Smart Caching for Web Browsers

Kaimin Zhang
University of Science and Technology
Hefei, China
coming@mail.ustc.edu.cn

Lu Wang
Hong Kong University of Science and Technology, Hong Kong
luwang@cse.ust.hk

Aimin Pan, Bin B. Zhu
Microsoft Research Asia,
Beijing, China
{aiminp, binzhu} @microsoft.com

ABSTRACT

In modern Web applications, style formatting and layout calculation often account for a substantial amount of local Web page processing time. In this paper[1], we present two novel caches, smart style caching and layout caching, for Web browsers. They cache stable style data and layout data for DOM (Document Object Model) elements, and apply directly without re-calculation when the same data is subsequently processed, possibly across different visits of a Web page. Redundant computations in both style formatting and layout calculation could be eliminated, resulting in more efficient local Web page processing. The proposed cache schemes are still applicable and effective even there are changes in the DOM structure or style rules of a Web page. Experiments on the Web pages of the Top 25 Web sites show that, in a subsequent visit of the same Web page, the smart style caching scheme could reduce the style formatting time by about 64% on average, and the combination of both caching schemes could reduce the layout calculation time by about 61% on average, with about 46% overall performance improvement on the local Web page processing time. For the overall performance when networking, Web servers, and local Web page processing were all included, our cache schemes could improve up to 56% when browsing these Web sites on a desktop PC and up to 60% when browsing on a netbook.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Information Browsers; I.7.m [**Document and Text Processing**]: Miscellaneous.

General Terms

Performance, Algorithms.

Keywords

Web, Browser, CSS, cascade style sheet, caching, JavaScript.

1. INTRODUCTION

Web tends to become a platform for modern applications. An increasing amount of data has been moved to the cloud as cloud computing is becoming a reality. In Web applications, the client side is a Web browser or a thin application with a Web browser engine embedded. Modern Web applications, e.g., Bing Maps [1] and Google Docs [2], have become increasingly complex and powerful that can rival desktop applications. This poses a challenge to Web browsers. A Web browser with a lousy performance would not be able to provide a user experience comparable to desktop applications. Current Web browsers may not meet this demanding performance requirement yet, particularly when rendering complex Web pages.

In loading a Web page, a Web browser does basically two tasks: fetching the Web content through the Internet and performing local computations to process the content. Networking is a performance bottleneck if data transmission is slower than local processing. This occurs typically when a lot of data need to be fetched over a network of limited bandwidth available. Network bandwidths have been improved dramatically in recent years. For example, 3.5G mobile networks, already available in some countries, provide a bandwidth up to 14.4Mbps for mobile Internet users. In addition, local cache has been widely used by modern Web browsers to reduce the amount of data that needed to be fetched over the Internet. On the other hand, Web pages become increasingly more complex that substantial computation resources are required to parse, format, and render properly [3]. Local Web content processing may not get much benefit from recent advances of hardware processing power which is mainly through parallel processing by including multiple cores in a single chip but the chip frequency remains the same or even reduced as compared to single-core chip. Web content is processed essentially in a single thread manner in order to get a proper result. It is still unclear in practice how to use parallelization capacity in a chip to render Web content, which is a new research topic [3]. In conclusion, the trend is that local Web content processing plays an increasingly important role in the performance of a Web browser at the cost of diminished impact from networking. In other words, Web browsing tends to be computation-intensive instead of network-intensive.

The results reported in [4] by profiling popular Web sites as well as our own experiments with the Webkit engine [5] indicate that the combination of layout calculation and style formatting account for more than half of the total computation time in a local Web page processing. Many modern Web pages use the cascade style sheet (CSS) [6] heavily due to its flexibility in supporting various visual effects. Computing style properties and applying them to Document Object Model (DOM) [7] elements are essentially a recursive, time-consuming process. Current Web browsers have to perform both tasks every time when a Web page is browsed. In addition, any change in style properties of an HTML element leads to re-calculation of its layout, which may affect its descendant elements in the DOM tree.

Existing efforts to reduce style computation include providing a guideline for writing JavaScript [8][9] and optimizing layout engines [10]. These approaches can minimize the effects of DOM modifications and localize the reflow scope, particularly when JavaScript code manipulates DOM elements [11].

In this paper, we propose a novel method to improve Web browsing performance by caching intermediate results in vital stages of Web page processing and applying the cached results whenever applicable in subsequent processing of the same data to avoid

[1] This work was done when Kaimin Zhang and Lu Wang were working as interns at Microsoft Research Asia.

repeating the same local computations. Repeated local computations typically occur when revisiting a Web page. They may also occur when processing a new Web page due to redundancy in the Web page. In particular, we construct a style cache and a layout cache to record the stable results of style formatting and layout calculation, and apply direct directly without re-calculation when the same data is subsequently used for style formatting or layout calculation. The combination of style formatting and layout calculation typically accounts for a substantial portion of the local Web page processing time in a modern Web application. Our caches can effectively eliminate redundant calculations in those operations, resulting in improved browsing performance. There are two challenges in this method: 1. What information is stable across different visits of a same page and also requires heavy computations to generate? 2. How to make cache still effective when there are changes in a Web page? Our cache schemes address these two challenges well. Our cache schemes can identify when cached data can be applied, and the caches are dynamically updated. They are still effective when there is reasonably large gap in time between two visits of a same Web page, and also when there are changes in the DOM structure or style rules of a Web page.

We have implemented a prototype of the proposed caching schemes based on Webkit [5], an open-source browser engine. Our experimental results on the Web pages of the Top 25 Web sites from comscore.com (2008) show that, in a subsequent visit of the same Web page, the smart style caching scheme could reduce the style formatting time by about 64% on average, and the combination of both caching schemes could reduce the layout calculation time by about 61% on average, with about 46% overall performance improvement on the local Web page processing time. For the overall performance when networking, Web servers, and local Web page processing were all included, our cache schemes could improve up to 56% when browsing these Web sites on a desktop PC and up to 60% when browsing on a netbook. The experiments on two typical dynamically changed Web pages show that most data in the style and layout cache can be valid in several hours. For some Web sites, they may be valid for several days to several weeks, or even longer.

This paper has the following major contributions: We propose the first style caching scheme and layout caching scheme for Web browsers to effectively reduce redundant local style and layout computations. Both cache mechanisms are based on the workflow of Web page processing and Web standards like HTML, DOM and CSS. They are therefore applicable in any Web browser. Furthermore, the two caching schemes are still effective even if the styles or content of a Web page is dynamically modified over time.

