SwiftPad: Exploring WYSIWYG LaTeX Editing on Electrionic 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 readability while attaching little importance to usability in document writing. In this paper, we aim to enhance the document composition and editing functionality on e-paper by introducing a new system named SwiftPad. Specifically, SwiftPad integrates the LaTeX typesetting engine with the WYSIWYG editing concept, which enables users to directly compose and edit documents in their final print layout. Meanwhile, the print aesthetics of well typeset LaTeX documents naturally fits the paper-like perception of e-paper, providing an excellent atmosphere for document composition in an analogy to a typewriter. In this paper, we identify unique challenges when building such a system and name workable solutions. We also provide a usability evaluation of the new system.

1 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, and wide viewing angle, e-paper has been widely adopted for various scenarios demanding high readability. The most typical application is the e-paper book readers (also commonly referred to as ereaders), whose number is believed to exceed one billion in 2019 according to a survey from Research and Markets [21].

More recently, ereaders 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 ereaders indeed deliver decent reading experience to users, nevertheless few of them ever attach importance to the writing experience. Notably, though there exists a limited number of document composition applications (e.g., drawing and sketching), the generated documents are seldom suitable for formal scenarios (e.g., education and office work) without a further polishing process in a PC. A particular deficit stems from the absence of typesetting functionality in the ereaders, 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 ereaders. 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 system should be sufficiently simple to use. This implies that the user interface should only consist of basic editing primitives with a clear meaning.
Low Distraction. The system should allow 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 ereaders, namely, SwiftPad. Specifically, SwiftPad integrates the LaTeX typesetting engine with the WYWIWYG editing concept, which enables users to directly edit various documents in their final print layout. This novel composition brings about immense advantages. Firstly, the application of the acclaimed LaTeX typesetting engine delivers excellent typographic quality in the generated documents. Secondly, the WYSIWYG interface can easily provide users with a low-distraction and simple 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 ereader, which is generally considered as a delightful and calming experience.

First, implementing such a system entails three major challenges. First, to function as a WYSIWYG editing system on LaTeX documents, the editor has to support not only faithful display, but also dynamic modification of PDF contents. When a modification is detected, the editor is further required to infer the corresponding source code position with character-wise accuracy in order to alter the underlying source file synchronously. However, no open-source tools on ereaders are available yet for these demanding requirements.

In addition, a WYSIWYG editor imposes a high requirement on the system responsiveness. It means the editor should issue a prompt response to user inputs, specifically, it should deliver the final print form in a reasonably small amount of time. This is a demanding target since it usually takes the order of seconds for the batching-style LaTeX engine to process a long document in a modern computer even with decent specifications.

The third major challenge is that the existing operating systems on e-paper readers are mainly designed for displaying static contents. As a result, they are not well optimized for the WYSIWYG editor which requires constant screen update. The interface designer of the editor has to deal with the notable drawbacks of e-paper such as low refresh rate and ghost effects.

In this paper, we present practical solutions to cope with the above challenges. Specifically, SwiftPad proposes a novel HTML5-based user interface and an enhanced TeX engine to enable PDF content modification and character-wise mapping between PDF elements and source codes. To ensure high reactivity of the system, SwiftPad employs a checkpoint technology on the TeX engine to accelerate the compilation process by skipping unnecessary computation. To combat the low refresh rate and ghost effects of e-paper reader, SwiftPad enhances the epaper display driver with a heuristic multi-queue scheduling algorithm.

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

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

1. We demonstrate a novel architecture for a document processing system on epaper 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 and its application to accelerate the batch-style LaTeX engine in Section 5
4. We optimize the display driver for epaper devices in Section 6 to deliver fast screen update and high display quality.
5. We conduct a preliminary pilot study evaluating usability aspects of the system in Section 7.

2 Related Work

2.1 Readability of E-paper

E-paper provides a unique user experience for viewers owning to its stable image, wide viewing angle, high readability, and non-glowing screen. These distinct characteristics make e-paper a promising display technology in the field of reading. An early work from Siegenthaler et. al [23, 24] from analyzed and compared reading behavior on e-reader 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 Benedetto et. al [4, 3] 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 classical 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 videooculographic 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.

