Slicing*-Tree Based Web Page Transformation for Small Displays

Xiangye Xiao, Qiong Luo, Dan Hong, Hongbo Fu
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{xiaoxy, luo, csdhong, fuhb}@cs.ust.hk

ABSTRACT

We propose a new Web page transformation method for browsing on mobile devices with small displays. In our approach, an original web page that does not fit into the screen is transformed into a set of pages, each of which fits into the screen. This transformation is done through slicing the original page iteratively with several factors considered, including the size of the screen, the size of each page block, the number of blocks in each transformed page, as well as the semantic coherence between blocks. The resulting set of transformed pages form a multi-level tree structure, called a slicing*-tree, in which an internal node consists of a thumbnail image with hyperlinks and a leaf node is a block from the original web page. Through this transformation, the contextual information in the original web page is preserved and the page scrolling effort is saved. We have implemented this transformation module on a proxy server and have conducted empirical studies on its performance.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications—*Information Browsers*; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Navigation*; I.7.5 [**Document and Text Processing**]: Document Capture —*Document analysis*

General Terms

Design, Algorithms, Human Factors

Keywords

Web browsing, small displays, proxy, thumbnails, Web page adaptation, slicing tree, VIPS algorithm

1. INTRODUCTION

Internet-enabled Personal Digital Assistants (PDAs) have become more and more powerful and compact. For instance, the new HP iPAQ hx4700 Pocket PC features a 624 MHz processor, 128 MB ROM, 64 MB RAM, and 4' display, all contained in a $3' \times 5' \times 0.6'$ thin pad. As these devices gain increasing popularity, it is desirable to make Web browsing, one of the most common activities on desktops, also convenient on these devices. However, the majority of current Web sites are designed for desktop displays (e.g., the page in Figure 1 (a)), and only a handful of browsers on PDAs (e.g., PalmScape [2], ProxiNet [3], and HandWeb [4]) support limited Web page adaptation for small displays. As a result, PDA users have to scroll constantly when viewing a Web page on a palm-sized screen (e.g., the page in the PDA in Figure 1 (a)). In this paper, we explore a new approach to automatic page transformation for small displays.

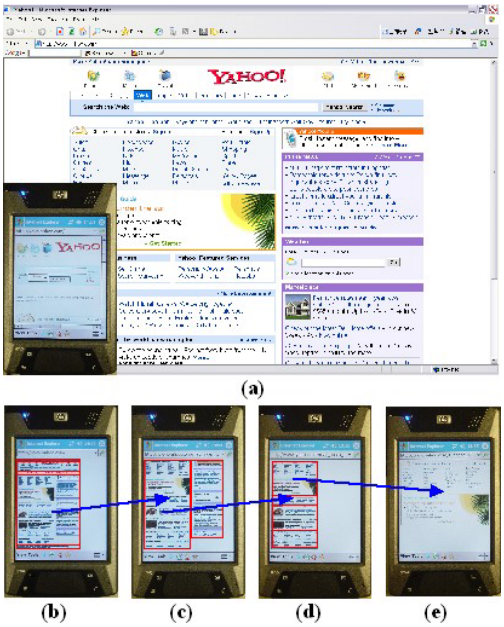

Figure 1: (a) Displaying an original Web page on a desktop computer and a PDA. (b)-(e) Transformed pages of the original page.

Our approach is based on the following key observation: PDA users draw heavily on their browsing experience on desktops when browsing on PDAs. In particular, page layout and visual context information are crucial for users to identify the content and links of their interests in a page. Therefore, for an original web page that does not fit into a small screen, our transformation method first displays its thumbnail image with multiple embedded hyperlinks (Fig-

ure 1 (b)). When a part of the page is pen-tapped, its corresponding screen-fitting sub-page is then displayed, and this tap-and-display process may continue a few more times until the target of interest is found (Figures 1 (c)-(e)). During the entire browsing process, the original page layout and context information is preserved and scrolling is seldom needed.

There are several interesting research questions to be answered in our approach. Especially, how do we divide a large page into several smaller ones so that each of the sub-pages fits into the screen? What parts of a page are "good" to be put into one sub-page? What is the right number of sub-pages in one page, since too many sub-pages in one page is hard for the users to browse, and too few may result in too many pen-taps during navigation? Moreover, how is the user browsing experience with the transformed pages in comparison with the original pages, as scrolling is nearly eliminated but pen-taps increase?

To divide a large page into smaller ones, we adopt a variation of the binary slicing tree [16], which we call a slicing*-tree. The only difference between a binary slicing tree and our slicing*-tree is that the fanout (or degree) of our tree is no less than two and no more than a threshold value, which we typically set to 4-12. We use the slicing*-tree to represent the organization of the set of transformed pages for an original web page. Each internal node in the tree is a thumbnail of the original Web page (the root) or that of an intermediate sub-page, and each leaf node is a leaf page (a sufficiently small block in the original web page). With this organization, the node degree requirement keeps the number of sub-pages in a page "about right". Additionally, we use the VIPS module developed by Cai et al. [12] to generate leaf pages, each of which has a high degree of coherence.

