure 8.7 shows boxplots of the number of detected states (8.7a) and the number of edges (8.7b) versus the number of threads. As can be seen from both figures, the shapes of the boxplots do not change when using multiple threads. The reason for the variation of states and edges is the loading-panel, which in some cases does not show a loading icon but the *Adsense* interface is still loading. Crawljax thinks it is ready for DOM-tree comparison but actually it is not, producing non-detected state changes.

8.2.4 Conclusion

When looking at both figures 8.5 and 8.6 and the data from table 8.2 we can conclude that an improvement in runtime is obtained. Using the ANOVA statistic analysis we showed the data gathered to be statistical significant. We also analysed the optimum number of threads to be 5, as a conclusion from table 8.6. The measured improvement also follows the theoretical optimal improvement, but not as close as we have seen for the experimental Ajax applications.

Table 8.7 shows all the speedups found in *Adsense*, the optimal speedup is printed in bold. The optimal number of threads used while running the experiments on the local workstation is 5 except for the **Left menu** configuration where it is 4 threads.

Table 8.7: Calculated speedups, compared with one browser.

	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
D - All anchors	1.56	1.82	1.97	**2.07**	2.00	1.99	1.97	1.92	1.85
D - Everything no forms	1.72	2.29	2.74	2.85	3.00	3.01	3.02	3.07	**3.12**
D - Left menu	1.66	2.28	2.73	2.90	**3.06**	3.01	2.91	3.01	3.02
L - All anchors	1.82	2.33	2.57	**2.63**	2.60	2.45	2.32	2.26	2.20
L - Everything	1.70	2.17	2.64	**3.24**	2.92	2.95	3.09	2.85	2.42
L - Left menu	1.73	2.15	**2.32**	2.27	2.23	2.13	2.06	1.98	1.88

On the distributed cluster the optimal number of threads is between 5 and 6 with one exception, the **Everything no forms** configuration keeps improving. But as calculated and shown in table 8.6 increasing from 6 to 7 threads did not result in a change which is significant enough. These conclusions are also visible from figures 8.5, 8.6, A.2, A.3, A.4, and A.5. We can observe that it is likely that the optimal number of threads to be used is the number of cores +1. To further investigate this claim and to further investigate our research questions we conducted another case-study.

8.3 Mijn Hostnet

Our second business case is *Mijn Hostnet*, which is an online customer portal used by *Hostnet* to offer customers self-service. Inside the portal customers can see and change their address, their domain-name registrations, their DNS-zones, etc. The portal is written using the PHP programming language and the Symfony[8] framework. The framework offers Ajax integration using the jQuery Ajax library.

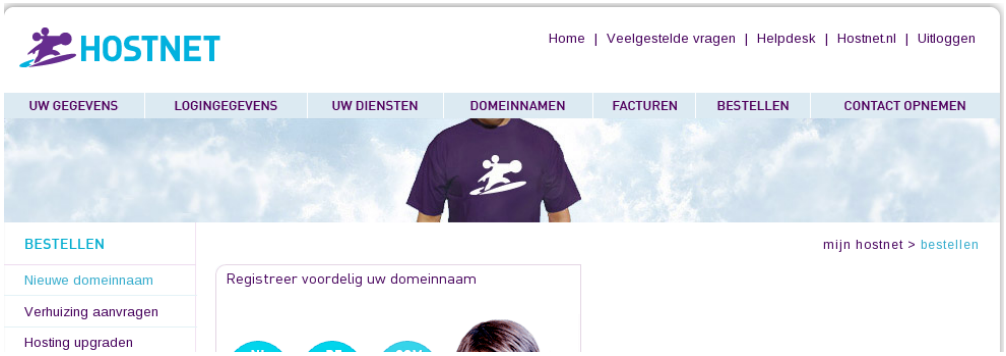


Figure 8.8: Index page of *Mijn Hostnet*

Figure 8.8 shows the index page of *Mijn Hostnet*. It follows the same design as the *Adsense* application with a tab-bar at the top and a menu box on the left side. The contents of a page is loaded by requesting a URL, but sometimes this also happens by the use of an Ajax call into a `div`, and this `div` has the `content`-id.

8.3.1 Infrastructure

The workstation we used for our evaluation experiments was equipped with an Intel core 2 duo E6600 CPU and 2 GB of memory. On this workstation we installed a web server to host the *Mijn Hostnet* application. The application also uses a database server and for this server we chose to use the staging server already setup by *Hostnet*. On this workstation we had control over the number of other processes active while running our experiments. The only external influence we could not control was the use of the staging database server.

8.3.2 Experiment Setup

We made four different Crawljax configurations to conduct our experiments, and we tried to use the same logic as with the *Adsense* case-study. The configurations below have an increasing number of states examined.

MainTabs This configuration is the base configuration for all other configurations, and it accesses all the `anchor`-tags of the main tabs. The configuration was created using the `crawler.click(''a'').underXPath()` call. The `div` which holds the main tab `anchor`-tags has the id `menu`. The configuration results in 8 states, 7 edges and 56 examined elements.

[8] `http://www.symfony-project.org/`

LeftMenu This configuration is an extension of the **MainTabs** configuration by including also all `anchor`-tags inside the `left`-menu. This configuration is extended by adding the `crawler.click(''a'').underXPath()` statement. The `left`-menu is identified by the `links`-id. Using this configuration will result in 19 states, 25 edges and 200 examined elements.