2.2 Use Cases of E-paper

E-paper, thanks to its commercial availability, has been applied in numerous useful applications across various domains For instances, Chiu et al. [6] evaluates the potential of using the e-paper readers to encourage students to cultivate healthy reading behaviors. Blankenbach et al. [5] proposed three smart packages for those

examples with graphics epaper driven by a microcontroller with Bluetooth LE and a smartphone app. It aims to address the issue that today's packaging for pharmaceutics provides no information about individual medicine intake neither remind when forgotten or alert when medicine runs empty. Similarly AlterWear [7] presents an architecture for new wearable devices that implement a battery-less 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-heavy applications scenarios that are realistic for both office work and school education. Motivated by this work, we have identified that a WYSIWYG editor just for PDF documents created with the LaTeX typesetting program would be a natural input-oriented extension for e-paper.

2.3 Integrating WYSIWYG with LaTeX

Previously, several attempts have been made to implement LaTeX based WYSIWYG editors, though the notion of a WYSIWYG editor may be used in loose manner. One preliminary approach is the 'Rich Text' feature adopted by Overleaf [18], 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 [13] 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 the page. More recently, a commercial tool Bakoma TeX [25] manages to deliver a faithful WYSIWYG editing experience for standalone computers. Elliott et al. [8] takes it a step further and implements a cloud-based LaTeX WYSIWYG editor.

Though these works shed light on SwiftPad, they cannot be 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 typically 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.

Fig.1 demonstrates a general hardware setup of SwiftPad, consisting an ereader 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.

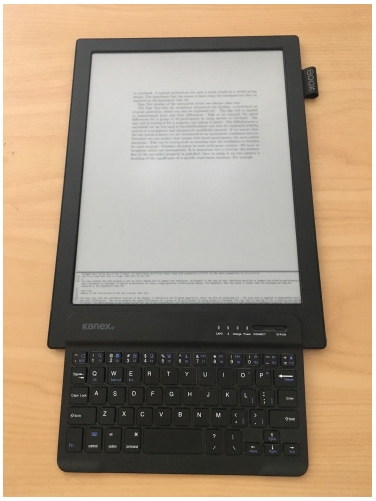

Fig. 1. General Hardware Setup

3 System Overview

The ereader 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 [19, 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 runtime environment and typesetting functionality for the UI. For example, the Webkit engine is in charge of parsing and rendering the HTML5 scripts of the UI. The LaTeX engine handles document compilation requests submitted by the UI. 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 a Linux-based Operating System (OS). In this layer, SwiftPad enhances the epaper display driver to automatically strike a balance between the display response time and display quality with a heuristic scheduling algorithm. It significantly improves the user experience of e-paper applications that require constant screen update.

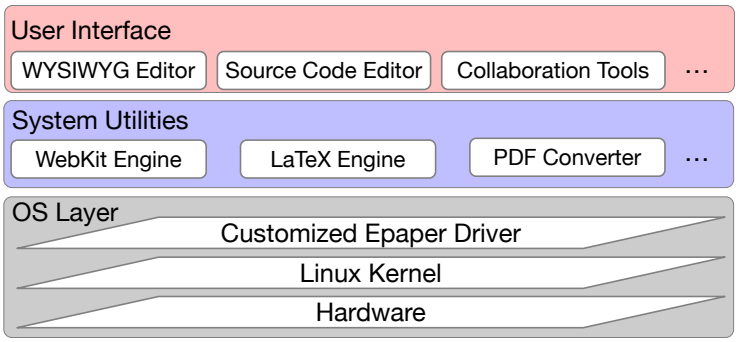

Fig. 2. System Architecture

Note that most off-the-shelf e-paper readers simply adopt Android OSs, which we argue is not an ideal solution. Notably, Android has been designed to offer 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 optimized for the epaper devices with low specification from the ground up. It is thus more resource-efficient and provides a longer battery life.

4 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 and to preview the PDF on a separate viewer. The source view needs to be preserved because only in source code can arbitrary LaTeX scripting be done.