We have implemented the slicing*-tree based transformation module on a proxy server and have conducted initial experiments using an HP iPAQ hx4700 PDA. We asked users to perform focused search tasks and other tasks on PDAs with and without the page transformation proxy, and compared the task completion time and input effort. We also examined the bandwidth consumption, the time breakdown, as well as the performance impact of the tree fanout threshold. Our results indicate that this transformation method eases user browsing experience and saves communication cost.

The remainder of this paper is organized as follows. We discuss related work in Section 2 and overview our system in Section 3. We present the slicing*-tree based transformation algorithm in detail in Section 4 and experimental results in Section 5. Finally, we conclude in Section 6.

2. RELATED WORK

Schilit et al. [19] classify the techniques of fitting desktop content into a small display into four categories: *scaling*, *manual authoring*, *transducing* and *transforming*. *Scaling* can reduce scrolling, but it reduces Web page readability as well. *Manual authoring* is laborious in that it requires professional Web designers to manually tailor Web pages to fit into particular devices. In contrast, automatic Web page reauthoring methods, *transducing* and *transforming*, release Web designers from the heavy manual workload. *Transducing*, such as AvantGo [1], translates HTML and images into other formats, and compresses and converts images to match device characteristics. *Transforming* goes further to modify both contents and structures of Web pages originally designed for desktop browsing to make them suitable to display on small screens.

The existing *transforming* methods can be further categorized into three classes according to their visualization techniques: *single column*, *fisheye*, and *overview + detail*. MS Internet Explorer and Opera SSR are example systems that provide a *single column* view. While this *single column* view eliminates scrolling in one dimension, it greatly increases the amount of scrolling in the other dimension. In comparison, Fishnet [5] is a *fisheye* Web browser that shows a focus region at a readable scale while spatially compressing page content outside the focus region, as its name suggests. In this fisheye view, users need to scroll to move their desired information into the focus region. Fishnet is targeted for desktop browsing, and therefore, does not attempt to address horizontal scrolling. Finally, the *overview + detail* method splits a Web page into multiple sections and provides an overview page with links to these sections. The overview page can be either a thumbnail image as in our work, or a text summary of the Web page.

Both the Stanford Power Browser [8, 9, 10, 11] and the Document Segmentation and Presentation System (DSPS) [17] provide text summaries in the overview page and reveal the detailed Web page content progressively. One advantage of using text summaries is its simplicity in the display, but the downside is that the visual context (e.g., styles, images) in the original Web page is lost. Consequently, users' browsing experience on desktop PCs cannot be utilized on PDAs to help discover the target information.

In contrast to text summaries, thumbnail overviews [6, 7, 14, 15, 18, 20] preserve the visual context information of the original web pages. However, previous work is limited to a two-level hierarchy with the thumbnail overview page linked to a set of sub-pages for detailed information. When an original Web page is large, the thumbnail overview is either too large to fit into the small screen or too crowded for users to identify the sections of interest. As a result, scrolls and pen-taps may not be effectively reduced for large pages.

Our work falls into the thumbnail overview category, but we generalize this method to allow transformed pages to form a multi-level hierarchy with each transformed page fitting into the screen and consisting of a small number of visual blocks. This generalization gives users sufficient but not overwhelming visual context information as well as reduces user input effort. We use the VIPS algorithm [12] to segment an original Web page into visual blocks and propose a variation of the slicing tree to build the hierarchy of transformed pages. The text summary overview method is complementary to ours in that we can embed a text summary for each visual block.

Our slicing*-tree based transformation achieves a balance between two extremes in the previous work. One extreme, the binary slicing tree organization [13], generates a deep hierarchy of transformed pages with each level consisting of

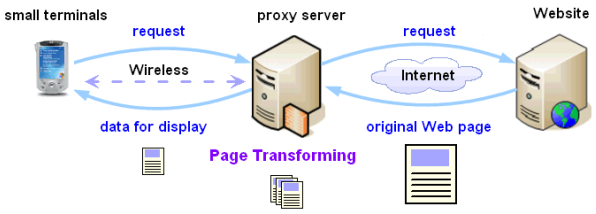

Figure 2: Page transformation proxy server architecture.

two sub-pages only. With this organization, users need to pen-tap many times to navigate down to the target sub-pages. The other extreme uses a two-level *overview + detail* hierarchy [6, 7, 14, 15, 18, 20], with the top level page consisting of the hyperlinks to all of the bottom level sub-pages. Its downside is that there may be too many sections in the top level page so that it is hard for users to choose from.

3. SYSTEM OVERVIEW

We have implemented our page transformation system on a proxy server. In this section, we describe the system architecture and the major components.

Figure 2 illustrates the architecture of our page transformation proxy server. We chose to implement the page transformation module on a proxy server rather than on a specific PDA browser for two reasons. One is that a proxy server is commonly used in local area networks as firewalls or accelerators and is more powerful than PDAs. Therefore, page transformation can be done more efficiently on a proxy server than on a PDA and can be shared by multiple PDAs. The other reason is that PDA users usually have their browsers of choice and may be reluctant to switch to a new kind of browser or to accept browser extensions. Consequently, our system resides on a proxy server and transparently serves transformed pages to PDAs that go through this proxy.

