SwiftPad: Exploring WYSIWYG LaTeX Editing on Electronic Paper

ABSTRACT

Electronic paper (i.e., e-paper) is a display technology that aims to imitate and substitute the conventional paper. Previous studies of e-paper mainly focus on evaluating or making practical use of its readability. However, there is little research to explore the potential of e-paper on input-oriented applications. In this paper, we introduce a novel document composition system named SwiftPad for e-paper. Specifically, SwiftPad integrates the LaTeX typesetting engine with the WYSIWYG concept, enabling users to directly edit well typeset LaTeX documents in their final print layout. Building such a system on resource-constrained e-paper with a low refresh rate creates unique challenges. We identify these challenges and name workable solutions. We also provide a usability evaluation of the new system. In short our finding is that the print aesthetics of LaTeX documents naturally fits the paper-like perception of e-paper and being able to edit them efficiently in their print forms provides writers with a calming and pleasant atmosphere.

INTRODUCTION

E-paper is a display technology that mimics the appearance of ordinary ink on printed paper. Owning to its excellent viewing experience including stable image, high contrast, wide viewing angle and non-glowing screen, e-paper has been widely adopted for various scenarios demanding high readability. The most typical application is the e-paper book readers, whose number is believed to exceed one billion in 2019 according to a survey from Research and Markets [21].

More recently, e-paper devices equipped with a 13.3-inch screen are increasing in popularity. Many manufacturers have been advertising their products as a digital device for writing and reading that feels like standard A4 paper. This design philosophy, however in reality, is not entirely implemented. Many devices indeed deliver decent reading experience to users, nevertheless few of them ever attach importance to the writing experience. Though there exists a limited number of document composition applications (e.g., drawing and sketching), they are mostly rudimentary in a sense that the generated documents, without a further polishing process in a PC, are seldom suitable for formal scenarios (e.g., office or scientific work). This deficiency mainly stems from the absence of typesetting functionality in the e-paper devices, causing a lack of aesthetics and formality in the generated documents.

In this paper, we aim to enhance the document composition and editing functionality in e-paper devices. We envision the new design meeting the following user experience goals:

High Typographic Quality: The system should produce documents with a high typographic quality, which can be directly applicable in formal occasions.

Simplicity: The user interface should only consist of basic editing primitives with a clear meaning, which allows users to concentrate on the actual document composition rather than tedious typesetting procedures.

To meet these requirements, we introduce a novel document processing system for generic e-paper devices, namely, SwiftPad. Specifically, SwiftPad integrates the LaTeX typesetting engine with the WYWIWYG editing concept, which enables users to directly edit well-typeset LaTeX documents in their final print layout. This novel composition brings about immense advantages. Firstly, the application of the acclaimed LaTeX typesetting engine delivers an excellent typographic quality in the generated documents. Secondly, the WYSIWYG interface can easily provide users with a simple and distraction-free editing experience. Most importantly, the intermediate results in the WYSIWYG editor are a faithful display of the final print form. It means users can instantly review their current editing results as if they were reading a well typeset document on an e-paper device, which is generally considered as a delightful and calming experience.

However, implementing such a system entails three major challenges. First, to function as a WYSIWYG editor, SwiftPad has to support not only faithful display, but also dynamic modification of PDF documents generated by LaTeX engines. However, a PDF document is essentially a vector graphic, which is generally considered difficult to modify. No reliable and open-source tools on e-paper are available yet for this demanding requirement.

In addition, the WYSIWYG editor imposes a high requirement on the system responsiveness. Specifically, it is expected to deliver the faithful print form of a LaTeX document in a reasonably small amount of time. However, this is a demanding target since a conventional

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

batch-processing LaTeX engine is generally slow; it may takes the order of seconds to compile a long document in a modern computer even with decent specifications.

The third major challenge stems from the notable drawbacks of e-paper, which are low refresh rate and ghost effects. This deficiency makes e-paper devices more suitable for displaying static contents rather than our WYSIWYG editor, which features constant screen update. Thus special optimization on e-paper devices is required in order to run our system smoothly.

In this paper, we present practical solutions to cope with the above challenges. Specifically, SwiftPad proposes a novel HTML5-based editor, which internally converts vector-based PDFs to semantic-based HTMLs so that the document contents can be faithfully displayed and easily modified by users. To ensure high reactivity of the system, SwiftPad employs a checkpoint technology on the LaTeX engine to accelerate the compilation process by skipping unnecessary computation. To combat the low refresh rate and ghost effects of e-paper, SwiftPad enhances the e-paper display driver with a heuristic multi-queue scheduling algorithm to support applications featuring fast screen update.

We consolidated the above techniques and implemented a prototype of SwiftPad on several popular off-the-shelf 13.3-inch e-paper devices, based on which we conducted a preliminary user study involving six participants to evaluate the usability aspects of the system. The evaluation results show that participants reacted positively to the innovative WYSIWYG editor for e-paper and praised its calming, distraction-free, and pleasant editing experience.

To sum up, the main contributions of this paper are as follows.