4.1 Definition for 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.

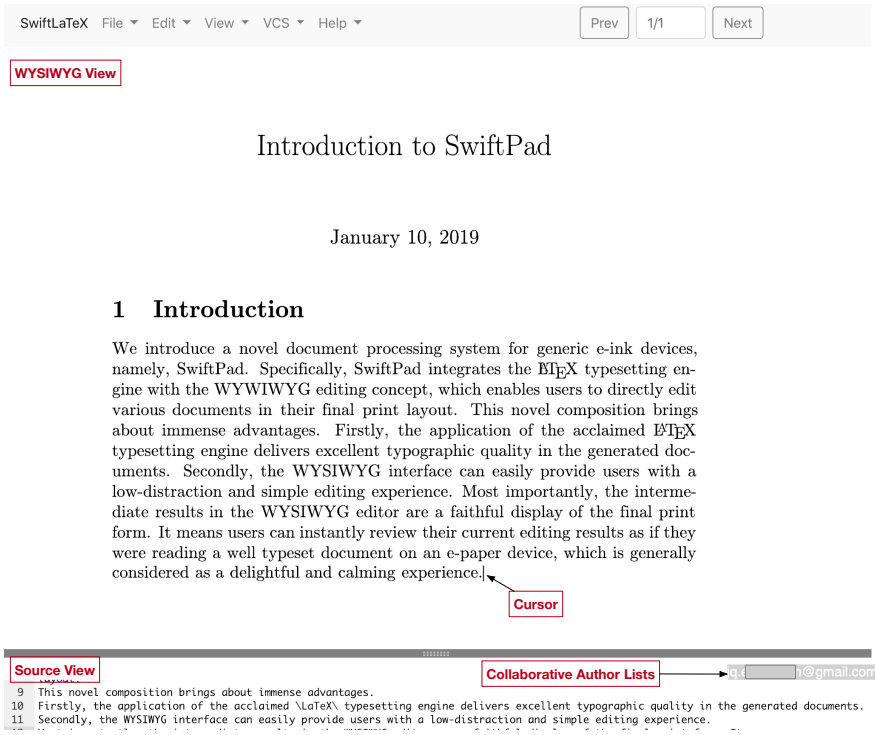

Fig. 3. User Interface of SwiftPad.

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.
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.

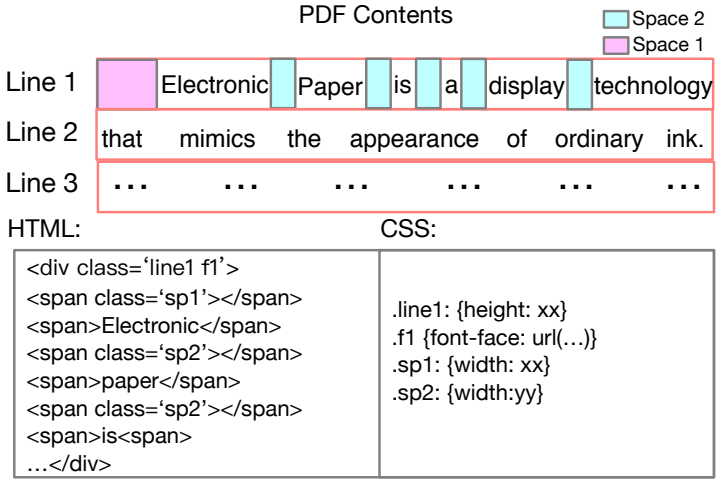

Fig. 4. Preserving Text Locations using Relative Positioning Rules

4.2 Enable Direct Editing of PDF Documents

To implement such an editor, especially the WYSIWYG view, the editor has to support not only faithful display, but also dynamic modification of PDF contents (i.e., 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 [10]) 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 less-powerful e-paper devices. Moreover, these products are generally not portable among different 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, rendering HTML pages is a relatively lightweight operation for devices with low-end specifications. Apart from that, the HTML pages are of high portability and can be displayed in various devices equipped with browsers. 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. This easily enables various 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.

Font Embedding. Fonts can be embedded in a PDF file. This prevents font substitution when readers view or print the file, in other words, ensures that readers 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 [14] and FontForge [30]. The converted fonts then can be embedded in HTML pages using a CSS rule named 'font-face' [26].

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 with the help of the CSS sprite technique [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 used in some conventional tools [20] is to convert the locations into CSS absolute position rules. However, since each text segment is now associated with a unique CSS rule, the resulting 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 instead attempt to use relative position rules to approximate the locations of each text segment. The intuition is that relative position rules can be reused by a variety of text elements so that the page size can be significantly reduced. Moreover, the relative position automatically enables text reflow functionality to a certain extent when texts are inserted or deleted. A simple example is demonstrated in Fig. 4. Specifically, we first attempt to merge text segments to text lines based on their geometric metrics. Afterwards, we position each text element from left to right for each line with the help of spacers, in other words, the relative positioning CSS rules. It can be seen that the number of generated CSS rules is tiny since they are usually referenced by multiple HTML elements. To further compress the CSS rules, it is possible to merge some rules which have nearly identical position values at the cost of faithfulness in locations.