As shown in Figure 2, when a PDA sends a request to a Web site through our proxy, the proxy forwards the request and transforms the received Web page into a tree hierarchy of thumbnail index pages as internal nodes and leaf sub-pages as leaf nodes. It returns the root index page to the PDA for display and stores the other sub-pages locally. When the user selects a region in the index page, it then serves the user the corresponding sub-page that has been stored.

The page transformation system at the proxy consists of four modules: page splitting, and the generation of thumbnail images, index pages, and leaf pages (Figure 3). The page splitting module builds the tree representation of the set of transformed pages, given the original web page, the screen size, and other information. We will defer the detailed presentation on page splitting to Section 4 and discuss only the other three modules in this section. All of these three modules take the tree representation, called a slicing*-tree, as an input.

The thumbnail generation procedure first captures a thumbnail image of the original Web page (corresponding to the

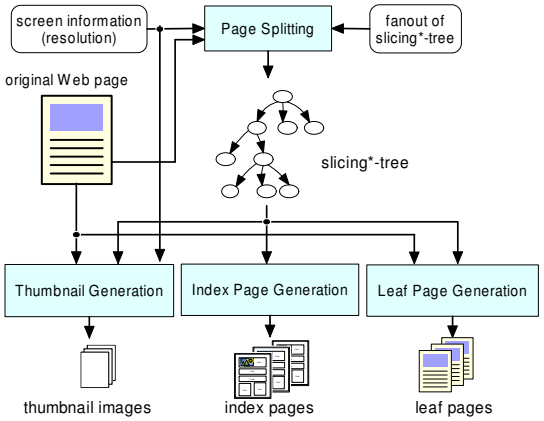

Figure 3: Page transformation process.

root node of the slicing*-tree). It then traverses down the slicing*-tree to examine each node in the tree. Each node has location attributes (*ObjectRectTop*, *ObjectRectLeft*, *ObjectRectWidth*, and *ObjectRectHeight*) to record the position and size of its corresponding block or section in the original Web page. The procedure uses this location information to border the corresponding blocks of the child nodes in the thumbnail image of the parent node with color. After that, it performs image clipping to cut out the corresponding thumbnail image for each internal node. For a leaf node, no thumbnail image is generated. After all thumbnail images are generated and blocks in them are bordered, these images are resized to fit into the small screen.

We say that each internal node of the slicing*-tree corresponds to a thumbnail image, but more accurately, it corresponds to an index page, or a sub-page, with the thumbnail image embedded. This is because in addition to the thumbnail image, the page needs to embed hyperlinks to its sub-pages at the corresponding sections of the image. We generate a hyperlink for a section based on the location attributes of the slicing*-tree node that corresponds to the section.

For each leaf node of the slicing*-tree, we generate a corresponding leaf page. The leaf page generation procedure extracts the source HTML code corresponding to the leaf node from the original HTML document. It also copies the header of the original page into the leaf page in order to keep the original appearance of this section. Finally, it modifies the hyperlinks in the page that point to other parts of the original Web page so that the updated hyperlinks point to the transformed pages correctly.

4. SLICING*-TREE FOR A WEB PAGE

After giving an overview to our page transformation system, we focus on presenting the page splitting module, in particular, the slicing*-tree construction, in this section.

4.1 Slicing*-Tree Representation

A slicing floorplan is a decomposition of a rectangle with horizontal and vertical cuts (as shown in Figure 4 (a)). A slicing floorplan can be represented as a binary tree, called a binary *slicing tree* (e.g., Figure 4 (b)). An internal node

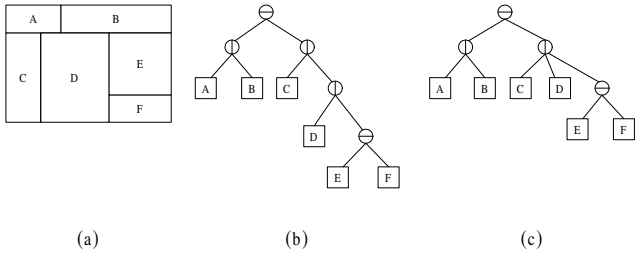

Figure 4: (a) A slicing floorplan. (b) The binary slicing tree of the floorplan. (c) The slicing*-tree of the floorplan with $T = 3$.

in the slicing tree represents a cut, either in the horizontal dimension or in the vertical dimension (labeled 'h' or 'v' correspondingly). A leaf node in the slicing tree represents an atomic rectangle that has no cut through.

In comparison with the binary slicing tree, our slicing*-tree does not require each internal node to have exactly two children (Figure 4 (c)). Instead, it requires each internal node has at least two children and has no more than a specific number of children, which we denote as the threshold T. We add this threshold based on the observation that it is easy for users to identify in a region the sub-region of interest as long as there are only a few, not necessarily two, sub-regions present.

Formally, a *slicing*-tree* is a slicing tree that satisfies the following property: for any internal node of the tree, it has at least two children and at most T children, where $T \geq 2$.

As a Web page typically consists of box-shaped sections, we can naturally use a slicing*-tree to represent a Web page decomposition scheme. Nevertheless, there are various ways of transforming a Web page for the small screen given its slicing*-tree.

