OPTIMIZED CSS ENGINE

Alessandro Cogliati and Petri Vuorimaa
Telecommunications Software and Multimedia Laboratory , Helsinki University of Technology,
P.O. Box 5400, FI-02015 HUT, Finland

Keywords: CSS, XML, optimization, browser.

Abstract: Future Web applications will be based on XML platform. CSS is a tool used to create different XML presentations' layouts in the heterogeneous set of client devices, which often have limited resources. In this paper, design and implementation of an optimized CSS engine are described. At first, the optimization algorithm is explained, and then the implementation of the CSS engine and its integration within an XML browser are described. Measurements taken with real Web XML documents styled with CSS style sheets show performance improvements of the optimization.

1 INTRODUCTION

Many open standards and the focus on communication among people and applications have created an environment where Web services are becoming the platform for building interoperable distributed applications. Extensible Markup Language (XML) (Yergeau, F. et al., 2004) is the basic format for representing data on the Web services platform. As consequence, application development is moving towards browser-based clients, eliminating the high costs of deploying applications to different environments. (Wolter , R., 2001)

In order to suit the needs of diverse applications, World Wide Web Consortium (W3C) (Jacobs, I., 2005) has been working on standard XML vocabularies to define individual XML languages. Some of these languages are meant only for structuring data, conveying the semantic meaning of documents; others are designed also to provide presentations, describing visual content that browsers can display. In addiction, W3C has been developing Cascading Style Sheets (CSS) (Bos, B. et al., 1998): a mechanism that allows authors to modify the default visual properties of those target XML languages, and where the target language is pure XML, CSS allows to define these visual characteristics from the ground up.(Didier,M., 2000)

CSS is used to add value to rendering layouts, offering many advantages to Web application developers. It reduces complexity of XML documents, moving style attributes to CSS documents, which are easy to utilize because of the simple grammar syntax and semantic. Also, it gives consistency to XML documents of same application, since a unique CSS style sheet can be shared. But, the most important features that make CSS one of the most used Web design tool are mainly two:

I. It permits to create different layouts for different devices such as PDAs, cell phones, and digital television set-top boxes. And, it makes this procedure robust, since style adaptation on devices takes place on client side after all the content is delivered, without needing extra protocols.
II. Interoperability with every kind of XML languages such as XHTML (Pemberton, S. et al., 2002), SVG (Ferraiolo, J., 2001), XForms (Dubinko, M. at al., 2003), SMIL (Ayars, J. et al., 2001), etc.

Programs supporting CSS have been developed by several companies, as well as by many open source projects. Some of them are applications running in normal personal computer, but many of them are used in the limited resources devices. It becomes critical to develop CSS engines with advanced functionalities but minimal memory consumption.

The aim of this paper is to explain design and implementation of an optimized CSS engine, focusing on the optimization algorithm. The engine's architecture is based on an efficient method to cache CSS style properties that apply to many elements of same XML documents. Those properties are organized in a lexicographic search tree that helps their sharing.

This paper is organized as follows. In Section 2, requirements are defined. In Section 3, two different CSS engine designs are described, trying to explain the differences between the optimized and the basic version. In Section 4, an implementation in real environment is described, and also the integration within an XML user agent. In Section 5, optimization results are reported, while in Section 6, comparisons with related works are discussed. Finally, in Section 7, conclusions are given.

2 REQUIREMENTS

Based on the aim of this paper, the objectives are mainly two with respective requirements. The first one is:

I. Find an algorithm that optimizes CSS engine processing.

The purpose is to abstract a general algorithm that optimizes time and memory consumption, from the most intuitive and simplest algorithm usually used to process CSS style sheets documents. The optimization algorithm should improve performances especially in processing large XML documents.

The second one is:

II. Implement the optimization algorithm as stand-alone CSS engine. Without changing its structure, such engine can be used by any user agent based on any XML languages.