Tabbies On some pages *Mijn Hostnet* features an embedded `tab`-panel called `tabbies`. This configuration digs into those embedded `tab`-panels and also covers those tabs. The configuration is extended by adding a `crawler.click(''a'').underXPath()` call. This enabled examining all the `anchor`-tag inside the `div` with identifier `tabbies`. This configuration results in 26 states, 37 edges and 299 examined elements.

All anchors This configuration conducts all links found on the page, except for the `anchor` tags in the `div` with identifier `headrightbottom`. This configuration will result in 47 states, 98 edges and 1381 examined elements.

We approached the configuration setup the same as for the *Adsense* case, but we also discared the **LeftMenu** configuration because the number of states covered was to low for measuring reliable data.

In all our configurations we used a static wait-time after every event to detect DOM-changes. The wait-timeout was not changed from the default value, which is a 500 milliseconds wait-timeout. Due to environment restrictions we were only allowed to use the Linux operating system, leaving only the Firefox web browser as a choice. We chose the same version of the browser as we used in our *Adsense* experiments. For all experiments we increased the number of threads used (*independent variable*), and we measured the runtime of executing a single experiment (*dependent variable*). Every experiment we repeated 20 times and afterwards we took the average to store as runtime.

8.3.3 Result

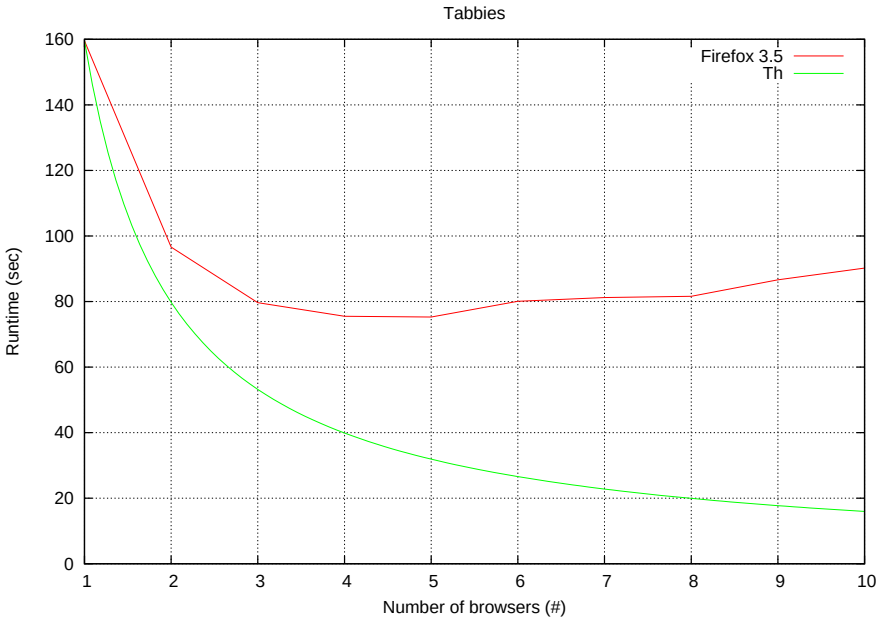
We performed experiments on all the configurations we defined. The two configurations with the least amount of states, **mainTabs** and **LeftMenu**, were dropped from our experiments because of their small runtime. Both resulted in inaccurate measurements because network delays in reaching the database cluster have too much influence on the runtime.

Table 8.8: Runtime results of concurrently crawling the *Mijn Hostnet* application

configuration	#1	#2	#3	#4	#6
Tabbies	2 min 39 sec	1 min 36 sec	1 min 19 sec	1 min 15 sec	1 min 20 sec
All anchors	6 min 44 sec	3 min 56 sec	3 min 5 sec	2 min 51 sec	2 min 56 sec

Table 8.8 shows the measured values for the runtime when executing the given configurations. The runtime decreases when adding an extra thread, the optimal number of threads in both configurations is 4, although the speed increase gained using 4 threads instead of 3 is giving a non significant improvement. Using 3 instead of 2

threads decreases the runtime between 17% to 21%, while using 4 instead of 3 threads only decreases the runtime between 5% to 7%.



(a) **Tabbies** configuration

Figure 8.9: Performance of Crawljax for *Mijn Hostnet* case-study, number of threads versus runtime