First, we decide on how many levels of transformed pages will be generated out of the original page. This level may be different from the height of the slicing*-tree. Previous work used a two-level hierarchy, with the top level page consisting of some hyperlinks to the sub-pages and some sections from the original page. For instance, Dress [13] studied the problem of choosing which sections from the original page to be put in the top level page, even though the slicing tree they used was a binary tree of many levels. Another extreme would be to generate a deep hierarchy of transformed pages, with each level consists of two sub-pages only. The danger of a two-level hierarchy is that there may be too many sections in the top level page so that it is hard for users to choose from. In comparison, the downside of a deep hierarchy of transformed pages is that users need to pen-tap many times to navigate down to the sub-page that they are interested in. Therefore, we take the middle of the road and adhere to the slicing*-tree hierarchy for transformation: each internal node is transformed into a sub-page and each leaf node is a section from the original page. Since the degree of the slicing*-tree is bounded, the depth of the hierarchy is usually shallow, but not necessarily two, in practice.

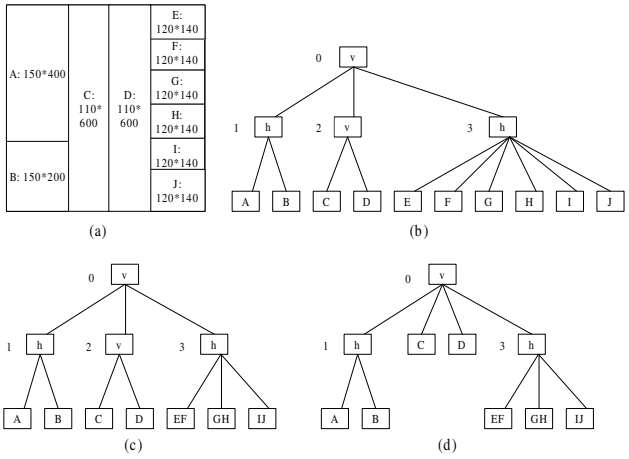

Figure 5: (a) Floorplan of a Web page. (b) VIPS tree of the Web page. (c) Tree structure after leaf extraction. (d) Final slicing*-tree after internal node adaptation. ($(s_w, s_h) = (300, 300), T = 4$).

Second, for an internal node, we decide on if we use a text summary or a thumbnail image to represent it in the transformed sub-page. We decide to use thumbnail images for the reason of preserving the visual context information of the original Web page. In order to save scrolling effort, we require the size of a thumbnail image to fit into the small screen.

In summary, a slicing*-tree represents the transformation of a Web page as follows:

1. Each leaf node represents a leaf page transformed directly from a section in the original page. The leaf page fits into the screen.

2. Each internal node represents a thumbnail image page of a large section of the original page. This thumbnail image fits into the screen. Furthermore, for each child node, there is an embedded hyperlink pointing to its corresponding sub-section in the thumbnail image of the parent node.

4.2 Slicing*-Tree Construction

Having described the slicing*-tree definition and its representation for page transformation, we proceed to present its construction. The construction goes through three steps: VIPS tree construction, leaf extraction, and internal node adaptation. We describe these three steps in order.

4.2.1 VIPS Tree Construction

As the slicing*-tree nodes represent sections in a Web page, we first need to identify sections or blocks in the page. One way is to use the HTML Document Object Model (DOM) tree, and another, which we use, is the VIsion-based Page Segmentation (VIPS) tree [12]. The VIPS algorithm extracts the semantic tree structure of a web page based on its visual presentation. Each node corresponds to a box-shaped section (i.e., a block) and is assigned a value, Degree

of Coherence (DoC), to measure the coherence of the content in the block based on its visual perception. The larger the DoC value is, the more semantically coherent the section is.

As we see in Figure 5 (b), the VIPS tree of a Web page has most of the properties of a slicing*-tree except that (1) a leaf page may be too small or too large with respect to the screen size, and that (2) the number of children of an internal node may be larger than the threshold. Thus, the problem of slicing*-tree construction becomes to first generate the VIPS tree and then to transform it into the slicing*-tree by addressing these two differences.

To be conservative about the leaf page size, we generate the VIPS tree so that its leaf pages are as small as possible. This is achieved by setting the Permitted Degree of Coherence (PDoC) parameter of the VIPS algorithm to its largest value, 10. With a large PDoC value, the VIPS algorithm segments a Web page at a fine granularity.

4.2.2 Leaf Extraction

After the VIPS tree with small leaf pages is constructed, we go through the leaf extraction step to transform the tree into another one. The goal is to make each leaf page as large as possible within the screen size limit. This is done by merging small-sized neighboring sibling nodes in the tree. Since the following step, internal node adaptation, does not remove or insert any leaves, the leaves in the result tree of this step is exactly the leaves in the final slicing*-tree. Therefore, we name this step leaf extraction. Figure 5 (c) shows a result tree of leaf extraction.

In this step, we traverse the VIPS tree through Depth-First Search (DFS). For each internal node, if its size is smaller than the screen size, we remove all of its children because there is no need to further decompose it. Otherwise, we examine its children and see if these children can be re-partitioned to decrease the number of children and to increase the size of each new child node within the screen size limit. Algorithm 1 shows the leaf extraction process.