The CSS engine implementation should be accessed through standard interface, and should respect CSS behaviour rules. Both these requirements are retrieved from CSS specifications, developed by W3C. At the moment W3C CSS Working Group is developing CSS Level 3, the third generation of CSS specifications (Bos, B. et al., 2005). Diverse aspects are considered: I. Grammar definition for creating an efficient parser, II. Interfaces definition for accessing stored data, III. Behavior definition for processing, and IV. Style attributes definition for layout rendering. Considering the second objective, only points II and III will be regarded as requirements at this moment, whereas points I and IV will be briefly brought up in the Section concerning integration of CSS engine within a precise client program application.

In next two Sub-Sections, points II and III will explained deeply for a better comprehension of the rest of the paper.

2.1 CSS Interfaces

W3C provides standard interfaces' specifications for CSS engine implementations: DOM Level 2 CSS (CSSOM) (Wilson, C. et al., 2000) and Simple API for CSS (SAC) (Bos, B. et al., 2000). Those two groups of interfaces complete one another, fulfilling the different ways to access the data (as SAX and DOM (Apparao, V. et al., 1998) for XML applications).

The CSSOM interfaces are designed with the goal of exposing CSS constructs to object model consumers, giving the possibility to easily manipulate the "cascade" of style sheets, their rules and their respective properties. They also make the interaction with other object model interfaces easier, for example, DOM. (Wilson, C. et al., 2000) Whereas SAC is a proposal for a standard API for event-based CSS parsing, meant to allow interoperability between the different CSS parsers. (Bos, B. et al., 2000)

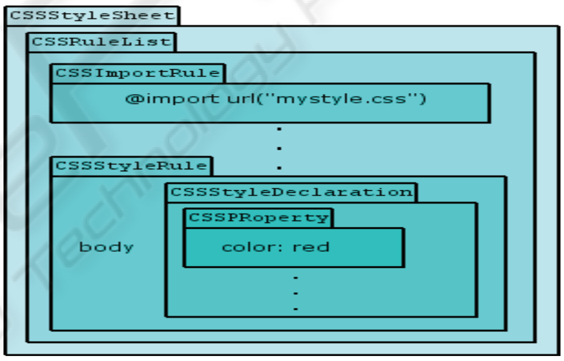


Figure 1: CSSOM interfaces structure.

Figure (1) shows the most common CSSOM interfaces, organized as a logical structure. Other important interfaces are the API for CSS selectors and the API for CSS property's values, both provided by SAC.

2.2 CSS Behaviour

A CSS style sheet is a series of rules that describe how XML elements are to be displayed by the client applications.

CSS classifies several categories of rules. CSS *style* rules are the core of CSS style sheets: they directly define a style for the matched XML elements. Other kinds of rules instead are needed to add supplementary functionalities: for example, CSS *@import* rule, used to import other CSS style sheets, or *Media Queries*, used to create specific presentations for different kinds of output devices.

Task of a CSS engine is to determine, which subset of CSS *style* rules applies to a XML element. This process depends on two factors: Media Queries and CSS *style* rules' weight.

2.2.1 Media Queries

A Media Query is a mechanism to adapt a CSS style to certain devices. It is used to determine a sub set of CSS style sheets in the "cascade" and a sub set of CSS *style* rules inside each CSS style sheets (Glazman, D. et al., 2002). A Media Query consists of a media type and one or more expressions involving media features. For example:

```
@media screen and (color){
  BODY   {font-size: medium;
  background:  silver ;}
  P#8  {font-size: 18pt; color: black;}
}
```

Expresses that those CSS *style* rules apply only to devices with color screen, otherwise they are ignored.