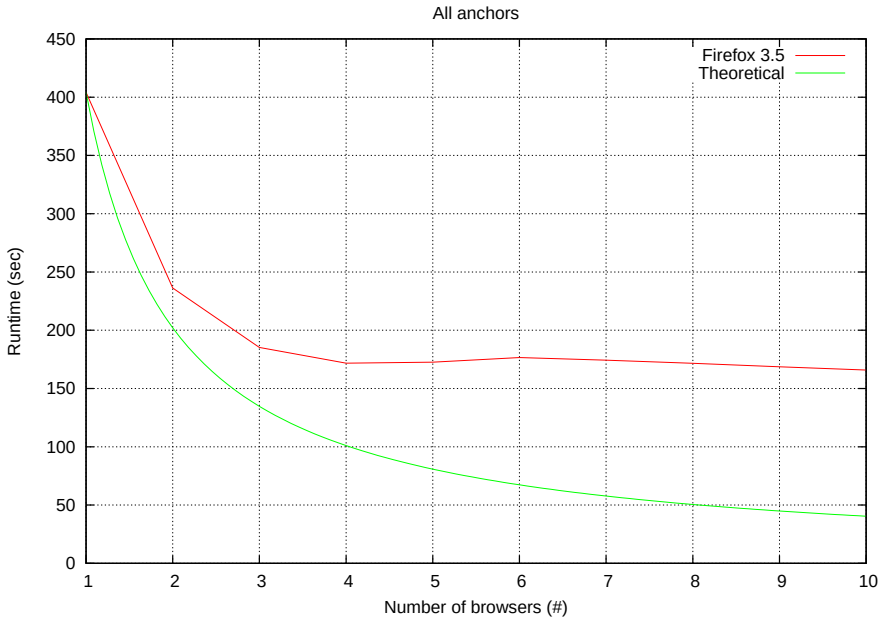
Figure 8.9a shows the runtime measured when increasing the number of threads while executing the **Tabbies** configuration. This figure also shows the theoretical improvement. Figure 8.9b shows the same measurements performed while running the **All anchors** configuration. The theoretical improvement is not followed that closely as in section 8.1. The improvement in runtime follows the same slope as our theoretical improvement.

8.3.4 Conclusion

Table 8.9: Calculated speedups, compared with one browser.

	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
All anchors	1.71	2.18	**2.35**	2.34	2.29	2.32	2.34	2.31	2.34
Tabbies	1.65	2.00	**2.11**	2.11	1.99	1.87	1.96	1.54	1.77

Looking at both figures 8.9a and 8.9b and the data from table 8.9 we can conclude that the runtime can be reduced. It is difficult to make a statement about the influence of the number of cores in the workstation because the workstation used only contains 2



(b) **All anchors** configuration

Figure 8.9: Performance of Crawljax for *Mijn Hostnet* case-study, number of threads versus runtime

cores. The data shows an optimal number of threads of 4, although 3 threads is already very near the optimal number of threads.

In this chapter we analysed the performance of our concurrent approach by performing an experimental case and two case-studies. In the next chapter we discuss the strengths and limitations of our concurrent approach.

Chapter 9

Discussion

In this chapter we discuss the results of our evaluation case-studies and our implementation. In the first section we revisit our research questions. Section 9.2 discusses the strengths of our approach, section 9.3 discusses the limitations of our approach. In section 9.4 we discuss the possibility to use a distributed approach for crawling Ajax applications.

9.1 Research Questions Revisited

In this section we revisit on our research questions proposed in section 1.5:

RQ1 What is the current performance of Crawljax?

RQ2 How can we improve the performance?

RQ3 How effective are the proposed improvements?

In chapter 4 we conducted some experiments to measure the performance of Crawljax. During this research we empirically determined that the total memory usage on average is 4 times the size of a single DOM-tree multiplied by the number of states. We concluded that enterprise applications most likely will not have such an amount of states that memory usage will be a limiting factor. We therefore focused on solutions to optimise the runtime. The first real limitation on an enterprise application was found during our case-study on the *Adsense* software. The **Everything** configuration had a runtime which was too large to execute on the distributed testing platform because its runtime was longer than the time limit of one hour. The **Everything** configuration, when executed locally, had around 120 states. If the states had a DOM-tree size of 1 MB, this results in 4 MB per state. The total memory consumption will therefore be $4MB \times 120 = 480MB$. These results indicate that we have come to the right conclusion in chapter 4 to optimise the runtime.

The first research question, **RQ1**, can now be answered. In terms of maximum number of states, Crawljax is limited by the number of states that fit inside the memory of a workstation. On average every state uses 4 times the amount of memory the DOM-tree uses, so for a given DOM-tree size the maximum number of states possible can be calculated. The runtime depends on the number of states, layout of the state-flow

graph and the wait-timeout used. In chapter 4 we concluded that the runtime increases slightly exponential, when the DOM-tree size increases linear.

To answer the second research question, **RQ2**, we proposed and implemented two possible solutions. In chapter 5 we discussed the back-tracking optimisation. This was our initial optimisation, but as concluded in section 5.4, the performance gain is dependent on the structure of an Ajax application. When the state-flow graph of an application results in a flat-tree, the performance gain will be low. A typical Ajax application most likely has a non-flat state model and therefore we believe that the back-tracking optimisation will provide a certain degree of performance improvement. The second optimisation we proposed is a concurrent solution, where by dividing the work over multiple threads all cores inside a workstation can be used, which leads to shorter runtimes.

In the next section we discuss the last research question, **RQ3**, and we focus on the strengths of our solution. In section 9.3 we focus on the limitations of the concurrent solution.

9.2 Strengths

For all case-studies performed we were able to gain a performance increase when using multiple threads. The theoretical maximum improvement, as introduced in section 8.1, is defined by the formula $T_{min} = \frac{1}{n} \times T_1$, with T_{min} giving the minimum runtime, n the number of threads used and T_1 the runtime of single threaded execution. The experimental Ajax applications we used for our measurements from chapter 4, approached the theoretical maximum improvement, using 8 threads we were able to achieve a speedup of 7.16. The *Adsense* and *Mijn Hostnet* case-studies did not approach the theoretical improvement but clearly showed the tendency towards the theoretical improvement.