The rest of this paper is organized as follows. In Section 2, we introduce briefly the background of local Web page processing in a Web browser, and then describe the main ideas behinds our method. The smart style caching scheme and the layout caching scheme are presented in detail in Section 3 and Section 4, respectively. The experimental results are reported in Section 5. Discussion and future work are presented in Section 6. Related work is presented in Section 7. We conclude the paper with Section 8.

2. BACKGROUND AND OUR METHOD

Web applications are built on top of HTML along with other Web standards [12] such as CSS and DOM. Web browsers process Web pages based on the syntax and semantics specified in the standards. This leads to the result that most browsers have a similar framework and internal representation of a Web page. In this section, we introduce briefly such a general framework, and then discuss how caching mechanisms can be introduced in the framework.

2.1 Workflow of Web Page Processing

Figure 1 shows the general workflow that a Web page is processed by modern Web browsers. After receiving a Web page, either from a remote Web server or a local store, a Web browser parses the page in the form of HTML data, and represents the parsed HTML data as a DOM tree in memory. The style properties are then generated for the elements in the DOM tree. These properties determine how the elements are presented in the screen. In order to render them, the browser must trigger a process to calculate the layout for each element in the DOM tree. It can then render those elements correctly to the screen.

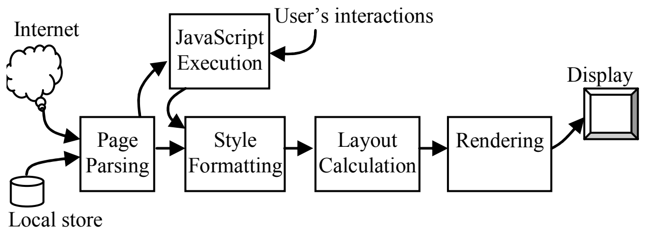


Figure 1. Workflow of Web page processing

These stages may not be done strictly one stage after another in the order as shown in Figure 1. They may occur concurrently in order to provide a better user experience, allowing a user to see a partially rendering result before finishing download and parsing of the whole page. This processing is essentially a sequential process since any change in a previous stage will incur execution of the following stages.

Scripts in Web pages are often in the form of JavaScript code because JavaScript is supported by almost all the existing Web browsers. The JavaScript code can be triggered either in the stage of page parsing or by user's interactions. If the JavaScript code manipulates DOM elements, the stages of style formatting and layout calculation may also be triggered in order to render the elements correctly on the display. These triggered operations are most likely a reason why JavaScript code is executed inefficiently.

2.2 Caching in Web Page Processing

The data flow is formed based on the workflow of page processing in a Web browser, as shown in Figure 2. The original HTML data is parsed to form a DOM tree in memory for a Web page, which should comply with W3C DOM standard [7]. Then the styles are applied to the elements in the DOM tree after the style rules in the page are processed. This often forms a data structure separate from DOM, for example, called a render tree in Firefox [10]. Each node in the render tree has a corresponding element in the DOM tree. Its purpose is to make DOM elements visible on the screen. The render tree is further processed to calculate the layout for each node in the stage of layout calculation. Finally, each node in the render tree is rendered to the screen in the rendering stage.

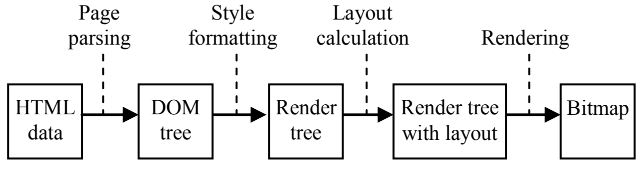


Figure 2. Data flow of Web page processing

Now we look at possibilities to cache data in the workflow of Web page processing. One extreme is the most straightforward HTML data cache (i.e. HTTP cache), which has been widely supported by almost all Web browsers. Another extreme is a render cache, which is the bitmap of the rendering result of a Web page. These two types of caches are effective if the cached Web pages do not change frequently. Note that the render cache is invalid even if only the window size changes for the same URL. A large number of Web pages, however, changes quickly because of real-time news, ads, and other dynamic data sources. In this case, the HTML cache and the render cache do not work. Pages requested via a same URL at different times are typically not exactly the same, but most contents of these pages may be identical. If we build caches in the middle of the data flow in Figure 2, the cached data may be still useful if the unchanged parts can be identified, then the subsequent computation for the cached data can be saved. Therefore, different types of caches can be developed for different stages. These caches are partially effective when a Web page changes over time: only the cached data of unchanged part is valid.

In this paper, we propose two types of intermediate caches. One is called a smart style cache, which contains the results after the stage of style formatting. The smart style cache works in a granularity of DOM elements. The cached data for a DOM element is valid only if both its path to the DOM root and its style properties do not change when users visit a Web page again. The content in a DOM element is not taken into account when checking validation of the cached elements. The other is called a layout cache, which contains the intermediate results after the stage of layout calculation. Unlike the style cache, the layout cache is content based. The data for a DOM element is used to both compute the cache and check validation of the cached layout data. If the cached layout data for an element is valid, it is passed to render the element without any re-calculation.

These two types of caches are chosen to save computations for style formatting and layout calculation of DOM elements. In the workflow of Web page processing, a change of style properties may trigger computations at the subsequent stages shown in Figure 1, including layout calculation and rendering when a page is processed. Thus, caching the style data can not only reduce the frequency of style calculation, but also computations at the subsequent stages. As reported in [4], the layout calculation accounts for the most of computing time among all the stages in the workflow. This agrees with our measurements with the Webkit engine. A layout cache is expected to reduce the time needed for layout calculation, which motivated us to build the layout cache.

3. SMART STYLE CACHING

Most modern Web browsers comply with CSS Standard 2.1 [6] in interpreting style information for HTML elements. In this section, we first briefly introduce the process of style formatting, and then describe our style caching scheme and the key algorithms in the scheme.