2.2.2 CSS Style Rules' Weight

A weight is calculated for each CSS *style* rule. When several rules apply to the same XML element, the one with the greatest weight takes precedence. Basically, the weight is characterized by "cascading order" and "specificity". According to CSS specifications, many CSS style sheets can affect a XML document at the same time. Cascading is a method that allows importing CSS *style* rules from other CSS style sheets. Rules in imported style sheets have lower weight than rules in the style sheet from where they are imported. Imported style sheets can themselves import and override other style sheets, recursively.(Bos, B. et al., 1998)

Inside every CSS style sheet, CSS *style* rules' weights are formulated by their specificities, which are relied exclusivity on their CSS selectors. The selectors are patterns matching rules that determine which CSS *style* rules apply to elements in the XML document. More specific selectors override more general ones like in the example below:

```
LI {}
/* a=0 b=0 c=1 -> specificity =   1 */
UL OL LI.red {}
/* a=0 b=1 c=3 -> specificity =  13 */
LI.red.level {}
/* a=0 b=2 c=1 -> specificity =  21 */
#x34y {}
/* a=1 b=0 c=0 -> specificity = 100 */
```

Where "a" is the number of ID attributes in the selector, "b" is the number of classes or other attributes, and "c" is the number of element names. In this simplified example, values set of "a","b", and "c" is {0…9}, but, usually, it is much bigger.

3 CSS ENGINE ARCHITECTHURES

The CSS engine described below is a component built to provide the processing of CSS files embedded in Web documents written in any XML language.

Two architectures are explained. The first can be considered basic or more intuitive, because it simply follows the W3C conformance directives. The second is a development of the first one and is based on an optimization algorithm.

There is no particular programming language used in this Section. Most of the algorithms are presented with flow charts and the structure of the program is exemplified with general classes' objects.

3.1 Basic Algorithm

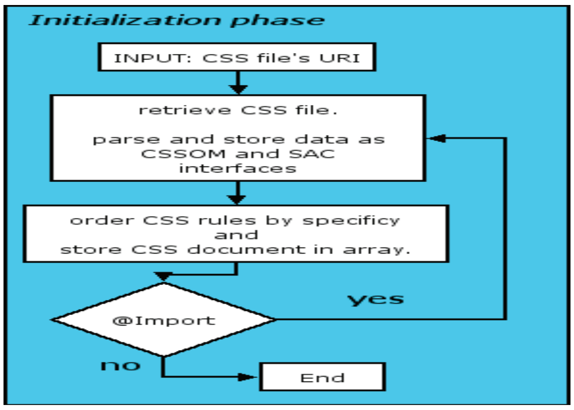
Processing of CSS files is done in two phases. In the first phase, CSS files are retrieved and parsed, and the contained CSS *style* rules are ordered according to their weight. In the second phase, CSS style properties are combined and passed as set of attributes to each element of XML document, according to the matching policy.

3.1.1 Initialization

In this phase of initialization, the implemented CSS engine orders style sheets and relative *style* rules so that, in the second phase, combining all the attributes will be easier.

As explained in Section 2.2, more than one CSS style sheet can influence a document presentation simultaneously. The CSS *@import* rule allows authors or users to import *style* rules from other style sheets. Since it is impossible to know whether a style sheet imports other style sheets before parsing it, it is useful to use a recursive solution in the implementation.

As shown in figure (2), applications that use this CSS package extrapolate and pass the CSS file's URI embedded in XML document. The CSS engine retrieves and parses the respective CSS style sheet, checks whether there are some *@import* rules and eventually extracts from those the URI of the imported style sheet. It retrieves the individual style sheet and parses it in the same way. This recursive procedure continues until there are no style sheets to be imported left.


Initialization phase
INPUT: CSS file's URI
retrieve CSS file.
parse and store data as CSSOM and SAC interfaces
order CSS rules by specificy and store CSS document in array.
@Import
yes
no
End
Figure 2: Initialization phase of the basic algorithm.