Algorithm 1 LeafNodeExtraction

Input:
$Tree$ is a VIPS tree or a sub-tree;
s_w is the screen width;
s_h is the screen height;
Output:
$Tree$ is the modified tree;

```
1: begin
2: root = Tree.root;
3: if root is an internal node then
4:    Get the rectangle width r_w and height r_h of root;
5:    if r_w > s_w or r_h > s_h then
6:       PartitionSiblingNodes(root.childnodes());
7:       for all item i ∈ root.childnodes() do
8:          LeafNodeExtraction(i, s_w, s_h);
9:    else
10:      remove all the children of root;
11: end
```

The subroutine *PartitionSiblingNodes* attempts to partition an array of sibling nodes into sub-arrays with the total size of the nodes in each sub-array fitting into the screen. If the number of nodes in a sub-array is more than one, these nodes will be merged into a new child node. Note that the order of the sibling nodes in the array is unchanged throughout partitioning and that only adjacent sibling nodes can be merged so as to preserve the original appearance of the Web page in terms of the spatial relationship between sections.

The problem of partitioning sibling nodes can be abstracted into the following positive number array partitioning problem:

DEFINITION 1. *Positive number array partitioning: given an array of positive real numbers* $(e_1, e_2, e_3, \ldots, e_n)$ *and a positive real number* B *as the bound, partition the array into a number of sub-arrays* $(\ (e_1, \ldots, e_i), (e_{i+1}, \ldots, e_j), \ldots, (e_m, \ldots, e_n)\ )$ *such that the sum of the elements in each sub-array does not exceed* B.

If the layout of the sibling nodes to be partitioned is vertical, the bound B corresponds to s_h and the array of numbers the heights of the nodes. If the layout is horizontal, the bound B corresponds to s_w and the array of numbers the widths of the nodes. Our goal is to merge the nodes in each sub-array so that the number of resulting nodes is the minimum. This goal translates into finding an optimal partitioning scheme for the array partitioning problem, in which the number of sub-arrays is the minimum among all possible partitioning schemes.

We developed a simple greedy algorithm (Algorithm 2) to find the optimal partitioning scheme for the array. It scans the array and keeps a running sum of the elements scanned so far. Whenever the sum becomes larger than B, it outputs the index of the current element. It then resets the sum to the current element value and continues to scan and sum. When the algorithm finishes scanning the array, it produces a partitioning scheme in the form of the indexes of the elements that serve as the boundaries between the sub-arrays.

Algorithm 2 GreedyPartition

Input:
array is an array of positive real numbers;
B is a positive real number as the bound;
Output:
division is an array of indexes that serve as the boundaries between the sub-arrays;

```
1: begin
2: i = 0;
3: while i < array.length() do
4:    sum = array[i];
5:    length = 0;
6:    while sum < B and i < array.length() do
7:       length++;
8:       i++;
9:       sum+ = array[i];
10:   if length == 0 then
11:      i++;
12:   division.add(i);
13: end
```

We prove that our greedy algorithm is optimal. Let G be the partitioning scheme produced by the greedy algorithm and O be an optimal partitioning scheme. We scan the arrays in G and O and compare them. Suppose the first pair of different sub-arrays in G and O are SA_G and SA_O, respectively. They start at the same element but end at different

elements. Since SA_G is produced by the greedy algorithm, its length must be larger than that of SA_O. We extend SA_O to be the same as SA_G. As a result, the length of the next sub-array in O is reduced but the resulting new O has the same number of partitions as the old O (the old O is not optimal if the new O has a smaller number of partitions). When this process of array scanning and sub-array extending finishes, the final O is the same as G. Therefore, G is an optimal partitioning scheme.

4.2.3 Internal Node Adaptation

The leaf node extraction step examines the node size with respect to the screen size, and the following step, internal node adaptation, adjusts the tree to satisfy the node degree requirement as well as to reduce the height of the tree as much as possible.

Internal node adaptation (Algorithm 3) is done through a DFS traversal on the tree. If a node has more than T children, the algorithm combines some children by adding more levels of internal nodes between the node and these children so that it has exactly T children. If a node has fewer than T children, the algorithm attempts to increase its degree up to T. Note that node degree decrease to T is required for a slicing*-tree, but node degree increase to T is not. The purpose of increasing node degree within the limit is to reduce the height of the tree.

Algorithm 3 InternalNodeAdaptation

Input:
$Tree$ is a tree constructed by leaf node extraction;
T is the fanout of the tree;
Output:
$Tree$ is the modified tree;
1: **begin**
2: $root = Tree.root$;
3: **if** number of children of $root > T$ **then**
4: CombineChildren($root$);
5: $pushup = TRUE$;
6: **while** 0 < number of children of $root < T$ **and** $pushup == TRUE$ **do**
7: $pushup$ = PushUpGrandChildNodes($root$);
8: **for all** item $i \in root.childnodes()$ **do**
9: InternalNodeAdaptation(i, T);
10: **end**

The node degree decrease adjustment is done through the CombineChildren($root$) procedure in Algorithm 3. First, it removes all children of $root$. Then, it partitions the array of these children to T disjoint sub-arrays with each sub-array having the same number of nodes. At this point, it begins to add T children to $root$: if a sub-array consists of more than one node, construct a parent node for them and add the new parent node to be a child node of $root$; if a sub-array contains a single node, directly append it as the child node of $root$. Note that this CombineChildren($root$) procedure is different from PartitionSiblingNodes in Algorithm 1 in that the children nodes here are not partitioned based on their sizes or merged into a new node.