1. We demonstrate a novel system architecture for a document processing system on e-paper devices in Section 3.
2. We propose a HTML5-based WYSIWYG editor for LaTeX in Section 4.
3. We present a generic user-space checkpoint technology to accelerate the batch-style LaTeX engine in Section 5
4. We optimize the e-paper display driver to support applications that requires fast screen update in Section 6.
5. We conduct a preliminary pilot study evaluating usability aspects of the system in Section 7.

RELATED WORK

Readability of E-paper

E-paper has been generally considered to be a promising display technology in the field of reading. A number of research has been done to evaluate the readability of e-paper. An early work from Siegenthaler et. al [23, 24] analyzed and compared reading behavior on e-paper displays and on printed paper. The results suggest that the reading behavior on e-paper is highly similar to the reading behavior on printed paper. Another clinical research from Benedetto et. al [5, 4] evaluates readability of different electronic reading devices and one classic paper book. The results indicate that reading on the LCD-based screens triggers higher visual fatigue with respect to both the e-paper and classic paper books, while there exists no significant difference between e-paper and classic paper. A similar work from Zambarbieri et. al [31] conducted an analysis of the eye movements during silent reading of the e-paper devices and a printed book with the help of the video-oculographic eye-tracking technology. The experiments reveal that subjects' reading behaviour on e-paper devices is similar to reading from a printed book. It also suggested that reading in e-paper generated a higher level of reading performance than reading in a LCD device.

E-paper Applications

E-paper devices, thanks to high readability and commercial availability, have been applied in numerous useful applications across various domains. For instances, Chiu et al. [7] evaluates the potential of using the e-paper readers to encourage students to cultivate healthy reading behaviors. Blankenbach et al. [6] proposed a smart medicine package, which is equipped with e-paper driven by a bluetooth-enabled microcontroller. It aims to address the issue that today's packaging for pharmaceutics provides no information about individual medicine intake. Similarly AlterWear [8] presents an architecture for new wearable devices that implement a batteryless design using electromagnetic induction via NFC and e-paper displays.

However, most existing research works solely focus on evaluating or making use of readability of e-paper. There is little research on applying e-paper in scenarios other than reading. Recently, a research work [29] discussed the possibility of using E-paper devices in input-oriented applications that are realistic for both office work and school education. Motivated by this work, we have identified that a WYSIWYG editor for LaTeX documents would be a natural input-oriented extension for e-paper.

Integrating WYSIWYG with LaTeX

Previously, several attempts have been made to implement LaTeX based WYSIWYG editors (note that the notion of a WYSIWYG editor may be used in a somewhat loose manner). One preliminary approach is the 'Rich Text' feature adopted by Overleaf [19], where the source code is styled in different colors and fonts accordingly to the categories of terms. Still, there exists a strong visual disagreement between the source and output document and the user has to alternate and switch focus between them in many cases. A more sophisticated work LyX [14] is an open source document processor adopting a WYSIWYM approach, where what shows up on the screen is only an approximation of what will show up on

Figure 1. General Hardware Setup

the page. More recently, Bakoma TEX [25] and SwiftLaTeX [9] manages to deliver a faithful WYSIWYG editing experience for users in conventional PCs.

Though these works shed light on SwiftPad, they are not directly applicable due to two major reasons. First, they are originally designed for PCs and tend to have high system requirements. However, e-paper devices tend to have strict resource constraints. Secondly, the implementation of existing works do not take the special properties of e-paper screens into consideration, potentially leading to a poor user experience. Instead, SwiftPad presented in this work is specially optimized for e-paper devices from the ground up.

SYSTEM OVERVIEW

Fig.1 demonstrates the general hardware setup of SwiftPad, consisting an 13.3-inch e-paper device and an optional wireless bluetooth keyboard. We argue that keyboards are so far the most reliable input instrument for SwiftPad because of their popularity and users' familiarity. Nevertheless, we will discuss the possibility of integrating other input technologies such as speech recognition and handwriting input in Section 8.

The e-paper device is now running a set of software components, whose architecture is demonstrated in Fig. 2. The topmost layer is the user interface (UI). Its WYSIWYG editor enables users to view and edit the PDF documents in their print forms. One key insight of this UI layer is that it is fully implemented in modern web-based technology (i.e., HTML5, CSS3, and Javascript), which can deliver high portability among different e-paper devices, tablets and PCs. Besides the portability, the HTML5-powered user interface, compared with conventional PDF viewers, features higher interactiveness and faster rendering on devices with low-end specifications [20, 28]. This is crucial because most off-the-shelf e-paper devices only possess limited computation ability (e.g., single-core 1GHz ARM CPU).

The component beneath the UI is the system utilities, which provides the UI with the runtime environment and typesetting functionality. For example, the Webkit engine is in charge of parsing and rendering the HTML5 pages from the UI and the LATEX engine handles document compilation requests. It is worth noting that, the typesetting engine has been enhanced with a userspace checkpoint technology, which reduces the compilation time by skipping the unmodified pages and makes the WYSIWYG editor more responsive.