CSS style sheets cascade can be seen as a tree. Every element of the tree is an imported style sheet and the root element is the style sheet linked in XML document. Every right sub-tree overrides every left sub-tree, and every root of a sub-tree overrides its children. Consequently, this virtual tree can be traversed using a *Post-order traversal* technique. Thus, all the style sheets are stored in one array following the origin overriding order: the style sheet with less weight is the first element of the array, the root is the last one. An instance of *CSSRuleList* is the object of the module that orders the rules of each style sheet according to their specificities. Once the specificity for every selector is calculated, the *style* rules are ordered using an algorithm based on QuickSort, which has time complexity $O(n\log(n))$.

Media Queries, if present, are evaluated either before importing a CSS style sheet or before ordering CSS *style* rules.

3.1.2 Style Retrieving

Style Retrieving is the phase, in which all the stylistic properties, contained in CSS *style* rules that apply to an XML element, are combined together according to the CSS specifications. The set of the resulting properties are then associated to the matching element.

As shown in figure (3), the first step is to get the XML element that has to be styled. An instance of *CSSStyleDeclaration* interface implementation class is the object used as container of CSS properties. A new instance of it is created for each XML element. The first CSS properties copied inside it are the ones inherited by the XML parent element's style.

The second step is to sequentially traverse all the ordered CSS *style* rules in the ordered cascade of CSS style sheets to analyze, which of those rules applies to the given element. This operation is done using CSS SAC Selectors, comparing those interfaces with the element name and its attributes.

All the *style* rules matched are combined together respecting the overriding rules. This procedure consists of taking the CSS properties contained in each rule's *CSSStyleDeclaration* and checking, one by one, which property has to be copied into the actual *CSSStyleDeclaration*, because none is present yet, or the new property overrides the existing one.

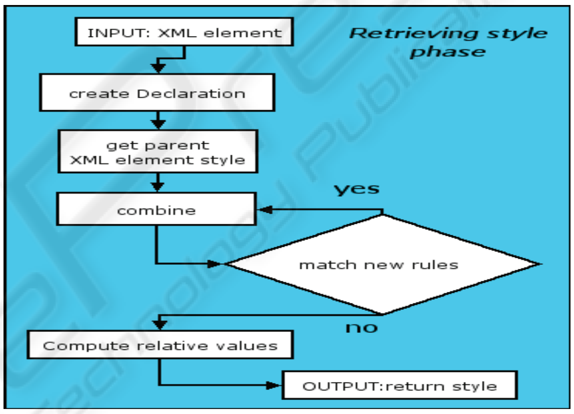


Figure 3: Style retrieving phase of the basic algorithm.

Successively, the engine computes the relative property's values that depend on XML parent element style (e.g., "font-size: *smaller*"). The relative values depending instead on layout values, cannot be calculated at this stage (e.g., "background-position: *center*"). Finally, the *CSSStyleDeclaration* containing all CSS properties is returned as style to the respective XML element.

3.2 Optimized Algorithm

The optimization is based on the fact that many elements of XML documents often match the same attributes of the embedded CSS style sheets. CSS engine implementation may take advantage of this circumstance to optimize time and memory consumption.

The proposed algorithm is based on the data structure shown in figure (4): the *RuleTree*. For each XML element the recursive building process starts always from the empty root node, which is created at the beginning, and will be the same for all the periods, in which the algorithm is running. Every time a new CSS rule is matched, two actions may occur:

I. If one of the children of the actual node has a reference to that matched rule, that child becomes the actual node.

II. Otherwise, a new child node is created, and it becomes the actual node.

Every new node of the tree has a reference to CSS *style* rule matched by the element and a new instance of the *CSSStyleDeclaration* implementation class that will be called in the rest of the paper with the name of *CombinedDeclaration*. In fact, in that *CSSStyleDeclaration* are combined all the CSS properties present in all the CSS rules matched from that node up until the root. Eventually, if the node is the last created for an XML element, a reference to that element is saved.

Figure (4), with XML and CSS source below show an example. As emphasized, XML element “<p class=”c2” id=”warning”>” matches in order CSS *style* rules with selectors “*”, “p”, “”p.c2” and “p.c2#warning”. As also shown, CSS *style* rules matched in same order from different elements are shared: rule with selector “*” is common for all the element of the given document, and rule with selector “p” applies to all XML “<p>” elements, etc.