The node degree increase adjustment is attempted in the PushUpGrandChildNodes($root$) procedure when a node $root$ has fewer children than T. The principle of pushing up grandchild nodes is All or Nothing: either remove the child node c of $root$ and push up all children of c to be the children of $root$, or keep c at its place and do not push up any children of c. This principle is to preserve the semantic coherence between sections in the original Web page.

Specifically, we define a child node c to be *removable* if (1) c is an internal node that has the the same label ('v' or 'h') as its parent node p, and (2) the sum of the degree of c and the degree of p minus 1 is no larger than T. If a node has multiple *removable* child node candidates, the priority of removing them is determined first by the number of descendant leaf nodes and then by the size of the node. The intuition is to reduce the height of the tree, especially the depth of the leaf nodes and that of large sections, so that the number of pen-taps can be reduced when users access the transformed pages.

After sorting the removable child node candidates of $root$ by the descending priority, the PushUpGrandChildNodes($root$) procedure removes them one by one and pushes up their children until the sum of current degree of $root$ and the degree of the next candidate c to remove exceeds $T+1$. The procedure returns $TRUE$ if it has pushed up one or more grandchildren of $root$. Otherwise, it returns $FALSE$.

In Figure 5 (c), node 2 has the same label as its parent node 0 and the sum of the degrees of node 2 and node 0 minus 1 is not larger than T, 4. Therefore, node 2 is a removable node. The resulting slicing*-tree after internal node adaptation is shown in Figure 5 (d), where node 2 is removed and its children C and D are pushed up.

5. EXPERIMENTAL EVALUATION

In this section, we report our experimental results about our slicing*-tree based page transformation system.

5.1 Experimental Setup

As the major goal of our system is to improve the browsing experience of PDA users, we recruited ten graduate students from the Computer Science department to be our participants. These participants are experienced Web users. They are familiar with Web browsing on desktop computers and have a history of accessing the Internet using small devices. They had not used our system before the experiments. We divided the ten participants into two groups, with one group of five browsing on PDAs connected through our transformation proxy and the other group of five directly accessing the Internet.

The devices all of the participants used were HP iPAQ hx4700 Pocket PC with a 624 MHz processor, 128 MB ROM, 64 MB RAM, and 4.0' Transflective VGA (480 × 640 pixels) TFT display, running Internet Explorer on Windows Mobile 2003. The transformation proxy server was on a PC with an Intel P4 3.20 GHz processor and 1.00 GB memory. The PDAs had a wireless connection and the proxy was on a wireline local area network.

We designed ten browsing tasks, including five focused search tasks and five reading tasks, for each of the ten participants to perform. These tasks and the Web pages involved are listed in Table 1. Tasks 1 to 5 are focused search tasks and tasks 6 to 10 are reading tasks. The focused search tasks are similar to those conducted for the Stanford Power Browser

Task	Involved Web page	Task description
1	ebay.com	Find the link on "How to register".
2	travelocity.com	Find the section "Last Minute Deals".
3	infoSpace.com	Find the option box "Find a business by Name".
4	iwon.com	Find the name of today's $100,000 winner.
5	hkex.com.hk	Find today's Hang Seng Index Movement Chart.
6	CFP of CIKM2005 conference Website	Read this Call for Paper page.
7	cnn.com	Read the news "China to all visits to Taiwan".
8	cnn.com	Read the news "Pentagon vows to probe Saddam photos".
9	ebay.com	Read the product description in the first page of list of soaps.
10	ust.hk	Read ITSC Survey on HKUST Notebook/Desktop Ownership Program 2005.

Table 1: 10 tasks for Participants to Complete on PDA.

1	yahoo.com	2	altavista.com	3	yesasia.com
4	ebay.com	5	iwin.com	6	travelocity.com
7	americangreetings.com	8	bizrate.com	9	earthlink.net
10	google.com	11	cnn.com	12	hotmail.com
13	msn.com	14	askjeeves.com	15	lycos.com

Table 2: 15 tested popular Web pages.

[8, 9, 10, 11]. Focused search tasks require the users to look for an object, like a button or a piece of information, on a given Web page. In focused search tasks, most of the user interaction time is spent on searching. The reading tasks aimed at testing the scenarios in which the visual context in the original Web page has little impact, because the users are given the URL directly and know the location of their interests at the beginning. In the reading tasks, most of the user interaction time is spent on content viewing. Additionally, we selected 15 index pages from popular Web sites [14] to further study the characteristics of the transformed pages in practice. These Web sites are listed in Table 2. Our experiment is comparable with previous work [10, 11, 14] considering the number of participants, the number and the design of tasks, and the source of the original Web page.