During the evaluation of our implementation we did not find any case where the runtime of the concurrent version was larger than the runtime when using only one thread. There is always a reduction in runtime when crawling with multiple threads instead of one. The lowest speedup we found was 1.56 when using two threads for the **All anchors** configuration of the *Adsense* case-study, running on the distributed testing platform.

Using our concurrent approach we were able to perform the **Everything** configuration on the *Adsense* case-study, which was not possible to execute because of the time constraint when crawling with only one thread. The concurrent approach enabled a wider spectrum of possibilities, e.g., we were able to execute both the **Main tabs and left menu** and **All anchors** configurations after every code commit at *Google* instead of only being able to execute the **Just the main tabs** configuration.

Table 9.1 shows the speedups reported by Holzmann et al. [30], Jabbar2006 et al. [35] and Dig2009 et al. [14]. They conducted experiments during their research, on 2, 3 and 4 core CPU's. When we compare their found speedups with the speedups we found, shown in table 9.2, we can conclude our concurrent implementation shows similar results.

When using 2 thread we found lowest speedup of 1.65, only in [14] there are some speedups exceeding our performance. Our speedups when running with 3 threads

Table 9.1: Speedup results found in [14, 35, 30]

Model	2 core
DEOS	1.34
Gardag	1.31
NVDS	1.51
EO1	1.50
CP	0.74

(a) Liveness [30]

Model	2 core	3 core
GIOP 3-2	1.45	1.64
GIOP 4-1	1.64	2.05
GIOP 4-2	1.64	2.00

(b) External A* in GIOP [35]

Program	2 core	4 core
mergeSort	1.18	1.60
fibonacci	1.94	3.82
maxSumConsecutive	1.78	3.16
matrixMultiply	1.95	3.77
quickSort	1.84	3.12
maxTreeDepth	1.55	2.38

(c) devide-and-conquer [14]

exceeds the results from [35] except for the MH - Tabbies case. The speedups found when running with 4 threads are nearly all lower than the speedups found by Dig2009 et al. in [14]. We can conclude that the results we found during our evaluation are similar as the results others have found. In some cases our concurrent implementation is slower compared to their speedups, but when compared with the results from the model checking domain ([30, 35]) we superseded their speedups.

Table 9.2: Speedup results found for the *Mijn Hostnet* (MH) and *Adsense* (A) case-studies running locally

Configuration	2 threads	3 threads	4 threads
MH - Tabbies	1.65	2.00	2.11
MH - All anchors	1.71	2.18	2.35
A - Left menu	1.73	2.15	2.32
A - All anchors	1.82	2.33	2.57
A - Full	1.70	2.17	2.64

In the next section we discuss the limitations of our approach.

9.3 Limitations

Our back-tracking optimisation has a slight limitation. When it is used it will be beneficial, even if there is absolutely no back-tracking (no optimisation) the execution will not be delayed. When using a plugin, which is executed during back-tracking, the plugin might not be executed that often. We expect back-tracking to be executed to further examine un-explored candidate elements within states, so plugins will not be skipped, but there is a theoretical possibility that a plugin might be skipped.

The major limitation of the concurrent optimisation is the number of threads used while crawling. At a certain number of threads the runtime can not be reduced any further. This number of threads is not a fixed number, we believe its dependent on the number of CPU cores available in the workstation used. If the crawler is performing its calculations, the browser used is idle and waiting for its next instruction. If the crawler sends an instruction to the browser the crawler becomes idle and waits for the browser to be finished. This correlation makes sure the thread is always fully loaded. As we noted in chapter 8 during our evaluation of the research on the optimal number of threads to use, it was found that the optimal number of threads roughly equals the number of CPU cores +1. We found this to be true while using a dual core workstation

(*Mijn Hostnet*), a quad core workstation (*Adsense*) and two quad core workstation (*Experimental Ajax Applications*). We did not investigate this relation ship thoroughly, if this relation ship turns out to be true it limits the concurrent optimisation to the number of cores available inside a workstation. If the runtime limit is reached again, we can not use the concurrent optimisation to decrease the runtime any further.

In our research we did not optimise the memory usage, but there are techniques possible to optimise the memory usage. Such as state compression [31], using the gzip or vdelta compression techniques[44]. An other possible memory reduction technique is the use of a diff-tool to store only the differences compared with the previous state[33, 57]. The memory usage is not increased by using our concurrent optimisation, because there is still a single state-flow graph that holds the same amount of states. The runtime is reduced leaving the memory consumption as the next target for optimisation.

9.4 Distributed crawling

When the number of CPU cores is not scalable enough, and the number of threads has to be increased beyond the number of CPU cores available, a distributed approach can be taken. During the implementation of the concurrent optimisation we identified, isolated and guarded the shared memory within Crawljax. This information can be used as a starting point for a distributed approach. During our research we performed some experiments to investigate the feasibility of running distributed crawling.