XML Document:

```
...
<html>
<body>
     <p class="c1">
          <p class="c2">
          <p class="c2" id="warning">
          </p>
          </p>
     <p class="c3"> <span> </span> </p>
     </p>
</body>
<html>
```

CSS Document:

```
*         {font-size: 12pt;}
html      {color: black;}
body     {background-color: yellow;}
p        {font-weigth: bold;}
p.c1     {margin: 2pt; font-weight: normal;}
p.c2      {font-size: 14 pt;}
p.c3      {border-style: solid;}
p.c2#warning      {font-size:18pt;}
span  {color: blue; border-width: 10pt;}
```

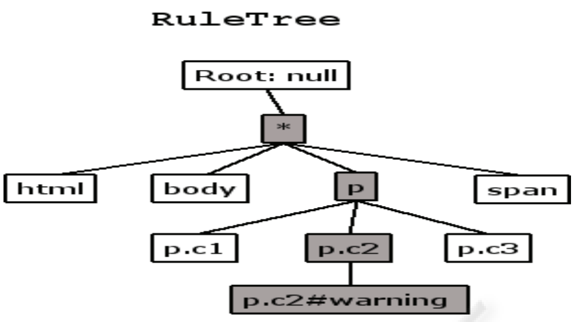


Figure 4: The resulting RuleTree.

3.2.1 Optimized Style Retrieving

The optimization takes place exclusively in the Style Retrieving phase.

Figure (5) is the flow chart of the process, and shows the differences from the unoptimized algorithm of figure (3) (The rectangles with white background represent processes that were used also by the basic algorithm. The rectangles with colored background represent steps that are added by the optimizing algorithm).

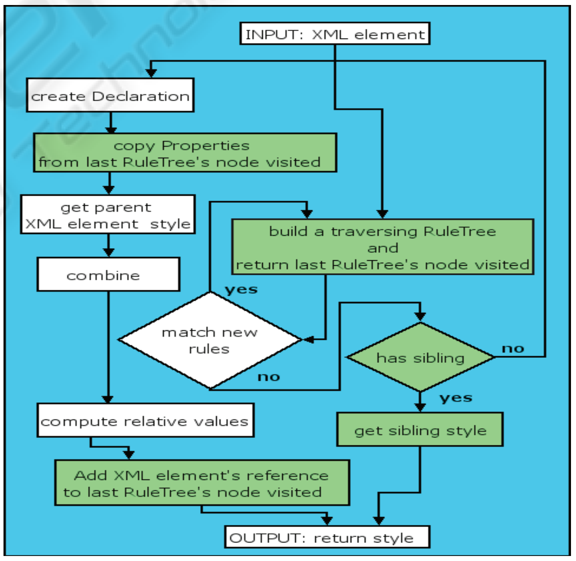

INPUT: XML element
create Declaration
copy Properties from last RuleTree's node visited
get parent XML element style
build a traversing RuleTree and return last RuleTree's node visited
combine
yes
match new rules
no
has sibling
no
yes
compute relative values
get sibling style
Add XML element's reference to last RuleTree's node visited
OUTPUT: return style

Figure 5: Style retrieving phase of the optimized algorithm.

The main difference is that there is no need to create a new *CSSStyleDeclaration* object for each XML element, but, in many cases, just share the existing one. The idea is that if two XML elements match the same CSS *style* rules (i.e., they arrive from root to the same *RuleTree* node) and their XML parent element’s style is the same (i.e. they inherit the same CSS properties) then they will have the same CSS style (i.e., same *CSSStyleDeclaration*).

As shown in figure (5), first step is to create or traverse the branch of *RuleTree* compound of CSS *style* rules matched by the given XML element. This is the point where the most important part of the optimization happens. Action called "has sibling" in the flow chart has two tasks:

I. Check whether the last RuleTree node visited, has a reference to some other XML element that has visited as well that node as last.
II. Check whether the parent's style of that eventual XML element is the same of the parent's style of the XML element that is actually retrieving style.