In our experiments, we asked the participants to each perform the ten tasks and compared the average performance of browsing with and without our page transformation proxy. We use task completion time, input effort, and bandwidth consumption as the performance metrics for comparison. The task completion time is measured after the initial URL is entered until the user completes his task. The input effort is measured in terms of number of pen moves, including taps on the display and pulls up and down on the scroll button. The bandwidth consumption for a task is the number of bytes received at the client device during the task completion time.

To study the performance of our system in more detail, we further divide the task completion time into three parts: user interaction time on the client device, processing time at the proxy, and data transmission time on the network. The user interaction time is the time the user spends on pen actions and content viewing. In addition, we tested the effect of the parameter T setting, as it is the threshold for the degree of a slicing*-tree.

5.2 Performance of Page Transformation

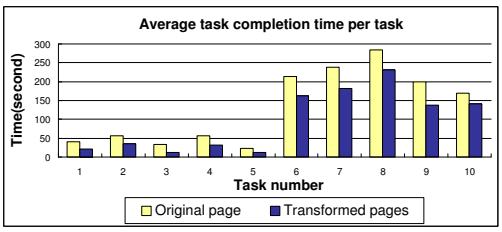

Figure 6: Average task completion time, $T = 8$.

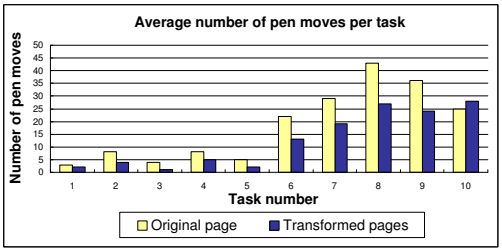

Figure 7: Average number of pen moves, $T = 8$.

Average task completion time: Figure 6 shows the average task completion time for browsing methods with and without our proxy. Our system achieved a shorter task completion time in all of the ten tasks, with 47% improvement on the focused search tasks and 22.7% improvement on the reading tasks. Our method is especially helpful in reducing the task completion time when the target in the task is an image, a chart, a controller-like textbox or other noticeable visual items, or when the users are familiar with the original Web pages. The task completion time was reduced because the time-consuming vertical and horizontal scrolling was almost completely eliminated on the transformed pages. Also, users could find their interests easier with the clear visual context.

Input effort: Figure 7 illustrates the average number of pen moves performed for each task. In nine out of the ten tasks, our system achieved a smaller number of pen moves. The improvement was 50% for the focused search tasks and 28.4% for the reading tasks. When the transformed Web pages are browsed, the number of pulls on the scroll bar is zero and the number of taps possibly increases because the users need to switch between pages by selecting links and pressing the back button. Since a pen tapping is instantaneous and is easier than holding the pen to pull on the scroll bar, this input effort reduction in our system is potentially unquanlified by the number of pen moves.

Bandwidth consumption: Figure 8 shows the bandwidth consumption of each task. When the original Web page is browsed, the amount of data received at the client device is the size of the original Web page including the HTML page, its embedded images and style sheets. When transformed pages are browsed, it is the size of transformed pages chosen by the user including some index HTML pages, leaf HTML pages, and embedded thumbnail images. Our system saved bandwidth consumption in seven out of the ten tasks, and the average saving was 18.9%. When the original Web page

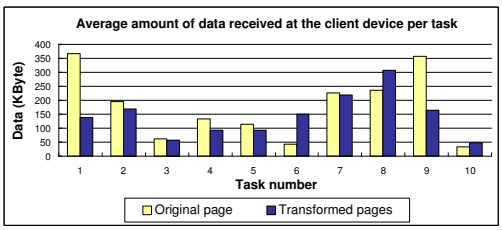

Figure 8: Bandwidth consumption, $T = 8$.

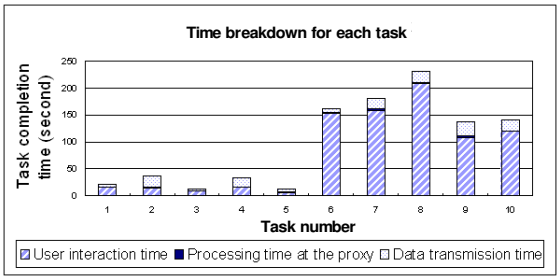

Figure 9: Time breakdown, $T = 8$.

consisted of mainly text and few images, the bandwidth saving was small or negative (e.g., in task 6).

Page transformation time: Figure 9 shows the breakdown of the task completion time. The processing time spent on page transformation at the proxy was only a small portion (less than 10%) of the overall task completion time. To further study the transformation processing time, we examine it for the fifteen selected Web pages with different T values in Figure 10. In this figure, we see that the processing time on page transformation drops as T increases. This is because the height of the slicing*-tree decreases when the maximum fanout increases. As a result, the time spent on generating thumbnail images is reduced. The improvement slows down as T becomes even larger, because it does not reduce the tree hight much any more. Also, the layout of the original Web page affects the transformation time. For instance, a large and complex web page such as cnn.com (Bar 11) takes the longest time and a simple one such as google.com the shortest (Bar 10). The processing time of google.com remains the same as T increases, because the layout of google.com is so simple that its slicing*-tree does not change with different T values.

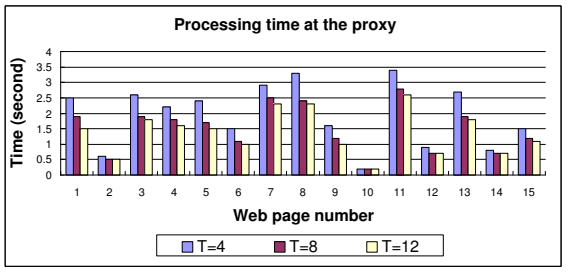

Figure 10: Processing time at the proxy.

Web page	Height of slicing*-tree			Size of original page (KByte)	Total size of transformed pages, T=8 (KByte)
	T=4	T=6	T=8		
1	3	2	2	95	336
2	3	2	2	22	71
3	4	3	3	138	391
4	6	5	4	386	340
5	5	4	3	195	472
6	3	2	2	196	348
7	4	3	2	146	374
8	5	4	4	149	216
9	4	3	2	168	77
10	2	2	2	16	237
11	5	4	4	238	770
12	2	2	2	31	80
13	4	3	3	133	448
14	4	3	2	43	75
15	3	2	2	118	362

Table 3: Characteristics of each tested Web page.

To study the characteristics of transformed pages in practice, in Table 3, we list the heights of the slicing*-trees of the fifteen Web pages with various T values, the sizes of the original Web pages, and the sizes of the transformed pages. The numbers confirm that the tree height in practice is small but not necessarily two. When T increases, the tree height decreases. The total size of the transformed pages is usually a few times larger than that of the original page. However, because only the pages chosen by the user will be transferred to the client device, our system actually reduced the amount of transmission in the ten tasks, as previously shown in Figure 8.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a slicing*-tree based page transformation method for improving Web browsing on small terminals. In our approach, a Web page is transformed into a set of thumbnail index pages and leaf pages that form a multi-level tree structure with bounded node degree. We have implemented our algorithm in a proxy server and have demonstrated by experiments that our approach significantly improves user browsing experience in terms of task completion time, input effort, and network bandwidth consumption. In the future study, we plan to combine text summaries with thumbnail images so that users can locate the target blocks more accurately and quickly.

7. REFERENCES

[1] Avantgo. http://www.avantgo.com/.

[2] Plamscape: building a connected world. http://www.palmscape.com/.

[3] Proxinet, inc., proxiweb. http://www.proxinet.com/.

[4] Smartcode software, inc., handweb. http://www.smartcodesoft.com/.

[5] P. Baudisch, B. Lee, and L. Hanna. Fishnet, a fisheye web browser with search term popouts: a comparative evaluation with overview and linear view. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 133–140, New York, NY, USA, 2004.

[6] P. Baudisch, X. Xie, C. Wang, and W.-Y. Ma. Collapse-to-zoom: viewing web pages on small screen

devices by interactively removing irrelevant content. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 91–94, 2004.

[7] S. Bjork, L. E. Holmquist, J. Redstrom, I. Bretan, R. Danielsson, J. Karlgren, and K. Franzn. West: a web browser for small terminals. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 187–196, 1999.

[8] O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Focused web searching with pdas. In *Proceedings of the 9th international World Wide Web conference on Computer networks*, pages 213–230. North-Holland Publishing Co., 2000.

[9] O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Accordion summarization for end-game browsing on pdas and cellular phones. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 213–220, 2001.

[10] O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Seeing the whole in parts: text summarization for web browsing on handheld devices. In *WWW '01: Proceedings of the tenth international conference on World Wide Web*, pages 652–662, 2001.

[11] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd. Power browser: efficient web browsing for pdas. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 430–437, 2000.

[12] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Vips: a vision-based page segmentation algorithm. Technical Report MSR-TR-2003-79, Microsoft, 2003.

[13] L.-Q. Chen, X. Xie, W.-Y. Ma, H.-J. Zhang, H. Zhou, and H. Feng. Dress: A slicing tree based web page representation for various display sizes. In *WWW' 03: The Twelfth International World Wide Web Conference*, May 2003.

[14] Y. Chen, W.-Y. Ma, and H.-J. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW '03: Proceedings of the twelfth international conference on World Wide Web*, pages 225–233, 2003.

[15] Y. Chen, X. Xie, W.-Y. Ma, and H.-J. Zhang. Adapting web pages for small-screen devices. *Internet Computing, IEEE*, 9(1):50 – 56, Jan.-Feb. 2005.

[16] M. Lai and D. Wong. Slicing tree is a complete floorplan representation. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 228–232, Piscataway, NJ, USA, 2001. IEEE Press.

[17] D. Lee, K. Hoi, J. Xu, and W. Wang. Web browsing on small displays. *IEEE Distributed Systems Online*, 4(10), October 2003.

[18] N. Milic-Frayling and R. Sommerer. Smartview: Enhanced document viewer for mobile devices. *Microsoft Technique Report*, MSR-TR-2002-114, November 2002.

[19] B. N. Schilit, J. Trevor, D. M. Hilbert, and T. K. Koh. Web interaction using very small internet devices. *Computer*, 35(10):37–45, October 2002.

[20] J. O. Wobbrock, J. Forlizzi, S. E. Hudson, and B. A. Myers. Webthumb: interaction techniques for small-screen browsers. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 205–208, 2002.