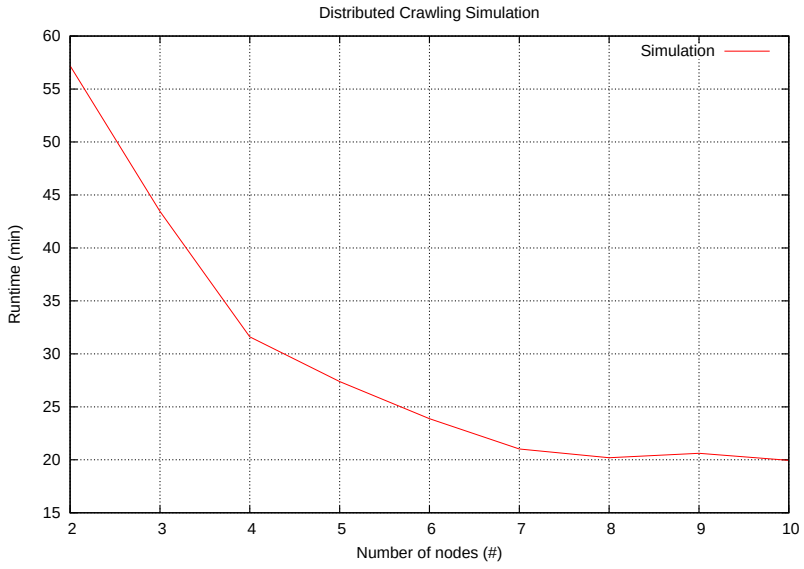


Figure 9.1: Distributed Crawling Simulation

Figure 9.1 shows the runtime for a given number of nodes participating. We per-

formed this experiment on Crawljax by inserting network communication delays we expect to occur when running in a real distributed environment. For every communication with the state-flow graph we added a sleep timeout of 1 second. The figure shows a decrease in runtime when the number of nodes is increased. After about 8 nodes the runtime does not decrease any more, and becomes stable. This is due to the fact that there is not enough work inside the work queue.

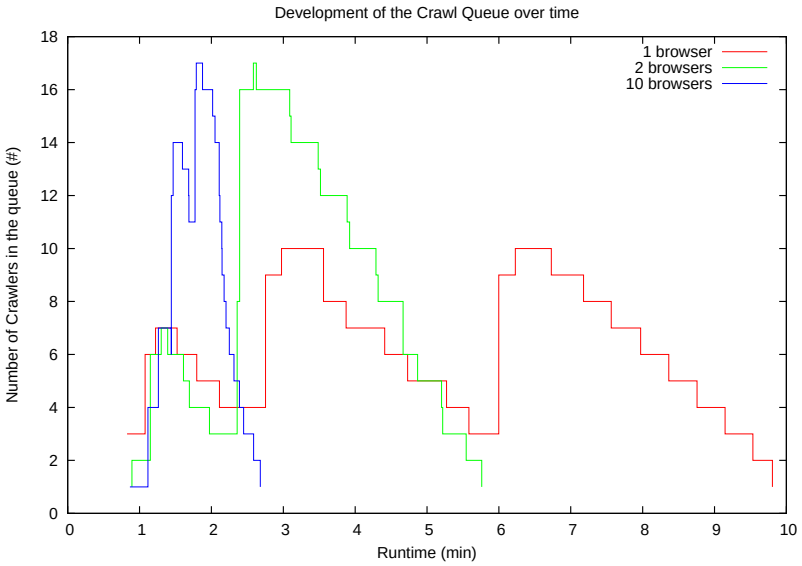


Figure 9.2: Crawl-queue development over time

In figure 9.2 the development of the queue-size is shown. We recorded the size of the work-queue every time a crawler was added or subtracted from the queue. On the x-axis the runtime is shown in minutes, while the y-axis shows the size of the queue. As can be seen from figure 9.2 the maximum size of the queue depends on the number of nodes used. If more nodes are participating then there is work available, the nodes are not participating in the reduction of the runtime any more; for this experiment this happened beyond 8 nodes. So the limiting factor of this solution will be the amount of work available in the work-queue. If the full state-flow graph is known in advance the distribution the number optimal maximum number of nodes participating can be calculated. A normal `exploring` crawl, where the crawler analyses the candidate elements and finds its way through the state space the optimal number of participating nodes can not be calculated, and the runtime depends on the development of the work-queue.

Chapter 10

Conclusions

This chapter concludes this thesis. In the first section we discuss all the contributions that were made. In section 10.2 we conclude with some pointers for directions in which future research can be performed.

10.1 Contributions

We started our research by measuring the performance of Crawljax to find its limitations. One of the results of this measurement on the performance is the benchmark plugin for Crawljax. After we detected the limitations we first optimised the back-tracking performance, and continued to develop our concurrent approach.

Our concurrent approach to crawl Ajax applications, implemented in Crawljax 2.0, has proven to be very successful. In our experimental Ajax applications we found that our optimisation reached the theoretical speed improvement possible, i.e., every time the number of threads are doubled the runtime is reduced by a factor of 2 until the number of CPU cores is reached. The evaluation case-studies showed a significant speedup of 1.56 at minimum, especially when the performance on two threads is compared with the performance on one thread.

The best practical results of our concurrent optimisation was achieved during our *Adsense* case-study. Using our concurrent approach we were able to execute the **All anchors** configuration after every code change. The major challenge was the tight runtime requirement of 15 minutes. At first it was only possible to execute the **Just the main tabs** configuration. The change from the **Just the main tabs** configuration to the **All anchors** configuration largely increased the testing coverage of the application.

The main contributions of this thesis include:

- Optimised back-tracking for crawling Ajax applications
- A concurrent approach for crawling Ajax applications
- Implementation of the concurrent approach for crawling Ajax applications
- Evaluation of the concurrent approach by performing two case-studies.

10.2 Future work

There are several possible directions for future research. In our research we focused on the performance of Ajax crawlers and specifically on the performance of Crawljax. As discussed in section 9.4 we performed a small simulation to research the feasibility of using a distributed approach. This simulation was performed during the internship at Google. Google also wanted to operate Crawljax in a distributed fashion to make it scalable to their needs. A direction for further research can be to research this possibility and finding the communication or other limitations of a distributed approach. An other direction is to examine the possibility to adapt the MapReduce programming paradigm to enable feeding work while executing. If that is possible our concurrent approach can be used on a MapReduce cluster.

The runtime of Crawljax is largely influenced by the wait-time detection. When a fixed wait-time is not sufficient, e.g., crawling an enterprise Ajax application, a DOM-tree element is queried to determine if the loading of a the state is finished. Every query send causes network traffic, calculations on the browser, and when not yet finished a small wait-time. This system is a poll-based querying system executed from within Crawljax. The most efficient system would be a push-based system where the Ajax application notifies Crawljax that it is finished loading or a blocking-operation, which waits until the DOM-tree is fully loaded as proposed by Montoto et al. in [45].

An other optimisation we did not investigate is the crawl-order. When crawling with one thread a depth-first approach is taken, but when using multiple threads the order is undefined as described in section 7.6. It might be beneficial in reducing runtime to change or guide the ordering in which candidate elements are explored. In an ideal world the work over the threads is equally balanced, all threads finish at the same time and are always busy. A different crawl-order might help to achieve this.

During our evaluation we encountered a concurrent bug in a web application, which might be a new type of bugs to investigate. Using Crawljax as a concurrent bug detection system.

During our research we noted that the optimal number of threads to be used was the number of CPU cores +1. We did not investigate this relationship thoroughly. Future research could try to find a relation between the number of cores and the optimum number of threads used.

Appendix A

Measurement Plots

This chapter contains all the figures not displayed in the evaluation chapter, but for which we measured performance data.

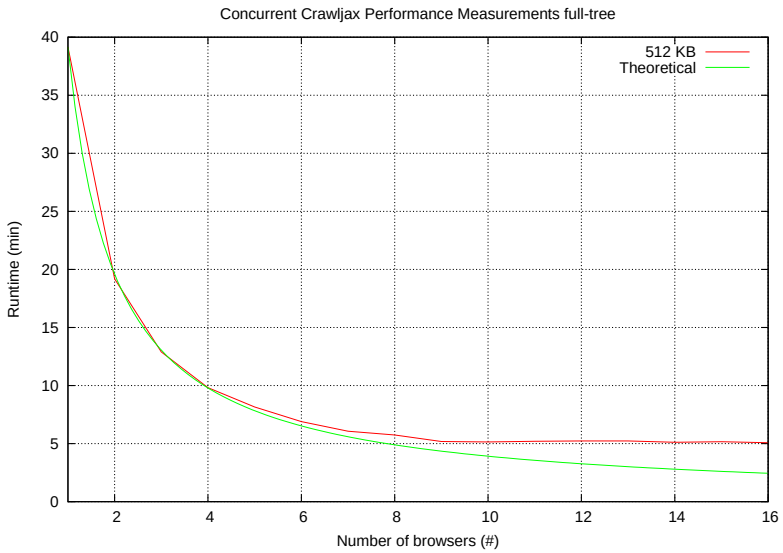


Figure A.1: Full-tree based Experimental Ajax applications, DOM-size 512 KB. Runtime for a given number of threads

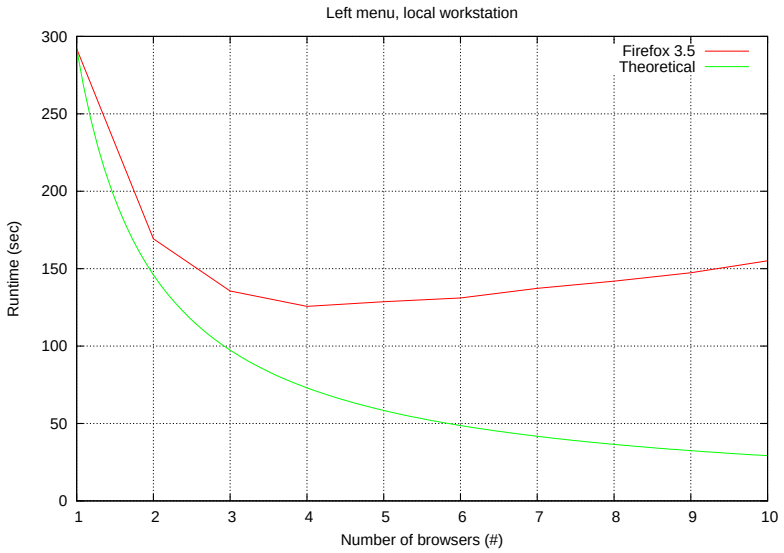


Figure A.2: Adsense, crawling **Left menu** on a local workstation. Runtime for a given number of threads

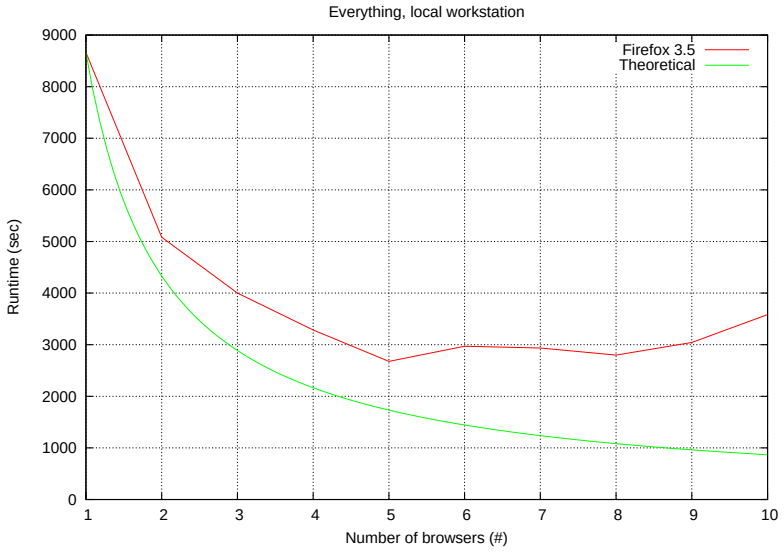


Figure A.3: Adsense, **Everything** on a local workstation. Runtime for a given number of threads

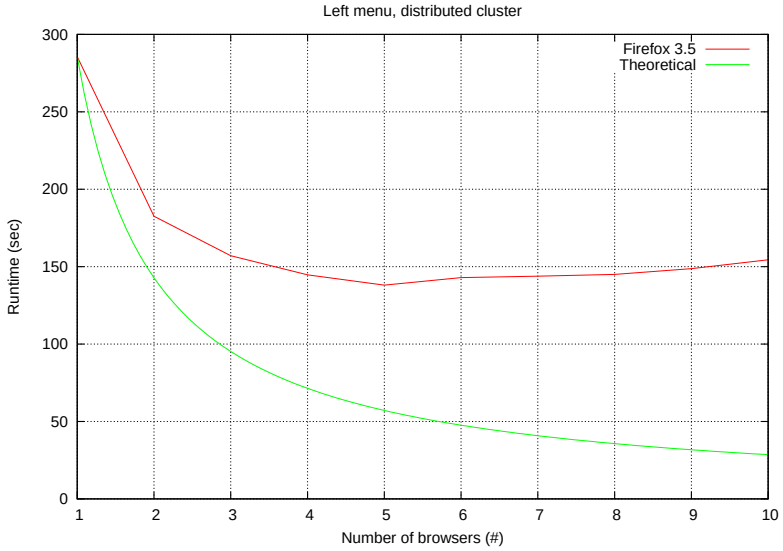


Figure A.4: Adsense, crawling **Left menu** on the distributed cluster. Runtime for a given number of threads

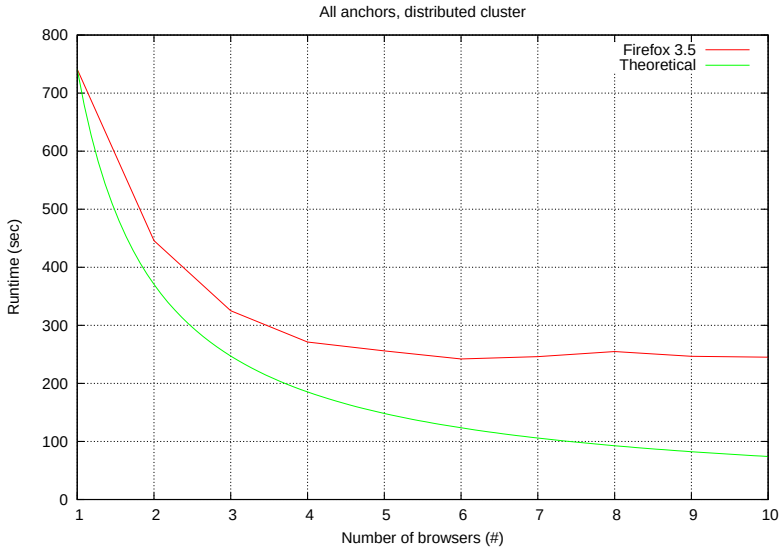


Figure A.5: Adsense, crawling **All anchor** on the distributed cluster. Runtime for a given number of threads

Bibliography

[1] M. Allen. *Palm webOS: Developing Applications in JavaScript using the Palm Mojo Framework*. O'Reilly, August 2009.

[2] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *Procedings of the 23rd International Conference on Data Engineering*, pages 6–20. IEEE Computer Society, 2007.

[3] M. Biberstein, J.Y. Gil, and S. Porat. Sealing, Encapsulation, and Mutability. In *15th European conference on object-oriented programming*, pages 28–52. ECOOP, Springer, 2001.

[4] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[5] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 896–903. IEEE Computer Society, 2004.

[6] L. Chai, Q. Gao, and D.K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 471–478. IEEE Computer Society, 2007.

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

[8] S. Christensen, L.M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. *Lecture Notes in Computer Science*, 2031:450–464, 2001.

[9] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176–194, 2001.

[10] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[11] E.M. Clarke, O. Grumberg, and D.A. Peled. Model checking. *Lecture Notes in Computer Science*, 1346:54–56, 1997.

[12] L. de Alfaro. Model Checking the World Wide Web. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 337–349. Springer Verlag, 2001.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, 2008.

[14] D. Dig, J. Marrero, and M.D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering*, volume 2009, pages 397–407. IEEE Computer Society, 2009.

[15] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 50:269–271, 1959.

[16] C. Eisner and D. Peled. Comparing Symbolic and Explicit Model Checking of a Software System. In *SPIN Workshop on Model Checking of Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 2002.

[17] A.D. Fekete. Teaching students to develop thread-safe java classes. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 119–123. ACM, 2008.

[18] R.T. Fielding. Maintaining distributed hypertext infostructures: Welcome to momspider's web. *Computer Networks and ISDN Systems*, 27(2):193–204, 1994.

[19] G.E. Gallasch, J. Billington, S. Vanit-Anunchai, and L.M. Kristensen. Checking safety properties on-the-fly with the sweep-line method. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3):371–391, 2007.

[20] J. Gao, H.S.J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc., 2003.

[21] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. *Lecture Notes in Computer Science*, 2057:217–234, 2001.

[22] J. Garnet. Ajax: A new approach to web applications. Online, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[23] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[24] A. Groce and W. Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[25] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[26] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[27] G.J. Holzmann. *Design and Validation of Computer Protocols*, chapter 11, pages 214–240. Prentice Hall, 1991.

[28] G.J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997.

[29] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[30] G.J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[31] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In *Proceedings of the International Symposium on Protocol Specification, Testing and Verification XII*, pages 349–363, 1992.

[32] J.E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, Stanford, CA, USA, 1971.

[33] J.J. Hunt, K.P. Vo, and W.F. Tichy. An empirical study of delta algorithms. *Lecture Notes in Computer Science*, 1167:9–30, 1996.

[34] C.N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, department of computer science, 1996.

[35] S. Jabbar and S. Edelkamp. Parallel External Directed Model Checking with Linear I/O. In *Verification, model checking, and abstract interpretation: 7th international conference*, pages 237–251. Springer-Verlag, 2006.

[36] L.M. Kristensen and T. Mailund. A generalised sweep-line method for safety properties. *Lecture Notes in Computer Science*, 2391:549–567, 2002.

[37] R. Kumar, G. Mercer, O. Snell, S. Morse, W. Embley, and G.R. Bryce. Load Balancing Parallel Explicit State Model Checking. *Electronic Notes in Theoretical Computer Science*, 128(3):19–34, 2004.

[38] R. Lämmel. Google's MapReduce programming model–Revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[39] D. Lea. *Concurrent programming in Java: design principles and patterns*. Prentice Hall, 1999.

[40] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. *Lecture Notes in Computer Science*, 1680:22–39, 1999.

[41] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by Inferring User Interface State Changes. In *Proceedings of the 8th International Conference on Web Engineering*, pages 122–134. IEEE Computer Society, July 2008.

[42] A. Mesbah and A. van Deursen. Invariant-Based Automatic Testing of Ajax User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2009.

[43] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[44] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. *SIGCOMM Comput. Commun. Rev.*, 27(4):181–194, 1997.

[45] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. López. Automating Navigation Sequences in AJAX Websites. In *Proceedings Web Engineering: 9th International Conference, ICWE 2009 San Sebastián*, pages 166–180. Springer-Verlag New York Inc, 2009.

[46] T. Oreilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. Online, September 2005. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

[47] R. Pelánek. Fighting State Space Explosion: Review and Evaluation. *Formal Methods for Industrial Critical Systems*, 2008:15, 2008.

[48] C.P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, 100(22):1099–1102, 1973.

[49] S. Pichai and L. Upson. Introducing the Google Chrome OS. Online, July 2009. `http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html`.

[50] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic Detection of Immutable Fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2000.

[51] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. *Lecture Notes in Computer Science*, 2988:497–511, 2004.

[52] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, volume 7, pages 13–24. IEEE Computer Society, 2007.

[53] Danny Roest, Ali Mesbah, and Arie van Deursen. Regression Testing Ajax Applications: Coping with Dynamism. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010.

[54] U. Schonfeld and N. Shivakumar. Sitemaps: above and beyond the crawl of duty. In *Proceedings of the 18th international conference on World wide web*, pages 991–1000. ACM, 2009.

[55] A. Valmari. The state explosion problem. *Lecture Notes In Computer Science, Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, 1491:429–528, 1996.

[56] T. Villa, T. Kam, R.K. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of finite state machines: logic optimization*. Kluwer Academic Publishers Norwell, 1997.

[57] Y. Wang, D.J. DeWitt, and J.Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings. 19th International Conference on Data Engineering*, volume 1063, pages 519–530, 2003.

[58] D.F. Zucker. What does AJAX mean for you? *Interactions, ACM*, 14(5):10–12, September & October 2007.