If the previous conditions are satisfied, the *CSSStyleDeclaration* object of the "sibling" element is returned as style without any other changes. Otherwise, a new *CSSStyleDeclaration* instance is created and last visited *RuleTree* node's *CombinedDeclaration*'s CSS properties are copied into it. Those style attributes are then combined with parent's style and relative values are computed, in the same way as in the non-optimized algorithm. In the end, before returning the final version of the *CSSStyleDeclaration* instance, in the last visited *RuleTree* node is saved as a reference to this XML element, so that in the future it can be taken in account as possible "sibling" element.

3.2.2 Benefits

This algorithm allows three levels of optimization:

I. For XML elements that match in same order CSS *style* rules already matched, there is not needed to recombine all the CSS properties, but just traverse the *RuleTree* and take the style of the last node visited. Sometimes, when *RuleTree* is rather flat and wider than deep, it is faster to combine few rules than traverse it, though.
II. Sharing *CSSStyleDeclaration* object for XML elements that match same CSS *style* rules and inherits same attributes. Usually, large XML documents have many elements of the same kind.
III. For those XML elements that share the same *CSSStyleDeclaration* object, applications may easily cache rendering objects (e.g., Fonts and Color).

4 INTEGRATION WITHIN THE X-SMILES USER AGENT

The CSS engine implemented following the optimizing algorithm was integrated within a more comprehensive client program application: X-Smiles (Vuorimaa, P. et al., 2002). It is a Java based XML browser intended for embedded devices. It is composed of several modules: the XML parser and the Browser Core are always used for every XML document, whereas the rendering process is committed to different modules for each XML languages supported.

The integrated CSS module contains the CSS parser and the CSS engine.

In order to parse CSS files, Steady State Software's CSS Parser was used (Schweinsberg, D., 2004). It was implemented as a package of Java classes, that inputs Cascading Style Sheets source text and outputs structured interfaces. New classes were created and many extensions were made, in order to support new CSS3 features. The parser was created using Java Compiler Compiler (JavaCC) (Javacc Project home, 2005) tool, so many modifications were made to the JavaCC's grammar file. JavaCC is a parser generator that generates parsers in Java. This software program accepts a syntax specification as input, in this case a grammar file, and generates a parser for that syntax as output. Since the content of files written in CSS language are seen as a sequence of tokens, the input grammar file is compound of productions of diverse grammars: context-free grammar production to describe the structure of the tokens and regular expressions to define them. (LOOKAHEAD MiniTutorial, 2005)

The generated parser contains the core components of CSS language compiler, which includes a lexical analyzer and a syntax analyzer. The parser was implemented so that the data parsed is available through the standard interfaces CSSOM and SAC.

The parser was used as the core of the CSS engine. It has the structure and uses the algorithm explained in the previous Section, with the exception that the general classes' objects are substituted by a real programming language like Java. The CSS module communicates with the XML modules of the browser through DOM and CSSOM interfaces. Since DOM's basic functionalities are the same for every XML language, interaction with CSS module occurs always in the same way, satisfying the requirements identified in Section 2.

As already discussed, the rendering process is done by separate modules. Anyway the browser provides a layout for general XML language's documents, styled with CSS. The layout is based on an own tree data structure, where the nodes are called Views, and are created according to styled DOM elements. Everything is painted in a container within browser window.

5 RESULTS

In this Section, the performance of the CSS module is evaluated.

Time and memory consumption of X-Smiles to process and display XHTML files with embedded CSS file was measured. This procedure was repeated several times, always with the same CSS file, but each time with a XHTML file of different dimensions: one, two, four, and eight times the original source. The documents used are located at http://www.csszengarden.com and the author is Petr Stanciek. Four series of data were created trying to separate CSS module's performance from general performance of the browser and to demonstrate optimized versions' improvements from basic version.