Note that when a modification on a text element in the WYSIWYG view is detected, the editor is required to simultaneously apply the change at the corresponding source code position. The position info is obtained with the help of a LaTeX plugin introduced in the work [8]. In short, the plugin infers the source code position of each element in the PDF in character-level accuracy by patching the typesetting engine with a bookkeeping mechanism. 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 then can be retrieved by our specifically-designed editor, while being safely ignored by the conventional PDF viewers.

5 Building a Responsive Editor

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 shows a preview version of the print view. It is main achieved 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 revised version would gradually accumulate along with the user's input. To address this issue, our editor still periodically has to replace 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). In order words, users may have to wait noticeable timespan to view the faithful output.

5.1 Achieving High Responsiveness

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 among consecutive compilations. The philosophy we apply here is called check-pointing, 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.

5.2 Applying Checkpoint technology

Several runtime checkpoint implementations have been proposed. Early implementations (e.g., [11]) are mainly kernel-based. 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 [9] 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.

These issues motivate us to propose a lightweight userspace checkpoint technology for the LaTeX engines. Specifically, we utilize the dynamic hooking technique to patch the engine with the checkpoint logic. It solely checkpoints 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.
Therefore, each checkpoint operation can be swiftly carried out (usually in less than 50 ms).

To fully recover the state of the LaTeX engine, extra attention 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 predetermined 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 named 'sbrk' to extend/shrink the heap segment. To efficiently manage the heap segment, the standard C library keeps track of two internal states 1) occupied/free space in the heap segment and 2) the overall heap segment size. When restoring the heap segment, these states have to be recovered correspondingly as well. However, the restoration functionality is generally 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 for obtaining memory from the system. By using *mmap*, we are able to create a heap memory region that has been assigned a direct byte-for-byte correlation

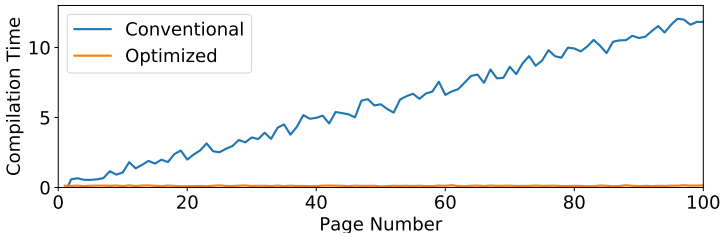

Fig. 5. Compilation time comparison between the conventional Engine and the enhanced engine.

with a filesystem object. By simply 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. Note that no operation is required to recover the internal state of the heap segment since we store all the bookkeeping info in the allocated heap region.

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*.

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 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.

To measure the response time of the LaTeX engine, we generate a variety of LaTeX sample documents using random LaTeX code snippets, whose page numbers range from 1 to 100. We then insert texts at random places of each document and conduct the measurements for 30 times. The experiments are carried out in the a single-core 1GHz ARM CPU. The results are reported 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.

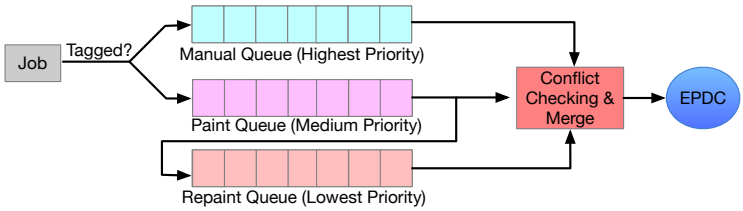

Fig. 6. Multi-Queue scheduling algorithm

6 Optimizing E-paper Driver for SwiftPad

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

6.1 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 generates significant ghost effect.

To communicate with the EPDC, the OS requires a kernel driver. However, most existing driver implementations possess an essential drawback on the update mode selection. Specifically, 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. One widely-used mechanism empowered by various e-paper devices is to provide a kernel interface to allow an application to 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

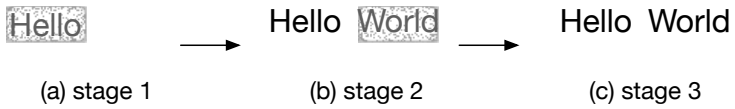

Fig. 7. A demonstrating example of the scheduling algorithm

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.

6.2 Optimizing Display Driver

In our work, we propose a novel e-paper display driver to automatically strike a balance between the display quality and response time with a heuristic scheduling algorithm. As shown in Fig 6, our algorithm maintains two separate update queues, a paint queue and a repaint queue. The paint queue stores all the incoming update jobs 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 repair the ghost effect. In other words, if no new update jobs are submitted to the driver, the high display quality of the screen contents will eventually be restored.

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 can 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, which is drawn in 2 graylevel mode. At stage two, the user types another word, the new word is still drawn in 2 graylevel mode while the previous word can be repainted using 16 graylevel mode at the same time. At the last stage, the new word is repainted and all the words are now in 16 graylevel mode.

Such an algorithm has the minor drawback that every region has to be painted twice. Clearly, it is inefficient for some applications that have a variety of static UI elements (e.g., menus and text labels). To address this issue, we extend the browser engine with a flexible interface, which enables the applications to pass down certain display hints to the driver. Specifically, for each HTML element, the developer is now able to specify the desired update mode with the help of a data attribute named *data-updatemode*. The default value for this attribute is *auto*, when our driver will do the two-queue trick. When the value is set to be 16 or 2, the element will be directly painted using the 16 graylevel mode or 2 graylevel mode respectively.

7 Evaluation

In this section, we provide detailed evaluation on SwiftPad in terms of system performance and usability.

7.1 Experiment Devices

In our experiments, we deploy SwiftPad on two popular off-the-shelf 13.3-inch devices BOOX MAX [12] and reMarkable [2]. The devices are built atop an NXP i.MX 6 Solo SoC [1], 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-reader device 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.

7.2 Usability Evaluation

In this section, we demonstrate the settings and outcomes of our evaluation on usability of SwiftPad. 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 [15]. Existing studies [17][16] 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.
- 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 approximately 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 78 (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 74 and 91 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.

Table 1. Axial coding of the participants' comments

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

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 high familiarity with the existing word processors, participants can concentrate on the actual document composition 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 atmosphere with low distraction for document composition.

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 a 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.

8 Further Discussions

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

8.1 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.

8.2 Display Scheduling Algorithm

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 performance guarantees.

8.3 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.

8.4 Collaboration

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.

9 Conclusion

In this paper, we presented SwiftPad, a document composition and editing system on e-ink readers. 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 the corresponding resuable solutions. In our user study, we have found positive feedback that the system has great potential and high efficiency for important editing tasks.

References

1. i.mx6 quad applications processors, https://goo.gl/mqtw9v
2. remarkable:paper-like tablet, https://remarkable.com/
3. 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)
4. Benedetto, S., Drai-Zerbib, V., Pedrotti, M., Tissier, G., Baccino, T.: E-readers and visual fatigue. PloS one **8**(12), e83676 (2013)
5. 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)
6. 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)

7. 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)
8. Elliott Wen, G.W.: Swiftlatex: Exploring the true wyswiyg editing for publication. DocEng (2018)
9. EMELYANOV, P.: Criu: Checkpoint/restore in userspace, july 2011
10. Foxit Reader: PDF Viewer from Foxit Software. https://goo.gl/w1918D (2019)
11. 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)
12. Intl, O.: Onyx boox electronic books, https://onyxboox.com
13. 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)
14. mupdf: Mupdf, a lightweight pdf, xps, and e-book viewer (2018), https://mupdf.com
15. 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)
16. 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)
17. Nielsen, J.: Why you only need to test with 5 users, march 19, 2000. Useit.com Alertbox **27** (2015)
18. Overleaf: Overleaf, collaborative writing and publishing (2018), https://www.overleaf.com
19. 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)
20. Poppler: Pdf rendering library based on the xpdf-3.0 code base (2018), https://poppler.freedesktop.org/
21. Research and Markets: e-Paper Display Market 2016-2020. https://goo.gl/PvVHTs (2018)
22. ShareDB: Sharedb: A database frontend for concurrent editing systems (2011), https://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), http://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), https://goo.gl/89SDo6
28. WANG, L., LIU, W.: Performance comparison between pdfs and htmls (2013), https://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)