The bottom layer is the Operating System (OS), which provides an embedded Linux environment (Busybox) implemented with size-optimization and limited resources in mind. It also contains an enhanced e-paper display driver, which significantly improves the viewing experience of e-paper applications that require constant screen update.

Figure 2. System Architecture

Note that most off-the-shelf e-paper devices simply adopt Android OSs, which we argue is not an ideal solution. Notably, Android has been designed to enable multimedia functionalities (e.g., gaming and video playback) in conventional mobile phones or tablets. Thus, Android comes with a great number of multimedia system services that are not utilized by e-paper devices. This leads to waste of system resources and unnecessary battery drain. In contrast, our system architecture is

Figure 3. User Interface of SwiftPad.

tailored for the e-paper devices with low specifications from the ground up. It is thus more resource-efficient and potentially provides a longer battery life.

USER INTERFACE DESIGN

We design a simple-yet-powerful user interface for SwiftPad as shown in Fig. 3. Note that the screenshot is directly captured from a e-paper device's graphics memory in pursuit for better presentation purposes. It can be seen that SwiftPad offers two different editing views. The first one is the source view, which allows advanced users to directly manipulate LaTeX source code in a classical ASCII editor. This view needs to be preserved because only in source code can arbitrary LaTeX scripting be done.

A WYSIWYG Editor for LaTeX

Nevertheless, the main contribution lies in the WYSIWYG view. This is an editable PDF viewer that allows the user to directly edit a document in its print form, but with effect on the source. More specifically, the viewer possesses the following features:

1. At editing quiescence (i.e., a moment when the editor has processed all previous edits of the user, and there are hence no pending edits that would further change the output), the editor shows the print layout of the PDF document, i.e., acts as a faithful print viewer.
2. At editing quiescence, the user can position the cursor anywhere in the document with the mouse/touchscreen, arrow keys or a combination thereof.
3. The user can perform edits at the cursor position by simply typing the keys or backspace and get immediate feedback in the sense that the editor shows a *preview version* of the print view. This is mainly achieved by mimicking the typesetting behavior used in the LaTeX engine. For instance, when a character is being appended, it will automatically inherit the font settings from the previous characters to make itself visually agreeable.
4. To retain the modification, the user's editing operations will also be applied at the corresponding position of the LaTeX source code. (This requires the editor to possess the ability to infer the source code position of each element in the PDF with character-wise accuracy. In our implementation, we achieve this with the help of a LaTeX plugin introduced in the work [9]. In short, the plugin enables position inference by patching the typesetting engine with a bookkeeping mechanism, where it constructs a position record for each character in the input source file and output them to the PDF file as metadata (also known as 'Tagged PDF'). The position info in the PDF then can be retrieved by our specifically-designed editor, while being safely ignored by the conventional PDF viewers.)
5. If the user input pauses, the editor reaches editing quiescence automatically in a reasonable amount of time. It is achieved by replacing the current preview version with the latest compilation result, i.e., faithful output, from the LaTeX engine.

These features also constitute a precise definition for the notion of a WYSIWYG editor for LaTeX used in our paper. We envision that such a WYSIWYG editor is satisfactory for e-paper devices where a sufficiently large proportion of scientific/technical documents could be edited.

Faithful Conversion from PDFs to HTMLs

To implement such a WYSIWYG editor, our system has to support not only faithful display, but also dynamic modification of PDF contents (i.e., instantly generating the preview version in response to the user's input). However, it is considered challenging because a PDF document is essentially a vector graphic, which is difficult to edit. Though there exist a limited number of commercial closed-source products (e.g., FoxIt PhantomPDF viewer

Figure 4. Workflow of the conversion algorithm

[11]) which support simple 'touch up' operations such as adding or removing words, we argue that these solutions are not directly applicable for e-paper devices. First, they are originally designed for PCs and tend to pose unacceptable computational burden on resource-constrained e-paper devices. Moreover, these products are generally not portable among devices with different hardware specifications and software architectures.

To address these issues, we explore a novel approach to implement the WYSIWYG view by converting PDFs to HTML pages in a nearly faithful manner. Displaying HTML pages brings about a wide range of advantages. First, HTML pages are of high portability and can be displayed in various devices equipped with browsers. Meanwhile, rendering HTML pages in modern browsers is a relatively lightweight operation even for devices with low-end specifications. Most importantly, unlike vector-based PDFs, HTML pages are semantic-based; texts in HTML pages are typically surrounded by HTML semantic markup, which allows them to be dynamically modified with the help of Javascript. It facilitates the implementation of commonly-used word processing functionalities (e.g., text selection, insertion, deletion, copying and pasting) in the WYSIWYG view.

To achieve a nearly faithful conversion, SwiftPad employs the following mechanisms to preserve the following kinds of elements in PDFs.

Images. PDF supports graphical instructions such as drawing and image embedding. To preserve the graphic elements, we rasterize them into bitmap images, which then can be displayed using the CSS sprite technique [26].

Font Embedding. Fonts can be embedded in a PDF file, which ensures that readers always see the text in its original font. To preserve the fonts in a PDF, we extract all the fonts from the PDF and convert them into web open font types (WOFF) with the help of two third-party libraries MuPDF [15] and FontForge [30]. The converted fonts then can be embedded in HTML pages using a CSS rule named 'font-face' [26].

Text Locations. Text elements are positioned with absolute coordinates in a PDF document. To preserve the locations in the HTML pages, one naive method is to convert the locations into CSS absolute position rules and assign them to HTML text elements. However, since each text segment is now associated with a unique CSS rule, the page may be too bulky to store and transfer. Moreover, the absolute positions are not flexible; when a user inserts or deletes some texts, a large number of CSS position rules must be changed accordingly in order to achieve text reflow functionality.

Inspired by a more recent tool [27], we attempt a relative positioning method to position each text segment. A simple example in Fig. 4 demonstrates the intuition of our approach. Specifically, we first attempt to merge PDF text segments to text lines based on their geometric metrics. Afterwards, we measure the space width between words in each line and turn them into CSS rules. Finally, these rules can be used to construct HTML spacer elements (i.e., empty span elements in a certain width) to help position each text element from left to right in each line. The advantage of this approach is that the number of generated CSS rules tends to be tiny since there are usually limited number of spacers with different widths in a PDF page. We can further reduce the number by merging some rules which have nearly identical width values at the cost of faithfulness in text locations. As a result, this approach can generate HTML pages with a much smaller size compared with the naive approach. Another important advantage is that this positioning method automatically enables text reflow functionality to a certain extent when texts are inserted or deleted.

ACHIEVING HIGH RESPONSIVENESS

One essential requirement for WYSIWYG editors is high responsiveness; it allows users to instantly see what the end result will look like while a document is being edited. In the context of SwiftPad, when a user types a key, our editor provides immediate response in the sense that it generates and shows a preview version of the document by mimicking the typesetting behavior used in the LaTeX engine. Though the typesetting imitation approach is feasible for minor editing, it possesses certain limits when dealing with lengthy edits. As a consequence, without any further precautions, the difference between the previewing version and the faithful output from the LaTeX

engine would gradually accumulate along with the user's input. To address this issue, our editor periodically replaces the current preview version with the latest compilation result from the LaTeX engine.

However, this replacement operation potentially leads to unresponsiveness of our editor due to the long turn around time of each compilation. A complication process on conventional engines, depending on the complexity of the input source codes, can take several seconds to complete even on a computer with decent specifications (e.g., Intel i7 6900 with DDR4 memory and SSD storage). As a result, users may have to wait noticeable timespan in order to view the faithful output.

Accelerating LaTeX Compilation Using Checkpointing

We find that the inefficiency of LaTeX engines may be attributed to the following two reasons. First, the LaTeX engine, which was programmed decades ago, does not utilize the multi-threading feature of the modern machines. Thus, its execution speed is bounded by the clock rate of a single CPU core regardless of the number of cores. Secondly, LaTeX is a batching system, which implies that every time a compilation is initiated, the LaTeX engine must process the input files from the very beginning. Such behavior is undesirable considering that, in most cases, users only append or modify characters located at the end of the input file, while leaving the preceding contents unchanged. Therefore, recompiling the unchanged contents leads to a considerable amount of repeated and unnecessary computation.

To accelerate the complication process, one potential approach is to overhaul the source code of the LaTeX engine to add multi-threading support. However, this requires extensive reworking of the engines. More importantly, it may introduce unheeded bugs undermining the software stability. Instead, we shift our attention on the second cause mentioned above and seek method to avoid the repeated computation between consecutive compilations. The philosophy we apply here is called checkpointing, which consists of saving a snapshot of an application's state, so that it can restart from that point in the future. In the context of our enhanced LaTeX engine, we create checkpoints periodically (e.g., after outputting a PDF page) and mark down the corresponding input file positions during the compilation process. When a new compilation job is submitted, based on the file position that users just edit on, the enhanced engine can determine the closest checkpoint and directly start from that point to skip the repeated computation.

Another optional trick to optimize responsiveness is that the engine does not have to process the whole input file in most cases. Instead, it may stop right after generating the page the user is currently viewing or editing. By combining this trick with the checkpoint technology, we can ensure the response time of the engine remains nearly constant regardless of the page number of documents. For instance, when the user is working on page 5, the engine can start from the checkpoint, which was generated when page 4 was being outputted, and only re-typeset the page 5. Though in some rare situations, contents from page 6 onwards may slightly affect the typesetting results of the page 5. We argue that the infrequent loss of faithfulness is negligible and would not seriously impact the overall user experience.

Userspace Checkpoint Technology for LaTeX

Several runtime checkpoint implementations have been proposed. Early implementations are mainly kernel-based [12]. They utilize a specially-designed kernel module to save or restore process-related data structures in the kernel. However, the existing in-kernel implementations do not focus on upstream compatibility. As a result, it is very difficult to integrate them into recent mainline kernels and these implementations are therefore not further developed and abandoned. To solve the issues of the in-kernel implementations, CRIU [10] proposes another approach; it implements as much functionality as possible in the user space and solely uses existing kernel interfaces. Despite the promising features of CRIU, it is a relatively heavyweight solution since it was originally designed for virtual machines or containers. It thus has to checkpoint a wide range of system-wide information not utilized by the LaTeX engine (e.g., TCP sockets and process trees). This results in inefficiency of each checkpoint operation, which can take multiple seconds to finish.

These issues motivate us to propose a lightweight userspace checkpoint technology for the LaTeX engine. Specifically, we utilize the dynamic hooking technique [1] to patch the engine on the runtime with the checkpoint logic. Each checkpoint operation solely saves three types of information including
1) CPU registers (e.g., Instruction Pointer)
2) memory segments including data, bss and heap
3) file position for each open file
and thus can be swiftly carried out (usually in less than 50 ms).

To fully recover the state of the LaTeX engine, a special mechanism is required for memory segments. At first glance, restoring memory segments seems quite straightforward; we could simply write back the contents from the dump file to the original memory addresses. However, it is highly likely to fail due the stateful nature of the heap segment. Unlike data or bss segments whose sizes are pre-determined during the compilation, the size of the heap segment can vary constantly during the runtime. Specifically, when an executable allocates/releases a memory block via the standard C library calls 'malloc'/'free', these functions internally invoke a system call 'sbrk' to extend/shrink the heap segment. To efficiently manage the heap segment, the standard C library has to keep track of an important system state, namely, the current heap segment size. When the program is recovering from a checkpoint, in order for the C library to continue functioning properly, this system state has to be restored as well. However, the restoration functionality is gener-

Figure 5. Compilation time comparison between the conventional Engine and the enhanced engine.

ally not available in popular runtime C libraries (e.g., libc).

To address this issue, we patch the built-in heap allocator in the runtime C library such that rather than *sbrk*, it now uses *mmap* as the basic mechanism to obtain memory from the system. The advantage of using mmap is that it allows us to create a heap memory region that has been assigned a direct byte-for-byte correlation with a filesystem object. By duplicating the file, we will be able obtain to a snapshot of the heap memory region. To restore the region, we could simply map the snapshot file back to memory using *mmap* again. This approach ensures that not only the heap segment contents, but also the heap segment size can be correctly restored across runs.

Special care is also given to the file objects, which are kernel objects and do not persist between runs. To address this issue, we will re-initialize every file object by reopening them and restore their file pointer positions by using the function *fseek*.

To demonstrate the efficiency of our approach, we compare the compilation time of the conventional LaTeX engine and our optimized engine in a same device with a single-core 1GHz ARM CPU and 512 MB RAM. Specifically, we first use a predefined set of LaTeX code snippets to randomly generate a variety of LaTeX sample documents, whose page numbers range from 1 to 100. Afterward, we insert texts at random places of each document and then start the compilation. Such a procedure will be repeated 100 times for each document and the average compilation time will be reported. The results are demonstrated in Fig. 5. It can be seen that the compilation time of the conventional LaTeX engines increases proportionally with the page number of the LaTeX documents. In the worst case, the 100-page document even consumes approximately 13 seconds. We again conduct the measurements on our optimized engine. It can be seen that, regardless of the page number, the compilation time stays level (112 ms), which ensures high responsiveness to our editor.

OPTIMIZING THE E-PAPER DISPLAY DRIVER

Existing e-paper devices are mainly designed for displaying static contents. As a result, they are not well optimized for applications like the WYSIWYG editor, which requires constant screen update. In this section, we will elaborate our enhancement to the e-paper display driver to address this issue.

How E-paper Screens Work?

Before we can demonstrate the intuition of our enhancement, we first explain the internal display mechanism of e-paper screens. An e-paper screen is controlled by a specifically-designed circuit, the Electrophoretic Display Controller (EPDC). It is responsible for driving corresponding electrical signals to the e-paper panel to update the screen contents upon receiving display update commands from the CPU. Most update commands can be described as a tuple (x, y, w, h, *data*, *mode*). Specifically, the first five parameters denote the coordinate, size and content of the rectangle region pending to be updated. The *mode* parameter denotes the update mode, whose possible options include 2, 4, 8, or 16 graylevels. The 16-graylevel update mode delivers the best display quality (i.e., highest contrast and little ghost effect) but consumes the longest timespan to finish (approximately 1 second). The 2-graylevel update modes enables fast animation of the screen contents (approximately 150 ms) but may generate poor contrast texts and significant ghost effects.

To communicate with the EPDC, the OS requires a kernel driver. One important task of the driver is to composite a update request tuple. The system can obtain the first five parameters (i.e., x, y, w, h, *data*) directly from the upper layer of the OS. For the update mode, this info is typically not available since most OSs are designed for LCD screens which do not require the update mode information. Therefore, the drivers have to provide a mechanism to infer the update mode. However, most existing driver implementations possess an somewhat rudimentary update mode selection mechanism. One of the most widely-used mechanisms is to provide a kernel interface to allow an application to manually specify the global update mode. For instance, an application featuring constantly varied contents can enforce the 2-graylevel update mode; on the other hand, a reading application requiring high display quality can enforce 16-graylevel update mode. However, this mechanism is relatively coarse-grained and may not be suitable for applications that simultaneously require fast response time and high display quality, so does our WYSIWYG editor.

Heuristic Multi-queue Scheduling Algorithm for E-paper Display

In our work, we optimize the existing e-paper display driver to automatically strike a balance between the display quality and response time with a heuristic multi-queue scheduling algorithm.

Figure 6. Multi-Queue scheduling algorithm

As shown in Fig 6, our algorithm maintains three separate update queues including a manual queue, a paint queue and a repaint queue. The manual queue is a simple queue which only stores the incoming jobs that have update modes specified by the developer. Specifically, we enhance the Webkit browser engine with an interface, which enables the HTML applications to choose a desired update mode for each HTML element by using a HTML data attribute named *data-updatemode*. For instance, when the browser is rendering the HTML text element *Open*, the screen update request will be sent to the manual queue and later painted using the 4 graylevel mode.

Nevertheless, most of the time developers do not manually specify the update mode. In this case, the paint and the repaint queue will be used. Specifically, the paint queue stores all the incoming update requests without update mode info from the application, which will then be sequentially drawn on the screen using 2-graylevel mode. Intuitively, this ensures the e-paper screen display the latest contents as soon as possible (i.e., to deliver a responsive user experience). After leaving the paint queue, the update job will enter the repaint queue and be later repainted again using 16-graylevel mode to improve the display quality (i.e., alleviate ghost effects and show high-contrast texts). In other words, if no new update jobs are submitted to the driver, the high display quality of the screen contents will eventually be achieved.

This algorithm also takes advantage of the simultaneous update feature of the EPDC, which allows multiple update requests to be processed at the same moment when their update regions do not overlap with each other. In other words, the painting jobs and repainting jobs may be carried out simultaneously if their update regions do not collide. This condition frequently holds true for our editor application. A typically example has been depicted in Fig. 7; at stage one, the user types a word 'hello', which is drawn in 2 graylevel mode. At stage two, the user types another word 'world', this word is still drawn in 2 graylevel mode while the previous word 'hello' can be repainted using 16 graylevel mode at the same time since the update regions of the two words do not collide. At the last stage, the user stops typing, so the word 'world' is repainted and all the words are now in 16 graylevel mode.

Figure 7. A demonstrating example of the scheduling algorithm

Note that the max number of simultaneous update regions is decided by available hardware resources of the EPDC and may be quite limited. When the application is overloading the EPDC by submitting too many update requests at the same time, we prioritize the three queues to optimize the average screen response time (manual >paint >repaint). For a update request in a queue to be executed, two conditions must be met; 1) the EPDC has available resources, 2) all the update requests in those higher priority queues should have either finished the execution or been scheduled (i.e., being drawn).

EVALUATION

In this section, we demonstrate the settings and outcomes of our evaluation on usability of SwiftPad.

Experiment Devices

In our experiments, we deploy SwiftPad on two popular off-the-shelf 13.3-inch devices BOOX MAX [13] and reMarkable [3]. The devices are built atop an NXP i.MX 6 Solo SoC [2], which is equipped with a single-core 1Ghz CPU, 512MB DDR3 RAM and an EPDC. Since the SoC's bootloader and Linux kernel source codes are both publicly available on the Internet, we can easily modify the e-paper devices to meet our requirements (e.g., installing the enhanced e-paper driver). Note that since SwiftPad features an implementation with high portability, it takes little effort to port SwiftPad to other devices with different hardware architectures or screen sizes.

Usability Evaluation

The evaluation has been carried out as a Discount Usability Test, which involves a small number of participants with a focus on qualitative studies on prototype design [16]. Existing studies [18][17] have suggested that a discount usability test with only 5 participants can offer reliable evaluation results and may identify up to 85% of the usability problems. In our work, we invited 6 participants from academia to evaluate our system and gathered many important insights and feedbacks. In the near future, we are planing to carry out a comprehensive usability study, which involves more participants from different domains.

Before undertaking the test, each subject was instructed to fill in a short questionnaire, which contains the following categories of questions on a five-point scale (strongly disagree, disagree, neutral, agree, strongly agree):

- Frequency and enjoyment of writing.

Table 1. Axial coding of the participants' comments

Categories	Positive	Negative
Paper-like viewing experience	6	0
Aesthetics	5	0
Simplicity	4	0
Low distraction	2	0
Latency	1	3
Keyboard input	0	1
Structural change	0	1
Math editing	1	1

- Familiarity of e-paper devices (i.e., how often do they read in e-paper).

- Experience in various typesetting programs.

After that, each participant was instructed to perform the following tasks in an unsupervised manner; the subjects had to finish each task by themselves without any prior knowledge of SwiftPad.

1. The participants were required to compose an essay with at least 500 words, describing their current research focus.

2. Each participant was provided with an one-page English article on a topic of general interest. In each line of the article, there existed an obvious typo and the participants were supposed to correct it.

After completing these tasks, we required the participants to complete a System Usability Scale (SUS) questionnaire, which is a commonly-used reliable tool for perceived usability evaluation even with a small sample size. We also encouraged participants to report their experience of using SwiftPad.

SUS Score. The mean SUS score of SwiftPad is 71 (in a range spanning from 0 to 100). It exceeds the threshold score of 68 which indicates a decent level of usability. In addition, we also compute the mean values for the usability sub-scale and learnability sub-scale, which are 70 and 75 respectively.

Meanwhile, by considering participants' answers given in the background questionnaire, we obtain three more background sub-scores. They are compared with the SUS score and sub-scale scores to obtain the Pearson correlation coefficients. We notice a weak positive correlation between the user's familiarity of e-paper and the learnability sub-score, which implies that users who get used to reading in e-paper devices may find it easier to master our system. However, due to the small sample size, each correlation measure appears to be not statistically significant. To reach a more solid conclusion, a more comprehensive dataset is required.

User Feedback. All the participants' comments are summarized in Table 1 using Axial coding. On one hand, most participants praised the natural combination of excellent viewing experience of e-paper and print aesthetics of LaTeX documents. Meanwhile, this system still maintains simplicity; with the WYSIWYG editor that possesses simple editing primitives and high familiarity with existing word processors, participants can efficiently concentrate on the actual document composition or editing without mastering the complex LaTeX scripts. Two participants further commented that conventional PCs are usually equipped with noisy cooling fans and glowing screens leading to eye strain. In contrast, e-paper devices are quiet and their paper-like non-glowing screens provide a calming and almost distraction-free atmosphere for document writers.

On the other hand, Table 1 also reveals some widely perceived issues. One important issue highlighted was the screen latency. Despite our effort to optimize the graphics stack, most participants, who get used to conventional LCD screens, can still perceive the display latency due to the hardware limit of e-paper screens. Nevertheless, the participants were happy to accept the minor imperfection, which has limited impact on the usability of SwiftPad. One participant even provided an interesting argument that conventional typewriters tend to have much higher latency, however, impeding no functionality. Meanwhile, one participant encouraged us to explore other input technologies other than keyboard. He suggested that handwritten input would also be a natural input method for e-paper devices. Another issue was the rudimentary functionality to apply structural changes (e.g., adding a new section or list structure). Currently, these structural changes could be applied via clicking the corresponding options in the context menu (i.e., pop-up menu). However, one subject suggested us to mimic the autoformat features (e.g., automatic bullets or numbering) from Microsoft Word. A related issue is about mathematical formulas editing. One subject noticed and praised the functionality of in-place mathematical formulas editing where users are allowed to change the text in an existing formula just like normal text. This functionality is quite useful in many scenarios, for instance, correcting the simple mathematical equations like $1 - \frac{1}{4} = \frac{2}{4}$. However, the subject also wished us to propose a method to insert or edit more complicated formulas without manipulating the LaTeX source codes.

FURTHER DISCUSSIONS

SwiftPad shows the potential for WYSIWYG editing on an e-paper device. Still, it bears several limitations that need further improvement.

Exploring Alternative Input Technologies

Currently, SwiftPad adopts keyboards as the main input technology due to their popularity and users' familiarity. Nevertheless, we also realize the potential of other alternative input options, for instance, voice input and handwriting input. SwiftPad currently provides experimental support for these two input options for users with the help of Google's voice API and text recognition API

respectively. We plan to conduct a usability study in the future to evaluate the efficiency of each input method under different scenarios.

Display Scheduling Algorithms

SwiftPad employs a multiple-queue scheduling algorithm in the graphics driver. Despite the lack of performance analysis, the heuristic scheduling usually finds a solution close to the optimal one in a swift manner. In the future work, we plan to reformulate the scheduling process as an optimization problem with the help of the queueing theory. We may be able to find an algorithm that computes the optimal scheduling orders or less ideally an approximate algorithm with performance guarantees.

Facilitating Structural Edits

One challenge of WYSIWYG tools is to find a natural interface for structural edits. Our preliminary design is a menu displaying pre-defined LaTeX snippets or formulas, which are examples of the needed elements. For instance, a section header snippet could be "`\subsection{Sectiontitle}`". In the menu, this is shown in its compiled version. If the user selects the snippet, essentially a paste (insert) of the snippet at the current cursor position is performed, which can be subsequently adapted by the user. This reduces the structural change to a simpler concept, the paste concept. It also provides a natural visual appearance and is in principle end-user-programmable.

Collaborative Editing

SwiftPad provides experimental support for real-time collaboration where multiple SwiftPad devices can be allowed to work on the same document. In details, this is implemented using an operational transformation database named ShareDB [22], which is invented for consistency maintenance and concurrency control in collaborative editing of text documents. A version control system is also implemented to allow users to track changes between different versions.

CONCLUSION

In this paper, we present SwiftPad, a novel document composition system for e-paper. It integrates the LaTeX typesetting engine with the WYWIWYG editing concept, enabling users to directly edit and instantly review documents in their final print layout. We have identified fundamental architectural challenges and named the corresponding resuable solutions. We have also conducted a user study, where we found positive feedback that the system provides a pleasant document viewing and editing experience for writers.

REFERENCES

1. Dynamic linker hooking techniques, black`http://man7.org/linux/man-pages/man8/ld.so.8.html`
2. i.mx6 quad applications processors, black`https://goo.gl/mqtw9v`
3. remarkable:paper-like tablet, black`https://remarkable.com/`
4. Benedetto, S., Carbone, A., Drai-Zerbib, V., Pedrotti, M., Baccino, T.: Effects of luminance and illuminance on visual fatigue and arousal during digital reading. Computers in human behavior **41**, 112–119 (2014)
5. Benedetto, S., Drai-Zerbib, V., Pedrotti, M., Tissier, G., Baccino, T.: E-readers and visual fatigue. PloS one **8**(12), e83676 (2013)
6. Blankenbach, K., Duchemin, P., Rist, B., Bogner, D., Krause, M.: 22-2: Smart pharmaceutical packaging with e-paper display for improved patient compliance. In: SID Symposium Digest of Technical Papers. vol. 49, pp. 271–274. Wiley Online Library (2018)
7. Chiu, P.S., Su, Y.N., Huang, Y.M., Pu, Y.H., Cheng, P.Y., Chao, I.C., Huang, Y.M.: Interactive electronic book for authentic learning. In: Authentic Learning Through Advances in Technologies, pp. 45–60. Springer (2018)
8. Dierk, C., Nicholas, M.J.P., Paulos, E.: Alterwear: Battery-free wearable displays for opportunistic interactions. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. p. 220. ACM (2018)
9. Elliott Wen, G.W.: Swiftlatex: Exploring the true wyswiyg editing for publication. DocEng (2018)
10. EMELYANOV, P.: Criu: Checkpoint/restore in userspace, july 2011
11. Foxit Reader: PDF Viewer from Foxit Software. black`https://goo.gl/w1918D` (2019)
12. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (blcr) for linux clusters. In: Journal of Physics: Conference Series. vol. 46, p. 494. IOP Publishing (2006)
13. Intl, O.: Onyx boox electronic books, black`https://onyxboox.com`
14. Kastrup, D.: Revisiting wysiwyg paradigms for authoring latex. COMMUNICATIONS OF THE TEX USERS GROUP TUGBOAT EDITOR BARBARA BEETON PROCEEDINGS EDITORS KAJA CHRISTIANSEN **23**(1), 57 (2002)
15. mupdf: Mupdf, a lightweight pdf, xps, and e-book viewer (2018), black`https://mupdf.com`
16. Nielsen, J.: Usability engineering at a discount. In: Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.). pp. 394–401. Elsevier Science Inc. (1989)

17. Nielsen, J.: Discount usability: 20 years. Jakob Nielsen's Alertbox Available at http://www. useit. com/alertbox/discount-usability. html [Accessed 23 January 2012] (2009)
18. Nielsen, J.: Why you only need to test with 5 users, march 19, 2000. Useit.com Alertbox **27** (2015)
19. Overleaf: Overleaf, collaborative writing and publishing (2018), blackhttps://www.overleaf.com
20. Peroni, S., Osborne, F., Di Iorio, A., Nuzzolese, A.G., Poggi, F., Vitali, F., Motta, E.: Research articles in simplified html: a web-first format for html-based scholarly articles. PeerJ Computer Science **3**, e132 (2017)
21. Research and Markets: e-Paper Display Market 2016-2020. blackhttps://goo.gl/PvVHTs (2018)
22. ShareDB: Sharedb: A database frontend for concurrent editing systems (2011), blackhttps://goo.gl/wAT43N
23. Siegenthaler, E., Wurtz, P., Bergamin, P., Groner, R.: Comparing reading processes on e-ink displays and print. Displays **32**(5), 268–273 (2011)
24. Siegenthaler, E., Wurtz, P., Groner, R.: Improving the usability of e-book readers. Journal of Usability Studies **6**(1), 25–38 (2010)
25. Soft, B.: Bakoma tex 9.77 (2011), blackhttp://www.bakoma.com
26. Souders, S.: High-performance web sites. Communications of the ACM **51**(12), 36–41 (2008)
27. WANG, L., LIU, W.: Pdf2htmlex (2013), blackhttps://goo.gl/89SDo6
28. WANG, L., LIU, W.: Performance comparison between pdfs and htmls (2013), blackhttps://goo.gl/s2yd1x
29. Wen, E., Weber, G.: Going grey: Exploring the potential of electrophoretic displays. In: Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers. pp. 1761–1764. ACM (2018)
30. Williams, G.: Font creation with fontforge. EuroTEX 2003 Proceedings, TUGboat **24**(3), 531–544 (2003)
31. Zambarbieri, D., Carniglia, E.: Eye movement analysis of reading from computer displays, ereaders and printed books. Ophthalmic and Physiological Optics **32**(5), 390–396 (2012)