The computer used for this test had a Pentium 4 processor with 2.80 GHz and 1.00 GB of RAM and Windows XP operating system.

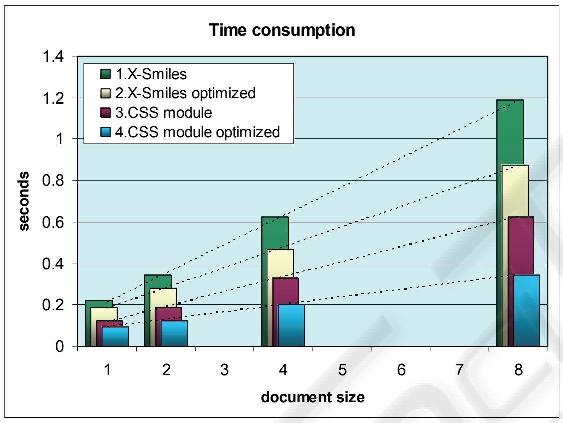


Figure 6: Performances- Time consumption.

In figure (6), series "1.X-Smiles" and "2.X-Smiles optimized" represents the time consumption for all the processes of the browser. Both series grow linearly, but with different angular coefficient. With the biggest document tested, optimized version is 26% faster. Series "3.CSS module" and "4.CSS module optimized" represents the time consumed by the browser only to parse and store the XHTML documents with embedded CSS style sheet and add style for each element of the DOM tree (using the CSS module). Also, in this case the two series grow linearly: the optimized version of CSS module takes always less time, arriving to be 45% faster with the biggest document tested.

In figure (7), the series concerning memory's utilization are reported. For every document, the XML parser and the non-optimized DOM implementation take the largest percentage of memory, so this consumption is subtracted from every series, trying to better illustrate the improvements. The most remarkable aspect is that "4.CSS module optimized" series does not grow, but remain constant, independently of the XHTML document's dimensions rising. With the biggest document tested, the optimized version of CSS module uses 85% less memory.

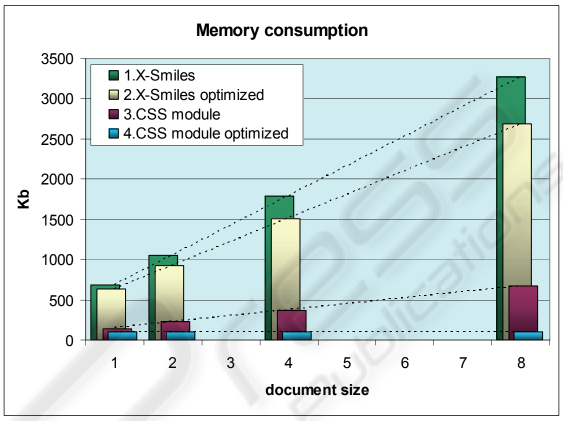


Figure 7: Performances- Memory consumption.

6 RELATED WORK

Nowadays, many CSS engine implementations are available. Since this paper is focused on an optimization algorithm, comparisons are possible only with those applications that publish their source code.

Mozilla Web browser is an open-source software project that supports many XML languages and CSS as well. Its CSS engine uses the same data structure used by the optimization algorithm explained: the lexicographic search tree that helps the sharing of the computed CSS styles objects (Baron , D., 2005). Meanwhile the processing algorithm is dissimilar: for each node of the search tree is not stored any *CombinedDeclaration* computed from the root until the actual node, but, at opposite, the CSS properties are recalculated every time from the actual node to the root.

Konqueror Web browser (Konqueror, 2005) is based on a HTML rendering engine called KHTML. It supports CSS specifications and its CSS engine is based on a processing algorithm that is pretty similar to the "basic algorithm" explained. It has other kinds of optimizations (e.g., it parses only *CSSStyleDeclarations* of the matched CSS *style* rules) that are not in contrast with the "optimization algorithm", but they can be complementary.