3.1 Style Formatting for Web Pages

There is one CSS style sheet for each web page. A CSS style sheet consists of a set of CSS rules. Each CSS rule consists of two parts: a selector and a declaration. The selector of a CSS rule determines which kind of elements will match the rule. The selector can be either simple, such as ID selector and class selector, or complex, such as the ones that refer to any attribute of a DOM element. Therefore, developers of Web pages can define a scope of elements via a selector and then assign specific style values to them. In practice, this kind of capability can be exploited to achieve some special visible effects. However, one side effect is that a browser must deal with possible complex selectors in order to render a Web page correctly. The second part, i.e., the declaration of a CSS rule, is a set of values of pre-set style properties, which determine the way how the selected element will look like. For example, in the CSS rule "p em { color: red }", the selector part is "p em", which indicates that all the <em> elements which are a descendant of a <p> element are selected as the target elements of this rule. The declaration part, in this example, is "{ color: red }", which defines the color property of all the selected elements as red.

CSS formatting usually happens when a browser needs to determine the style of a newly created or modified element. It typically consists of two steps. First, the browser checks each CSS rule against the element. The selector of a rule determines whether the rule is a match to the element or not. Second, all the matched rules are applied to the element in a proper order defined in the CSS specification, to generate the style properties of the element. Basically, the applying process is to collect the declarations of all the matched rules and then merge them. Since one style property may appear in multiple matched rules, the value declared in the rule with highest priority is used as the final result.

There are lots of style properties which may affect the visual effect of an element. It is often tedious for Web authors to specify each property of an HTML element in CSS. Fortunately there is a mechanism called derivation in CSS, which can be used to determine the value of properties that are not explicitly declared. If a style property is not defined for an element, its value is either derived from the style of its parent element, or is set to a default value by the browser, depending on the type of that property. This requires that the style of a parent element is always determined before all of its children. Furthermore, a browser should always define a default style sheet (called UA rules).

Therefore, the process of style formatting depends on not only the set of style rules and DOM elements, but also the structure of the DOM tree. DOM structure must be taken into account when implementing or optimizing the algorithm of style formatting.

In order to concretize the process of CSS formatting, let's look at an example. Suppose there is an HTML file as following:

```
<html>
<head>
  <style>
    p em { color : red  }
    p   { color : green }
    em  { color : blue  }
  </style>
</head>
<body>
  <p> The first part <em> The second part </em> </p>
</body>
</html>
```

In the example, there are three rules which are bracketed by the <style> and </style> tags. In order to render the <em> element, a browser needs to determine its style. The browser first checks all the CSS rules provided in the HTML file against the <em> element, and finds that both the first and the third rules are a match. Those two rules as well as the default rules provided by the browser are then merged according to the CSS specification. In this case, only the 'color' property is specified by the page author, while other style properties are set as a default value. In this example, both matched rules specify the color property, and according to the CSS specification, the value declared in the first rule is used because the first rule is more special and thus has a higher priority. Therefore the text “The second part” is in red.

3.2 Smart Style Caching Scheme

If a Web page, including its content and style sheet, does not change over time, a simple yet effective style caching algorithm is to record the style properties for each element in a page at the first visit to the page, and then to restore them at a subsequent visit to the same page. No style calculation is needed for the subsequent visit. Obviously, this is an ideal case, and this algorithm works for only a small percentage of Web pages.

In practice, most Web pages have dynamic content, e.g. live news, search results, or ads. Therefore a practical algorithm must address possible changes in the DOM tree and CSS rules for a page. The goal is to reuse the style properties for the DOM elements that have not changed, compute the style properties only for new and modified elements in order to minimize re-calculation.

Our smart style caching (SSC) scheme takes only the following selectors into consideration: the selectors involving ID, Class, TagName attributes of a single element as well as the basic descendant & child relationship in the DOM tree. These types of selectors are referred to as normal selectors in this paper. This means that our SSC scheme does not cache any element that is selected by any non-normal selector. It is possible to include elements selected by non-normal selectors in our SSC scheme. This is a tradeoff between the caching scope and the complexity of caching implementation. According to our statistical results of occurrences of different selectors in the homepages of the Top 25 Web sites from comscore.com (see Section 5.1 for the list), more than 95% of selectors are normal selectors. Therefore, we decided to cache only the elements selected by normal selectors.

In order to identify the elements covered by our SSC scheme, we construct an SSC tree, which is similar to a DOM tree but only the structure information of the DOM tree is recorded. Only the DOM elements matched by normal selectors have corresponding elements in the SSC tree. In a SSC tree, the sibling elements with the same triple <ID, class, TagName> are merged into one element. Figure 3 shows an example of SSC tree. In this example, the first and second <li> elements in the DOM tree share the same <li> SSC element since “foo” is not a style property that would affect identification of an SSC element. However, since the third <li> element has a special value for the “class” property, there is a separate SSC element corresponding to that element, as shown in Figure 3(c).

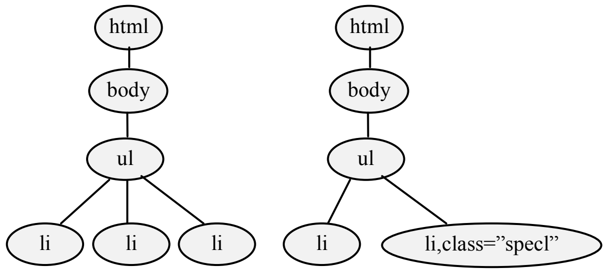
The style cache for a Web page consists of:

- The rule set of its cascading style sheet;
- The SSC tree, and the style properties and matched rule list for each element.

For any element in the SSC tree, if no change is detected for its matched style rules during a page loading, the style properties are retrieved from the style cache recorded in a previous visit to the page; otherwise the style properties need to be re-computed. Note that if a DOM element is selected by any non-normal selector, then it is not recorded in the SSC tree or style cache.

```
<html>
<body>
   <ul>
      <li> First Line </ li>
      <li foo=”bar”> Second Line </ li>
      <li class=”specl”> Special </ li>
   </ ul>
</ body>
</ html>
```

(a) HTML



(b) DOM tree (c) SSC tree

Figure 3. A Web page and its DOM tree and SSC tree

3.3 Key Algorithms in SSC Scheme

In this section, we first describe the algorithm of maintaining SSC elements for DOM elements, and then discuss how the SSC scheme tolerates changes in the DOM tree or CSS rule set of a Web page. These algorithms can guarantee correctness of the final DOM elements, i.e. the style properties for each DOM element are equivalent to the original ones when the SSC cache is not used.

3.3.1 Maintaining SSC elements for DOM elements

According to the definition and generation of the SSC tree for a Web page, a DOM element has exactly one corresponding SSC element while an SSC element may correspond to one or more DOM elements. For each SSC element, we store the necessary properties (i.e. ID, Class, and TagName) that are used to identify elements, as well as the cached style properties that would be retrieved and applied to the identical elements in subsequent visits to the same page.

Given a DOM element, say E, the corresponding SSC element is located or created in the following way:

> *Check if E is the root of the DOM tree. If not, since E’s parent EP should have already been checked, we know EP’s corresponding SSC element, EP_{SSC}. Then check the child elements of EP_{SSC}. If we find an element with exactly the same properties (ID, Class and TagName) as E, then it corresponds to E; otherwise, E is treated as a new element (it could also be an existing but modified element). Finally create a corresponding SSC element for E with E’s properties (ID, Class and TagName) and attach it to the SSC Tree as a child of EP_{SSC}.*
>
> *If E is the root and the SSC root element does not match E, the whole cached SSC tree is invalidated and a newly created SSC element as its root. Otherwise, if the SSC root element matches E, then it is the corresponding element for E.*

Note that we have assumed here that any parent element in the DOM tree is always processed before its child elements. This assumption holds when a browser is loading a Web page since the DOM tree are constructed with elements in pre-order in HTML.

Once we have identified the corresponding SSC element for E, the style properties are retrieved from the SSC element. If it is a newly created SSC element, then E's style properties are calculated and recorded into the new SSC element. In this way, we can ensure that all the style information of an element E that has been calculated during a visit to a Web page could always be retrieved in subsequent visits if E appears in the page again.

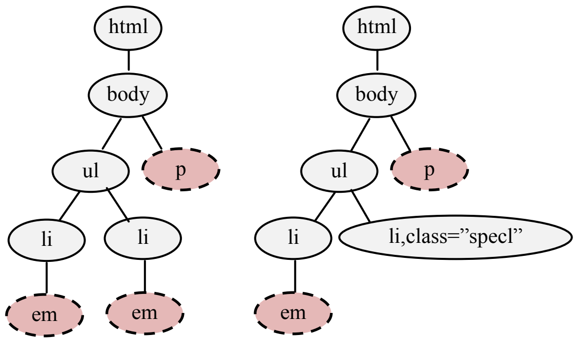
3.3.2 Tolerating Changes in DOM tree

The maintenance algorithm described in the previous section implies that a DOM element and its corresponding SSC element have the same path to the roots in the DOM tree and SSC tree, respectively. If any element is moved in the DOM tree, it can no longer be matched in the original SSC tree, as well as its descendant elements in the DOM tree.

Let's take an example to see how our SSC scheme tolerates changes in the DOM tree. Suppose the Web page shown in Figure 3(a) is modified to that shown in Figure 4(a). The corresponding DOM tree and SSC tree are shown in Figure 4(b) and Figure 4(c). Element with dotted border in Figure 4 are new elements in the tree.

```
<html>
<body>
  <ul>
    <li> This is <em>First</ em> Line </ li>
    <li> <em>Second</ em> Line </ li>
  </ ul>
  <p>
    New paragraph
  </ p>
</ body>
</ html>
```

(a) HTML



(b) DOM tree (c) SSC tree

Figure 4. A modified Web page and its DOM tree and SSC tree

In this example, note that:

1. The <em> and <p> DOM elements are new elements, therefore new SSC elements should be created for them;
2. The two <em> DOM elements share a same <em> SSC element, although their parents are different. In fact, they are 'identical' from the point of view of 'style rules' (including the rules along the path from the root to them);
3. The old <li> SSC element still corresponds to the two <li> elements in the new page;
4. The old <li> SSC element with special value of the class property still exists in the SSC tree because such element may appear again in future visits. To make the new SSC tree compact, this kind of unused elements can be easily removed after the entire DOM tree is traversed.

3.3.3 Tolerating Changes in CSS Rule Set

To tolerate changes in a CSS rule set, the SSC scheme records not only the final style properties for each element but also the list of matched rules for it. Both of them are stored in the corresponding SSC element of a DOM element.

In page processing, two sets of CSS rules are involved. One is the rule set retrieved from the style cache, either recorded in a previous visit to the same page or an empty set if there is no style cache, which is denoted as R_{cache}. The other is the set of CSS rules for the current Web page, denoted by R_{cur}. Note that since Web pages are usually downloaded and processed incrementally, R_{cur} is also constructed incrementally. Therefore, we cannot determine the missed rules (i.e. those are in R_{cache} but not in R_{cur}) until the page is completely processed. We can, however, always identify the new rules (i.e. those are in R_{cur} but not in R_{cache}) immediately once they are added into R_{cur}. When a page is being loaded, the following process is executed for each element:

- If there are no new rules in R_{cur}, the cached style properties for the element are employed directly without any re-calculation;
- Otherwise, all the new rules are examined against the element, and the matched ones are inserted into the list of matched rules of the element at proper positions. Finally, the new list of matched rules is used to generate the style properties for the element. In this way, we can avoid re-checking the selectors of the existing rules in R_{cache} .

As soon as the page is loaded completely, we can identify which rules are missed, i.e. the rules in R_{cache} but not in R_{cur}. Then we process the elements affected by those rules as following:

- If there is no missed rule, do nothing;
- All elements whose matched rule list in the style cache contains any of missed rules need to be re-formatted. For each element, the missed rules are eliminated from its matched rule list, and then its style properties are re-computed.

In this way, we can always identify the same rules that appear in both the current visit and a previous visit, and avoid duplicated calculations for the elements of which the matched rule list has not changed. Furthermore, the new rules for the current visit are stored in the style cache to be retrieved for future visits to the same page.

4. LAYOUT CACHING

The layout caching is designed for reducing time-consuming layout calculation by reusing the layout results in previous visits to a Web page. The result of layout calculation for a visible element is recorded, along with the necessary information for checking its validation later. Unlike the style cache, which depends on only the style properties of each DOM element and its path to the root, the layout data for a DOM element is content-dependent. For example, in order to calculate the layout of a piece of text, the content of the text must be taken into account.

According to the data flow of page processing, as shown in Figure 2, the layout calculation is based on the render tree generated after style formatting. Unlike the DOM tree, the render tree is not standardized, but we can think that the render tree has a hierarchical structure similar to the DOM tree, and it includes render objects for only the visible elements in the DOM tree. Our layout cache is built atop the render tree, in a similar way that the SSC cache is built atop the DOM tree. In this section, we first present our layout caching scheme, and then describe its validation checking algorithm.

4.1 Layout Caching Scheme

The layout calculation for a render object is done by a certain type of layout operation. Our layout caching scheme records the results of layout operations. Therefore, when a layout operation is needed for a render object, we determine if it is recorded in the layout cache or actual execution is needed. If there is a layout operation in the layout cache which matches the current one, then its cached result can be retrieved and returned directly; otherwise, it needs actual execution. The validation checking algorithm will be described in the next section.

Like in the style cache, in order to reuse the cached layout results, identical render objects must be first identified in the render tree. A straightforward method is to build a companying render tree like the SSC tree described in Section 3.2. However, by using the existing SSC tree, there exists a simple and efficient method without any companying render tree if only the elements that are cached by the style caching scheme are taken into account by the layout caching scheme.

Since each render object is associated with one DOM element from which it is generated, and the DOM element is associated with one SSC element, a render object is also associated with one SSC element. Therefore, we can record the render object along with its layout result in its associated SSC element. In order to identify a render object in the layout cache, we find its associated DOM element, and then find the associated SSC element of the DOM element with the algorithm described in Section 3.3.1. Each SSC element may associate with a set of cached render objects, and usually the set is small since typically there are only a few render objects generated from the associated DOM elements. Finally, the identical cached render object, if exists, can be identified from this set by matching its type and content.

In order to balance the efficiency and complexity of the layout caching scheme, the floating objects, complex render objects such as render media and render table, are not included in the layout cache. We only cache the layout results for several types of frequently used render objects, including render box, render block, render button, render text control, render text, render image and inline render objects. Our profiling results with Webkit for the homepages of the Top 25 Web sites (see Section 5.1 for the list) show that more than 70% of layout calculation time is spent on these kinds of objects.

4.2 Validation Checking of Layout Operations

In order to determine validation of the cached result for a layout operation on a render object, there are four conditions to check against:

- **Global Information of the Browser.** This includes the size of the browser's window and the theme of the browser. If the global information changes, all cached results are invalidated.
- **Parent-child Relations in the Render Tree.** In the render tree, the layout calculation is a top-down and recursive procedure, starting from the root of the tree. The layout calculation for a child element depends on its parent's layout result. For example, the outer box's size affects the layout of all its inner boxes, which are the children of the outer box in the render tree. Therefore, a cache miss on a render object causes cache miss on the entire sub-tree rooted at this object.
- **Style of the Render Object.** Any change on the style invalidates the cache for the render object.
- **Content of the Render Object.** The layout calculation for a render object depends on its content. However, for certain types of render objects, the layout calculation may only be sensitive to a part of their content. For example, to calculate the layout of an image, only the size of the image is concerned. Therefore, by extracting and checking only the layout-related content, the hit rate of the layout cache could be improved.

While our style cache can tolerate changes in the CSS rules in a Web page with partial re-calculation. The layout cache, however, typically does not tolerate any changes, as we have seen above. This is a big difference between the two cache schemes.

5. EXPERIMENTAL RESULTS

We have implemented a prototype of the proposed caching schemes based on the Webkit layout engine (version 1.1.5-GTK) [5] running on the Linux platform. In our experiments to compare the browsing performances with and without using our proposed cache schemes, GtkLauncher, a simple and lightweight Web browser packaged with Webkit GTK, was used. In this section, we first describe the experimental environments and the Web sites employed in the experiments. Then we present the performance results of both the style caching scheme and the layout caching scheme, as well as the overall performance. Finally, we report the effectiveness of our caching schemes on several typical dynamic Web pages.

5.1 Experimental Setup

The homepages of the Top 25 Web sites from comscore.com (2008) were used in our experiments. These Web sites are listed in Table 1. We have conducted two types of experiments to study browsing performance. The first type of experiments is to remove the network impact on the browsing performance so that only the local Web page processing performance was compared. This was done by fetching the Web pages with the WGet utility [13] and storing them into a local disk before the experiments. During the experiments, GtkLauncher browsed the locally stored offline Web pages with different settings in Webkit to enable or disable our caching schemes. Note that there might still exist some network traffics such as Ajax requests during the experiments. The second type of experiments is to compare the actual browsing performance on both desktop PC and netbook with the impact of the networking and Web servers included. In both types of experiments, we evaluated only the process of page loading, so that GtkLauncher could shut down automatically when receiving a *load-finished* signal from Webkit.

The experimental desktop PC was a mainstream PC with an Intel Dual Core 2.13GHz processor and 2GB of DDR2 RAM, and the experimental netbook was a typical one with an Intel single core 1.66GHz Atom processor and 2GB DDR2 RAM. Both computers ran the 32-bit Ubuntu 9.10 Linux with all latest patches installed. The window size of a browser would affect the performance of rendering. In our experiments, the window size was fixed at 800 by 600 pixels. Each experiment was repeated 20 times. For the second type of experiments, initial rounds of measurements dropped since they typically had a large fluctuation on the performance due to Internet cache.

Table 1. Top 25 Web sites from comscore.com (2008)

www.google.com	en.wikipedia.org	www.mozilla.com
www.google.cn	www.myspace.com	www.apple.com
www.aol.com	www.qq.com	www.adobe.com
www.xunlei.com	buzz.blogger.com	www.amazon.com
www.facebook.com /barackobama	www.ask.com	www.microsoft.co m
www.yahoo.com	www.163.com	www.sina.com.cn
www.youtube.com	www.wordpress.com	www.ebay.com
www.baidu.com	www.soso.com	
www.msn.com	www.bing.com	

The results reported in this session were the average over the 20 measurements. The local Web page processing performance results (i.e., the first type of experiments) are reported in Sections 5.2 to 5.4, while the overall browsing performance results (i.e., the second type of experiments) are reported in Sections 5.5. In those reports, the data under "*Original*" are the results with the original version of Webkit without any modification. The data under "*First*" are the results of our proposed schemes when the Web pages were visited for the first time, i.e., the caches were empty before the first. The data under "*Subsequent*" are the results of our proposed schemes when the Web pages were visited previously, i.e., the caches were not empty.

5.2 Performance of Style Formatting

In Webkit, the main function for style formatting is CSSStyleSelector::styleForElement. We wrapped this function in measuring the time of style formatting. The performance of our SSC scheme against the original Webkit for local Web processing is shown in Table 2. The data in the table is an overall performance summed over the homepages of the TOP 25 Web sites listed in Table1 due to the space limitation. From the table, our SSC scheme improves the style formatting performance dramatically. On average, the performance is improved by 34% for the first visit and by 64% for subsequent visits. The style cache is empty when a Web page is visited for the first time, but our SSC scheme still improves the speed by 34% on average in computing style properties for DOM elements as compared to the original Webkit engine. This is because

- The style formatting for the elements with the same styles is automatically aggregated in the first visit to a Web page with our SSC scheme since they are merged in the SSC tree, which can be considered as a kind of optimization for style formatting.
- The in-memory cache, which is being built during the first visit to a Web page, has already been employed by our smart style caching scheme in processing subsequent data of the same Web page.

Our experiments show that the improvement is larger for large and complex Web pages. For example, the performance improvement is about 80% for subsequent visits to MySpace.com or AOL.com, larger than the average improvement of 64% for the average of the TOP 25 Web sites.

Table 2.Performance of style formatting

	Original	*First*		*Subsequent*	
Time(ms)	1269	835	34%	453	64.3%
Count	2814713	809548	71.2%	296121	89.5%

Our SSC scheme improves the style formatting performance because it eliminates the duplicated or unnecessary computations, which are mainly the matching operations between the DOM elements and the CSS selectors. The second row in Table 2 shows the numbers of corresponding matching operations. Compared with the original Webkit, our SSC scheme eliminates about 71% of matching operations for the first visit, and about 90% for subsequent visits.

5.3 Performance of Layout Calculation

Both style caching and layout caching can improve the performance of layout calculation, but in different ways. Layout caching is targeted to reduce the number of layout operations by reusing previously calculated layout results while style caching does not touch the logic of layout calculation directly. Style caching makes the style properties of each DOM element more stable and closer to the final style results, layout re-calculation would be triggered less frequently than the case without style cache. Webkit has carefully maintained the dirty-bits to indicate whether a layout operation is really needed or not. Less frequency in style changes leads to less layout re-calculation. In this section, we first report the performance of layout calculation affected by the style caching scheme, and then the performance of layout caching scheme. Again, the results were summed over the homepages of the TOP 25 Web sites due to space limitation.

5.3.1 Layout Performance with SSC Scheme

In order to evaluate the effects of the style caching scheme on layout calculation, we first report the performance results of layout calculation with only the style caching scheme enabled. The results are shown in Table 3 .

Table 3. Layout performance with SSC scheme

	Original	*First*		*Subsequent*	
Time(ms)	21895	21532	1.7%	20857	4.8%
Count	18844	16513	12.4%	14611	22.4%

We can see from the results that about 22% of layout operations were eliminated by the SSC scheme. The reduced time, however, is not very significant, only at a gain of 1.7% for the first visit and of 4.8 for subsequent visits. We have investigated the problem. The reason is that, in the original version of Webkit, about 80% of the time for layout operations is consumed to calculate the layout of an element for the first time, referred to as the first-time layout operation in this paper. Furthermore, the first-time layout operation for an element often consumes much more time than the subsequent layout operations for the same element. Since the style cache does not carry any layout data, it is obviously impossible to eliminate the first-time layout operations by our SSC scheme. Table 3 shows that 22.4% of subsequent layout operations were eliminated by our SSC scheme, the time reduction can be estimated to be 22.4% * 20% = 4.48, very close to the actual result of 4.8% time reduction shown in Table 3.

5.3.2 Layout Performance with Layout Caching Scheme

Table 4 shows the performance results for the layout caching scheme and the original Webkit. Since the layout caching scheme affects only subsequent visits to Web pages, Table 4 does not have first visit results. Table 4 shows that both the count and the time consumption of layout operations are significantly reduced. About 31% of the layout operations were eliminated by the layout caching scheme. Since the eliminated operations were mainly the first-time layout operations, the reduction of the time was about 56$, much larger than the 31% reduction in the layout operations. This is because the first-time layout operations need more time than subsequent layout operations

Table 4. Layout performance with layout caching scheme

Original		*Subsequent*			
Time	Count	Time	Improvement	Count	Improvement
21895ms	18844	9613ms	56.1%	12933	31.4%

5.3.3 *Layout Performance with Both Schemes*

Both style caching and layout caching improve the performance of layout calculation, as we have mentioned. Table 5 shows the layout calculation performance results when the both caching schemes were applied. The layout caching mainly improves the performance of the first-time layout operations and the style caching mainly improves the performance of the subsequent layout operations. Therefore, the overall layout performance is approximately the sum of the above two, confirmed by the results in Table 5. The layout calculation performance has improved by about 61%, and about 54% of the layout operations were eliminated for subsequent visits.

Table 5. Overall layout performance

	Original	*First*		*Subsequent*	
Time(ms)	21895	21672	1.0%	8503	61.1%
Count	18844	16513	12.4%	8687	54.0%

5.4 Performance of Page Processing

As shown in section 5.2 and 5.3, since our caching scheme reduces the time consumption of both style formatting and layout calculation significantly , the page processing time, which means the actual processor execution time during loading a page, could also be reduced notably. The results in Table 6 show that the caching schemes can significantly speed up the page processing of Web. With the cached data, on average, the performance of page processing can be improved about 46%.

Table 6.The overall page processing time (ms)

Original	*First*		*Subsequent*	
29977	29906	0.2%	16170	46.1%

5.5 Overall Performance

We also employed the page loading time to measure the overall browsing performance when networking, server, and local Web page processing were all taken into consideration. This would be close to a user's real world browsing experience. Almost all modern Web browsers support HTTP cache, but the simple Web browser we used in our experiments, i.e., GtkLauncher, didn't support HTTP cache. To mimic a real world Web browser, we used Squid, a Web caching proxy [14], was used for HTTP cache. Table 7 shows the performance results obtained with a mainstream desktop PC on a group Websites selected from the TOP 25 Web sites. Table 8 shows the corresponding results obtained with a typical netbook.

Both Table 7 and Table 8 indicate that our caching schemes could improve the performance of page loading for most of the web pages running on both desktop PC and on netbook. By comparing the data in both tables, we can find that browsing on the netbook took longer time to load than browsing the same Web page on the desktop PC. This gap should be due to the differences in the processing power of the two machines. The netbook's processor was much weaker than that of the desktop PC, therefore took more time to finish Web page processing. The two machines had the same networking environment during the experiments. That means the local Web page processing contribute more to the overall performance in the netbook as compared with the desktop PC. Therefore, our caching schemes should be more efficient for the netbook, since the essence of the cache schemes is to reduce the local computation. The results in Table 7 and Table 8 have confirmed this conclusion.

Table 7. Page loading time (ms) on Desktop PC

Sites	*Overall Page Loading Time(ms)*			
	Original	First	Subsequent	Subsequent Improvement
Baidu	978.83	992.6	863.61	11.77%
Google	1616.54	1581.68	1601.79	0.91%
Google.cn	1123.54	1056.48	1151.69	-2.51%
Soso	686.84	691.51	522.66	23.90%
ask	3616.88	3523.66	3640.06	-0.64
eBay	3258.54	3331.65	3310.19	-1.59%
Blogger	4304.16	4266.92	4124.46	4.18
MySpace	3332.88	3185.94	2218.34	33.44%
Msn	3764.09	3802.88	3650.72	3.01
Wikipedia	2294.64	2240.89	2191.71	4.49%
Sina	8122.57	8183.64	5769.79	28.97%
QQ	5318.75	5340.8	2339.21	56.02%
Xunlei	5448.7	5265.55	4533.04	16.81%
Yahoo	2376.88	2173.89	2220.96	6.56%
Youtube	3215.22	2939.68	2488.55	22.60%

Table 8. Page loading time (ms) on Netbook

Sites	*Overall Page Loading Time(ms)*			
	Original	First	Subsequent	Subsequent Improvement
Baidu	1738.99	1711.79	1372.85	21.05%
Google	2027.25	2038.15	1945.99	4.01%
Google.cn	1940.1	1967.24	1786.95	7.89%
Soso	1506.04	1523.97	990.55	34.23%
Ask	4659.01	4457.81	4398.3	5.60%
eBay	4119.29	4141.53	3959.2	3.89%
Blogger	5482.91	5178.19	5036.68	8.14%
MySpace	7773.25	7602.94	4865.85	37.40%
Msn	6781.88	6831.8	6088.81	10.22%
Wikipedia	3050.34	2968.14	2753.68	9.73%
Sina	20659.9	20893.52	14254.97	31.00%
QQ	15062.21	14716.96	5923.44	60.67%
xunlei	11993.54	11415.57	9724.33	18.92%
Yahoo	4460.77	4306.92	4108.89	7.89%
Youtube	5381.17	5421.33	4291	20.26%

Considering the complexity of the Web page listed Table 7 and Table 8, large and complex Web pages such as QQ.com and Sina.com, tend to can get more benefit from our caching schemes. This is reasonable since complex pages require more local computations, and our cache schemes were designed to reduce local computations. It is obvious that improving the page processing and loading performance for large pages is more meaningful than for simple pages like Google.com and Baidu.com. In fact, during the experiments, we perceived that the browser presented the complete appearance of the large pages more quickly if the cached data was available for them. This would definitely bring better user experiences for Web browsers.

We also measured the size of the cached data for each Web page. Usually, the cache size is tens to hundreds of kilobytes, depending mainly on the size of the Web page. Note that our implementation of caching schemes has not been optimized yet. Redundancy exists in the cache. In practice, such size of cached data should not be a concern of performance for today's computer storage configuration.

5.6 The Experiments on Dynamic Pages

The local Web page processing performance experiments reported in Sections 5.2~ 5.4 were conducted mainly with the same copies of Web pages (except Ajax and other dynamic part). This is about the best case for our caching schemes. Web pages in the real world may have much more dynamic and modified content, esp. when the gap in time between two subsequent visits is large. In this section, we study the local Web page processing performance on browsing real-world dynamic Web pages with the caching schemes. We selected two Web sites from the Top 25 sites, AOL.com and YouTube.com, which could represent two popular types of Web sites. The content of both sites changed frequently. These two sites were monitored for 12 hours, and their contents were fetched every hour during the mentoring duration. 12 copies of each Web site's content were collected. The first copy was used to generate a cache. Then, we used GtkLauncher to browse all the 12 copies without updating the cached data for any of the visits. In actual usage, the caches keep constantly updated. We did not update the caches to study the performance when cache miss increases. It is expected that the effectiveness of our caching schemes declines over time. In order to measure the effectiveness of the caching schemes, we counted the reductions of the CSS rule matching and layout operations for each page. The results are shown in Figure 5. From the figure, even though the Web page of AOL changed frequently, the cache effectiveness didn't decline quickly. Over the first 10 hours, the reduction of the matching operations declined only by 2%, from 96% to 94%, and the reduction of the layout operations declined from 34% to 30%. However, at the tenth hour, the validation checking for the SSC tree failed, thus all the cached data, including both the style cache and the layout cache, were invalidated. After the tenth hour, the layout cache was completely invalid, and the style cache performed as if it were the first visit. The results for YouTube are very different. Even though the cache was completely invalid at the 10^{th} hour, the same as AOL, the effectiveness had been significantly declined from the 3^{rd} hour. After the 3^{rd} hour, the cache was mostly invalid.

According to our experiments on other well-known Web sites, such as MySpace.com, EBay.com, Sina.com and so on, validation of the cache could last from 7 to 11 hours. For those static or rarely changed Web sites, such as Wikipedia.org and Google.com, the period of validation of the cache can extend to several days or even several weeks.

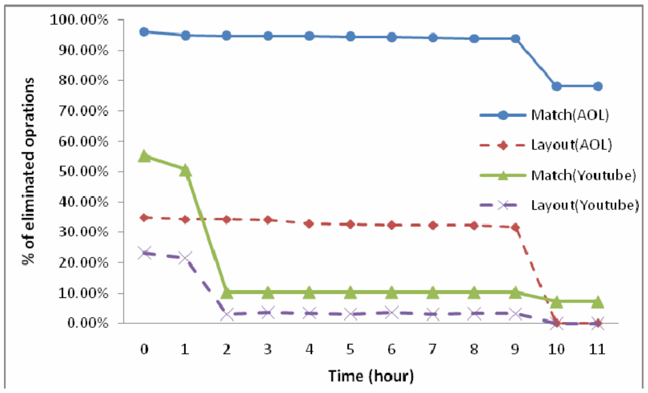


Figure 5. Reduction of the operations for YouTube and AOL

6. DISCUSSIONS

According to our experiences when we implemented the proposed two caching schemes and did experiments with the popular Web pages, in addition to their effectiveness for page loading, two caching schemes have additional advantages for Web browsers.

The first advantage is that even with an empty style cache, our algorithm of maintaining the style cache can choke the unnecessary style formatting and layout calculation. As shown in Section 5.2 and 5.3, when the Web pages are processed with empty caches, the amounts of style formatting and layout calculation can be reduced 71% and 12.4%, respectively. The reduced time consumption exceeds the overhead of our caching schemes, so we still have a little performance gain, as shown in Section 5.4, though our implementation of the caching schemes is still rough and not optimized. Probably the empty-cache setting of our style caching scheme can be thought as a kind of optimization to reduce the unnecessary temporary computations.

From the view point of user experiences, the user can see the final layout and visual results with a valid style cache more quickly than in the case of no cache. One reason is that the DOM elements get the final style properties as its initial values, thus they tend to be in the right styles at the very beginning. So users can see the right layout and style of the Web pages if the pages are not changed dramatically, leading to better user experiences.

Currently our caching schemes only work for the scenario of loading a Web page. In fact, they can be extended to support multiple versions of DOM elements in the style cache and layout cache, thus the cached data can be activated when the user interacts on the page and the JavaScript codes are triggered to respond to user's interactions. In this case, the extended caching schemes can potentially improve the responsiveness of Web applications. This shall be meaningful for complex Web applications like Google docs and Live map. This is our one future work.

At another direction, the cacheable and stable style properties and layout results for DOM elements can be extracted from a Web page so that they can be pre-processed and stored somewhere. When Web browsers load the page, they need not process these elements at all, but just load the cached results into memory and then render them directly. This kind of pre-processing can be transparent to Web browsers if the format of style and layout data is well defined. This shall be more meaningful for low-end systems like smart phones. It is another future work.

7. RELATED WORK

Improving the performance of Web page browsing has been paid more attention in industry than in academy. However, as multi-core systems are pervasive in the market and the requirement for Web browsers on handheld devices is emerging, parallelizing Web browsing is a possible solution for improving Web browsing. The Parallel Web Browser project in the Par Lab of UC-Berkeley attempts to parallelize different stages of Web page processing [3].

Venders of Web browsers have done lots of efforts to speed up the performance of Web page processing via various optimizations. Firefox contains many optimizations to reduce reflows [11], and Webkit [5] maintains a set of dirty-bits to avoid unnecessary internal re-computations. Internet Explorer 8 has done much to improve performance from various aspects, including memory management, JavaScript engine, networking, and rendering engine [4]. In addition to layout and render engines, JavaScript engine is another focus to improve Web browsers' performance. Chrome's V8 [15], Firefox's TraceMonkey [17] and Safari's Squalfish [18] employ Just-In-Time technology to execute JavaScript code.

The Opera Mini [19], a popular mobile Web browser, employs a server between the client and the Web site to improve user experience. Each Web page is compressed and pre-processed in the Opera's server before it is transferred to the mobile client, in order to speed up the networking and simplify the page processing. The solution reduces the network traffics significantly, but it cannot improve the internal performance of page processing.

On the other hand, some Internet companies have published guides to write more efficient Web pages [16]. This represents another kind of efforts to reduce the local computations in Web page browsing. These guides do not touch the internal logics of Web browsers, but take advantages of the internal process logics in Web browsers to avoid the heavy computations.

8. CONCLUSIONS

In this paper, we proposed two caching schemes to improve the performance of Web page processing. We focus on the two important stages in the workflow of page processing: style formatting and layout calculation. With our smart style caching scheme and layout caching scheme, the style data and layout data for DOM elements are recorded when a page is browsed, and then reused when the page is revisited later. The validation checking is done in the granularity of DOM elements. Therefore, even for dynamically changed Web pages, the cached data is still partially valid so as to reduce the local computations for the unchanged elements. The experimental results show that the two caching schemes can significantly improve the performance of Web page browsing. With both style caching and layout caching schemes enabled, the performance of sequent visits to the same pages can be improved by about 46% on average.

9. ACKNOWLEDGMENTS

We would like to thank Zhenbin Xu of Microsoft for valuable discussions and detailed information on Web browsers, particularly Internet Explorer.

10. REFERENCES

[1] Microsoft Corp. Bing Maps. http://www.bing.com/maps.

[2] Google Inc. Google Docs. http://docs.google.com/.

[3] Jones, C. G., Liu, R., Meyerovich, L., Asanović, K., and Bodik, R. 2009. Parallelizing the Web browser, 1st USENIX Workshop on Hot Topics in Parallelism (Mar. 30-31, 2009).

[4] IE 8 Performance. http://blogs.msdn.com/ie/archive/2008/08/26/ie8-performance.aspx.

[5] The Webkit Open Source Project. http://webkit.org/.

[6] Cascading Style Sheets 2.1. http://www.w3.org/TR/CSS2/.

[7] W3C. Document Object Model (DOM). http://www.w3.org/DOM/

[8] Reflows & Repaints: CSS performance making your JavaScript slow. http://www.stubbornella.org/content/2009/03/27/reflows-repaints-css-performance-making-your-javascript-slow/.

[9] Wilton-Jones, M. Efficient JavaScript. http://dev.opera.com/articles/view/efficient-javascript/.

[10] Baron, D. Faster HTML and CSS: layout engine internals for web developers, https://library.mozilla.org/Faster_HTML_and_CSS:_Layout_Engine_Internals_for_Web_Developers.

[11] HTML Reflow, http://www.mozilla.org/newlayout/doc/reflow.html.

[12] W3C standards. http://www.w3.org/standards/.

[13] GNU WGet, http://www.gnu.org/software/wget/

[14] Squid Web Caching Proxy, http://www.squid-cache.org/

[15] V8 JavaScript Engine, http://code.google.com/apis/v8/design.html

[16] Best Practices for Speeding Up Your Web Site, http://developer.yahoo.com/performance/rules.html

[17] JavaScript: TraceMonkey, https://wiki.mozilla.org/JavaScript:TraceMonkey

[18] SquirrelFish-Webkit, http://trac.webkit.org/wiki/SquirrelFish

[19] Opera Mini homepage, http://www.opera.com/mini/