Squiggle (The Apache Software Foundation, 2005) is the browser of the Batik project. It is meant for SVG documents style by CSS style sheets. As a consequence, its CSS engine is specifically optimized to be used within SVG applications (e.g., it gives priority of computation to those CSS properties that are most used within SVG documents). Anyway, also in this case, the "optimization algorithm" could be added without modifying the existing optimizations.

7 CONCLUSIONS

In this paper, implementation of an optimized CSS engine was discussed.

It can be used for every XML applications that need to adapt the layout of their Web services depending on different devices that often have limited environments.

The focus was on an optimization algorithm that reduces memory and time consumption to process CSS documents. The idea for the optimization was that usually in a XML document, many elements match the same set of CSS *style* rules, so the final set of CSS style properties can be cached and shared, and not recalculated every time. This algorithm is based on a search tree data structure of matched CSS *style* rules. It was described both with flow charts and general classes' objects, in order to emphasize the algorithm processes independently of the implementation chosen.

This algorithm allows three levels of optimization (cf. Section 3.2.2); two are intrinsic in the CSS engine and in its structure and one depending on the XML applications' layout engines.

Based on the optimizing algorithm, the CSS engine was implemented in a real environment, using Java as programming language. To evaluate such optimization, a browser was used, that supports many XML languages such as XHTML, SVG, XForms and SMIL. In order to enable a straightforward integration, the CSS parser exposes the CSS data through standards interfaces (i.e., CSSOM and SAC).

Considering the innumerable factors that could be taken in account, no minimum threshold of performances requirements were defined. For the tests we used common well formatted XML documents styled with CSS style sheets found at site "CSS Zen Garden". The optimization results are remarkable: measurements show that with largest document used, the optimized CSS engine can be 45% faster and spare 85% of the memory.

REFERENCES

Apparao, V. et al., 1998. Document Object Model (DOM) level 1 specification – version 1.0, W3C Recommendation.

Ayars, J. et al., 2001.Synchronized Multimedia Integration Language (SMIL 2.0), W3C Recommendation.

Baron , D., 2005. Mozilla Style System Documentation available online at http://www.mozilla.org/newlayout/doc/style-system.html.

Bos, B. et al., 1998. Cascading Style Sheets, level 2 CSS2 Specification, W3C Recommendation.

Bos, B. et al., 2000. SAC: Simple API for CSS," W3C Note.

Bos, B. et al., 2005. CSS Level 3, available http://www.w3.org/Style/CSS/current-work.

Didier, M., 2000. *A Family Affair*, O'Reilly XML.com.

Dubinko, M. at al., 2003. XForms 1.0, W3C Recommendation.

Ferraiolo, J., 2001. Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation.

Glazman, D. et al., 2002. Media Queries, W3C Candidate Recommendation.

Jacobs, I., 2005. About the World Wide Web Consortium (W3C) available online at http://www.w3.org/Consortium/, referred Jun. 2005

Javacc Project home, 2005, available at https://javacc.dev.java.net/.

Konqueror, 2005. available at http://www.konqueror.org

LOOKAHEAD MiniTutorial, 2005, available at https://javacc.dev.java.net/doc/lookahead.html.

Pemberton, S. et al., 2002. XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), W3C Recommendation.

The Apache Software Foundation, 2005, Squiggle the SVG browser, available at http://xml.apache.org/batik/svgviewer.html

Schweinsberg, D., 2004. CSS Parser, available at http://cssparser.sourceforge.net.

Vuorimaa, P. et al., 2002. A Java based XML browser for consumer devices. *In the 17th ACM Symposium on Applied Computing*, Madrid, Spain.

Wilson, C. et al., 2000. Document Object Model (DOM) Level 2 Style Specification Version 1.0, W3C Recommendation.

Wolter , R., 2001. XML Web Services Basics, Microsoft Corporation.

Yergeau, F. et al., 2004. Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation.