Accumulative Display Updating for Intermittent Systems

HASHAN ROSHANTHA MENDIS, Academia Sinica, Taiwan
PI-CHENG HSIU, Academia Sinica and National Chi Nan University, Taiwan

Electrophoretic displays are ideal for self-powered systems, but currently require an uninterrupted power supply to carry out the full display update cycle. Although sensible for battery-powered devices, when directly applied to intermittently-powered systems, guaranteeing display update atomicity usually results in repeated execution until completion or can incur high hardware/software overheads, heavy programmer intervention and large energy buffering requirements to provide sufficient display update energy. This paper introduces the concept, design and implementation of accumulative display updating, which relaxes the atomicity constraints of display updating, such that the display update process can be accumulatively completed across power cycles, without the need for sufficient energy for the entire display update. To allow for process logical continuity, we track the update progress during execution and facilitate a safe display shutdown procedure to overcome physical and operability issues related to abrupt power failure. Additionally, a context-aware updating policy is proposed to handle data freshness issues, where the delay in addressing new update requests can cause the display contents to be in conflict with new data available. Experimental results on a Texas Instruments device with an integrated electrophoretic display show that, compared to atomic display updating, our design can significantly increase accurate forward progress, decrease the average response time of display updating and reduce time and energy wastage when displaying fresh data.

CCS Concepts: • Computer systems organization → Embedded and cyber-physical systems; Sensors and actuators.

Additional Key Words and Phrases: Accumulative updating, data freshness, electrophoretic displays, intermittent systems

ACM Reference Format:

1 Introduction

The Internet of Things (IoT) are gradually adopting more sustainable, self-powered node designs which harvest energy from the environment (e.g., solar, thermal, piezoelectric and RF), as opposed to traditional battery-powered solutions [6]. Ambient energy sources are inherently unstable and causes frequent power failures, thus such devices execute intermittently [19, 20]. Similar to battery-powered devices, I/O operations on intermittent systems require atomicity guarantees.
(uninterrupted power), causing them to perform expensive re-executions until completion [11] or incur additional hardware/software/programmer overhead for sufficient energy buffering [17]. Moreover, the complications of atomic I/O execution are exacerbated when the system has to perform a long latency task such as updating an electrophoretic display (EPD), a very low-power display technology that has become popular in several IoT domains [7, 14, 23], due to not requiring power to retain a rendered image (bistable characteristic). Therefore, a new display updating methodology is required for intermittently-powered systems, different from previous designs for battery-powered systems.

Intermittent systems overcome unstable power conditions by enabling *accumulative task execution* using various hardware circuits, system architecture and software designs. From a hardware standpoint, *non-volatile processors* (NVPs) have emerged as a hardware-only solution that saves a snapshot of the system state residing in volatile memory (VM) to non-volatile memory (NVM) on power loss and restores the snapshot on subsequent power resumption [6, 19], allowing execution to be resumed from the interruption point. Previous studies have examined hardware circuit level designs, such as ferroelectric flip-flops [34] and non-volatile controllers [30], to improve backup/restore operation efficiency. Other studies have explored the design space and trade-off between performance and energy consumption at the system architecture level, like processor architecture [21, 35] and power supply system [29]. Recently, much work has explored system software based techniques, including task schedulers to maximize forward progress [8], concurrency control mechanisms to allow concurrent task executions [9, 10], programming models to ensure hybrid memory data consistency [11, 20, 22], language support to handle program atomicity [17] where certain code sections cannot be executed intermittently, and design time tools to improve reliability and energy efficiency [16].

While existing research in self-powered systems mainly focuses on intermittent computation and lightweight peripheral operations such as sensing and wireless communication, *intermittently-powered displays* have received less attention, despite being valuable sub-systems in IoT designs that enable human operators to quickly and conveniently view and react to key information. Several recent self-powered IoT devices have adopted EPDs, not only due to their bistable property but also because they are lightweight, thin, flexible and have paper-like reflectivity. EPDs have been incorporated into wearables and are powered by body-heat, vibration and RF energy to display sensor readings and user-specific images [13, 23]. In such applications, the slow display update rate can significantly limit the functionality and responsiveness of the wearable. In [12], an EPD companion display obtains power and data via near field communication (NFC), and RFID powered EPDs have been used to display dynamic pricing in retail environments [14]. In such user-centric applications, the device has to delay updating the display with new data, until sufficient energy has been harvested and buffered, thus severely limiting the usability and functionality of the device. In fact, standalone EPD-based situational displays powered by standard indoor ambient lighting can take up to 25 minutes to buffer sufficient energy for a complete display refresh [15]. To the best of our knowledge, all existing self-powered EPD designs follow the *atomic I/O operation* methodology, where sufficient energy is buffered (e.g., into a supercapacitor or rechargeable battery) for a complete display update, thus requiring *display update atomicity*. They do not adapt the display update process of traditional battery-powered systems to suit intermittent systems, and thus are subject to long buffering latencies, incur expensive hardware/software overhead and require energy budget indication by the application programmer.

This paper presents the novel concept of *accumulative display updating* for intermittently-powered systems. Our design’s main goal is to *relax the atomicity constraints* of display updating, such that the EPD update process can be accumulatively completed without complete atomicity guarantees. In other words, accumulative display updating can make forward progress in updating
the EPD (i.e., the process by which a new image is effectively drawn on the display), across power cycles without re-execution, removing the need for buffering sufficient energy for a complete update and without burdening the programmer with display energy estimation. An EPD external driving controller indirectly provides the flexibility to draw on any row on the display given appropriate instructions and data, but cannot handle frequent power loss. Losing power during an ongoing display update causes the driving controller to lose state and, after power resumption, it will restart the screen drawing from the beginning. Furthermore, abrupt power loss can cause the EPD image content to distort/fade due to leakage current, which can cause long-term display contrast and performance issues. The first challenge in realizing our concept is to adapt the driving controller to offer seamless display update continuity and operability, where we need to enable state persistence across power cycles and maintain EPD physical state upon sudden power loss. Secondly, display data freshness must be maintained. The challenge arises when new data becomes available to the system at any point during an active accumulative update process, causing the display content to be in conflict (i.e., be inaccurate) with the new data that need to be shown immediately.

Our design addresses the first challenge by performing progress tracking and safe EPD shutdown. We track the minimum but adequate amount of information related to the display update progress rather than checkpointing the entire process state. The tracked progress is then used in the next power cycle upon reboot, to inform the display controller where to continue the update process from, rather than start from the beginning. To ensure that the physical state of the EPD at the point of power interruption is unaltered until power resumption and to allow for safe and effective operability of the display, the EPD is safely shutdown before power loss. We address the data freshness challenge by using a context-aware display updating policy that uses stage-level information to intuitively introduce new data into the EPD updating workflow. Compared to abruptly re-issuing a new update command with new data, our approach reduces the average update latency while minimizing the impact to the EPD’s performance and longevity. To reduce application programmer involvement, we integrate our proposed design at the driver layer, decoupled from the application layer. To validate the practicality of our concept, we conduct extensive experiments on the Texas Instruments MSP-EXP430FR5994 LaunchPad lightweight platform, and a 1.44 inch monochrome EPD module with EXT2 extension board produced by Pervasive Displays\(^1\). Compared to an atomic display updating technique that uses re-execution based atomic I/O [5, 11], our design increases the accurate forward progress of display updating by an average of 28%. When the power cycles are much shorter than the display update duration, our data freshness handling updating policy further improves the accurate forward progress by 15% and the average response time of display updating by 41%, while also reducing the time and energy wastage by 3 times.

The remainder of the paper is organized as follows. Section 2 provides background information and motivation examples to illustrate the drawbacks of atomic display updating when directly applied to intermittent systems. In Section 3, we present the rationale and details of our accumulative EPD updating design with data freshness handling capability. Evaluation results are reported in Section 4 with some concluding remarks given in Section 5.

2 Background and Motivation

In this section, we first present an overview of the architectural components and procedures of typical self-powered intermittent systems. We then describe the conventional display updating process of EPDs and point out some drawbacks associated with performing atomic display updating on intermittent systems.

\(^1\)Interested readers may view a demo at https://youtu.be/OjSKWU27cAU on YouTube.
2.1 Self-powered Intermittent Systems

A typical structure of a self-powered intermittent computing system, as shown in Figure 1, is composed of a lightweight processing unit with essential elements (e.g., CPU and memory), internal and external peripherals and an energy harvesting management unit (EHM) [16, 17, 21]. The on-chip hybrid memory subsystem consists of volatile memory (VM) which loses its state without power and slower non-volatile memory (NVM) which retains state without power. Static and dynamic random-access memories (SRAM/DRAM) are some examples of VM, whilst flash memory and more recently introduced ferroelectric random-access memory (FRAM) are examples of NVM. External peripherals such as sensors/actuators are connected via general-purpose IO (GPIO) ports and communicated using hardware communication protocols such as serial-parallel-interfaces (SPI) or inter-integrated circuit (I2C). Analog-to-digital converters (ADC) are also commonly included to translate analog electrical signals into digital for processing. These systems can also have human-machine-interfaces (HMI) such as displays, touch panels and buttons to enable human operators to view key information and interact with the devices.

The EHM unit up converts the harvested low voltage and accumulates enough energy into an energy buffer (e.g., a capacitor) to briefly operate the device. A power control circuit can be used to switch the power supply path to the system ON/OFF (e.g., [3, 16]). Power is provided when the buffered voltage reaches a pre-defined start threshold ($V_R$). When the device is powered ON, the energy in the buffer starts to rapidly discharge. Power is cut-off when the voltage drops below a pre-defined threshold ($V_O$). As a result, the system experiences power bursts and repeatedly power cycles, thus executing intermittently. Figure 1 (bottom left) illustrates this typical power ON/OFF profile. The system load and capacitor size directly influence the power ON period but smaller capacitor sizes are preferred due to their small form-factor, lower leakage current and shorter charging time (i.e., power OFF duration).

Intermittent computing systems conventionally use checkpointing, where the system state (e.g., CPU registers, VM contents etc.) is regularly backed up to NVM. Upon power resumption the system state is rolled back to the latest checkpoint to ensure forward progress without restarting the application from the beginning [28]. This concept is similar to transaction log checkpointing used in distributed database systems to maintain durability and consistency [4]. Intermittent systems can be equipped with an NVP to facilitate checkpointing at the hardware/architecture level [19, 31], but can also be equipped with a MCU with integrated NVM and the operating system/program developer takes charge of system checkpointing [3, 5, 16]. However, software-based checkpointing for intermittently-powered systems has been shown to be insufficient due to overhead concerns,
forward progress limitations if energy cost between checkpoints exceeds available energy and the challenges when trying to ensure hybrid memory data consistency [9, 11, 20]. To overcome some of these issues, recent work has looked into checkpointing only when power loss is detected, i.e., when the supply voltage goes below a certain threshold (e.g., $V_C$ in Figure 1). Note that $V_C$ must be set appropriately high to ensure a successful checkpoint and correct intermittent execution.

Checkpointing is especially poorly suited for peripherals state persistence due to two primary reasons. Firstly, certain peripherals may not provide access to their full internal state, making it not possible to capture their correct state (e.g., some digital signal processors, digital-to-analog converters and display peripherals do not provide access to certain internal write-only/temporary registers). Secondly, upon power resumption, additional steps must be taken to properly reconfigure and re-initialize the peripherals before use [1, 5], as opposed to application state restoration where the checkpointed state is simply copied from NVM back to VM. Furthermore, in particular, EPDs have a unique property where, when power is lost abruptly, the residue energy can distort or cause unwanted artifacts on the display. These issues cannot be addressed using the checkpointing paradigm and raise the need for a lightweight novel solution that does not rely on complete peripheral state accessibility, can correctly recover the peripheral execution context and retain the physical state of the display upon power loss.

2.2 EPD Technology and Display Updating

As shown in Figure 2, EPD pixels are made up of minuscule capsules with black and white pigments which are respectively negatively and positively charged. A driving waveform (voltage timing sequence) is applied on the top and bottom plates of the EPD to move the charged pigment towards/away from the top electrode plate, thus driving the white/black state of a pixel [2]. EPDs only consume power to update the display. They do not consume power to retain the image after the display power is powered OFF. The driving controller can be embedded inside the EPD’s chip-on-glass (CoG) integrated circuit or can be produced externally by an MCU. The latter offers more design flexibility and customization, whereas the former is simpler to implement. The charge pump circuitry inside the CoG driver provides the large voltage required to drive the pixels.

Figure 3 illustrates the conventional EPD global updating process [18, 24, 25]. Prior to updating the display, the new image frame buffer (FB) has to be updated and the EPD peripheral must be powered ON and initialized. This prepares the charge pump and configures the CoG driver to be able to draw a new image on the display. Lastly, after the display content has been updated, the EPD has to be powered OFF (i.e., turn off the CoG and discharge the charge pump). Unlike other display technologies such as LCD/OLED, effectively driving the EPD pixels to a new image state requires writing data to the display in the following four unique stages:

![Fig. 2. Basic overview of an electrophoretic display cell](image-url)
Stage 1 - write the inverted state of the current/old image  
Stage 2 - set all pixels to white  
Stage 3 - write the inverted state of the new image  
Stage 4 - set the pixels to the new image state

Stages 1 and 2 help to erase the current image before the new image is written out in Stages 3 and 4. The non-linear movement of electrophoretic particles in terms of speed and required energy complicates the display of different specific gray levels on neighboring pixels. Therefore, Stage 3 is used to improve the sharpness of the new image. During each stage, the same frame is written out multiple times to the display with equal number of frames \(N\) in each stage. \(N\) is set depending on the ambient temperature, as colder environments slow down the EPD pigment particles. Frame data is written out in a sequential row-wise order (top to bottom of the display). Two FBs are stored in memory and are accessed during the update process. The old/current image FB \(\text{FB1}\) is used in the erasing process (Stages 1 and 2) whilst the new image FB \(\text{FB2}\) is used by the new image activation (Stages 3 and 4). This multi-stage update procedure increases the display update latency and energy consumption (e.g., 2.7 s and 24.5 mJ respectively, for a 13 frames per stage update [24]), but eliminates the "ghosting effect", which is seen when remnants of the old image remains on top of the new image. Such a 4 stage driving process is generally implemented to guarantee clear imagery with high contrast, reduced ghosting and increased display lifespan [2, 18, 25].

2.3 Atomic Display Updating on Intermittent Systems

In conventional battery-powered systems, I/O operations (e.g., accessing a sensor, updating a display, wireless transmission) have atomicity constraints (i.e., cannot be power interrupted) that need to be taken into account. The same atomic I/O principles have also been applied to self-powered systems where if the power fails, peripherals must be re-initialized and I/O operations re-executed in the next power cycle [5, 11]. By including additional hardware/software and accurate I/O energy cost estimation by the programmer, the system can guarantee that the buffered energy is sufficient for an atomic I/O operation [17].

All current self-powered EPD solutions conform to the I/O atomicity constraints described above when performing a display update. They either repeatedly re-execute until successful completion or use sufficiently large energy buffers and custom software/hardware, to provide atomicity guarantees by energy buffering (e.g., [12, 15, 23, 24]). Figure 4(a) illustrates the case where applying repeated

---

**Fig. 3. EPD global display update driving process**

---

**Stage 1** - write the inverted state of the current/old image

**Stage 2** - set all pixels to white

**Stage 3** - write the inverted state of the new image

**Stage 4** - set the pixels to the new image state
re-execution to EPD updating does not successfully complete the process, unless the power ON duration is sufficiently long. Insufficient available energy can cause the latency-intensive EPD update process to be interrupted mid-way and restart from the beginning upon reboot, thus wasting significant amounts of time/energy and impeding forward progress of a display update. Figure 4(b) shows the case where the system spends a significant amount of time buffering enough energy before being able to start the update process after the update command has been issued. Upon each reboot, the system software has to check if the buffered energy is sufficient, relying on additional expensive hardware, large energy buffers and heavy application programmer involvement.

Whilst the concept of atomic EPD updating is sensible in the case of battery-powered systems, it brings several drawbacks when applied to intermittently-powered systems. Performing re-execution based atomic display updating does not guarantee forward progress, and situations can arise where a sufficiently long power ON time window may not become available and the display update will not complete. Providing display update atomicity guarantees via energy buffering primarily requires large energy buffers (e.g., rechargeable batteries or super-capacitors), resulting in a larger form-factor, higher design cost and inefficient energy storage, making such designs impractical especially for large-scale deployments. Also, additional hardware circuit support in the EHM unit is required to perform more flexible power control [17], and the programmer is burdened with estimating energy requirement for an EPD update, which can vary depending on the manufacturer, display size and operating temperature. Both re-execution and long energy buffering durations lead to higher display update response times, which can reduce system functionality and usability.

These observations motivate us to develop **accumulative display updating**, a new design methodology, more suited to intermittently-powered systems. We relax the atomicity constraints of EPD updating to carry forward and continue the update progress across subsequent power cycles without re-execution or requiring sufficient energy for a complete display update. As illustrated in Figure 4(c), compared to the aforementioned atomic display updating approaches, accumulative display updating provides more forward progress and shorter response times, while effectively handling data freshness issues.
3 Accumulative Display Updating

This section presents our accumulative display updating (ADU) design for intermittently-powered systems. In Section 3.1, we provide a high level overview of the ADU objective and related components of the system architecture. Section 3.2 details how continuity and operability of display updating are ensured despite frequent power loss, by progress tracking and safe EPD shutdown. In Section 3.3, we propose a context-aware display updating policy to address display data freshness, by increasing the display accuracy and reducing latency. Lastly, some implementation details and timing/energy consumption analysis are provided in Section 3.4.

Fig. 5. ADU system software architecture

3.1 Design Overview

A lightweight system with integrated EPD typically has a horizontal layered software structure, as shown in Figure 5. To update the display with new data, a user process within the application layer first updates the new image frame buffer (FB2) with new data, and issues a display update command (i.e., an API call) to the EPD driving controller in the driver layer. The EPD driver then initiates the update cycle (as described in Section 2.2) by first starting the procedure to power ON and initialize the display. Next, the multi-stage update process is executed, where the frame-level EPD data writer is used to transfer row-wise data (generally stored in the VM) to the EPD. An external peripheral communication interface (e.g., SPI/I²C) is used to send data and control signals directly to the EPD hardware module in the hardware layer. The user process waits until the update is complete, then powers OFF the display and lastly swaps the FB1/FB2 pointers in preparation for the next update.

The EPD power ON & initialization stage is non-preemptable and mandatory to operate the EPD and requires re-execution if in a rare case, power fails during this step. However, previous studies and our own have shown this step consumes only 5 to 15% of the total update duration. Most of the time/energy is spent in the actual EPD multi-stage update process.

The ADU design components are added to the driver layer (shaded elements shown in Figure 5), decoupling the ADU functionality from the application layer, to reduce the involvement of the application programmer. Display update progress tracking at runtime is jointly performed by the main state, stage-level and frame-level handlers. They act as wrappers to the EPD data writer to allow for correct display update logical continuity. Upon power resumption, the ADU handlers indicate the EPD driver to continue execution from the last completed point, instead of re-executing from the start (control flow is illustrated in Figure 6). When power loss is imminent, the voltage monitor invokes the shutdown manager to halt the ongoing update process and perform safe EPD shutdown to ensure long-term EPD operability, in an intermittent-power environment. The shutdown manager also backs up the FB contents to the NVM for persistence, and the main state handler restores the contents back to the VM upon system reboot\(^3\). The application layer is notified of update completion by the main state handler.

Enabling ADU gives rise to data freshness issues related to the correctness and timeliness of displaying the latest data on the screen with minimum delay. External data can be received in many forms, e.g., periodic sensor readings, sporadic data received wirelessly or a screen refresh event triggered arbitrarily by a user to show fresh information. For atomic updates, the new data display rate is bounded by the energy buffering duration (e.g., \([12, 15, 24]\)), but in ADU new data display requests can be received in any power cycle and should be addressed immediately. Displaying out-of-date and inaccurate data can mislead or confuse a human operator. In a conventional EPD updating flow, new data display requests are addressed sequentially, after completing the current update. This results in the display being unsynchronized with newly received data, thus reducing

\(^{3}\)Directly processing the FBs in NVM would remove the need for FB checkpointing, but would incur high NVM access latency and high energy inefficiency [16].
system usability and functionality. To address this data freshness challenge, we employ a context-aware display updating policy (denoted as ADU+F) integrated into the stage/frame-level ADU handlers, to reduce the response time of completing a new update request and to minimize the time wasted in updating the display with old data. Upon receiving a new data display request, the handlers will either seamlessly inject the latest data into the ongoing update workflow or re-issue a new update, taking steps to first erase the display before displaying new data.

3.2 Progress Tracking and Safe EPD Shutdown

3.2.1 Progress Data and Trackers. To minimize the overhead required for display update progress accumulation, we track the minimum amount of data related to the display update process in the NVM. As shown in Figure 6, only four key data elements corresponding to the hierarchical update process are tracked; namely the current state ID, stage ID, frame count and row index. Collectively these data items will be referred to as the progress data. The state ID refers to the high-level operations as shown in Figure 3, such as EPD initialization, update process, shutdown etc. The stage ID can be one of the four EPD update stages, where the frame count and row index represent the current frame being processed within a stage and the row within that frame, respectively. As an entire row is updated simultaneously, the row index is the lowest granularity of tracking required.

The ADU handlers are contained in the EPD driver layer (Figure 5) and they track the progress of the hierarchical display update process. Upon receiving an update command from the user application, the main state handler invokes the stage-level handler to begin the multi-stage update process. After completing each respective entity in the update procedure the stage/frame-level handlers update (track) the stage ID, the frame count and the row index pointers stored in NVM. When power fails during an active display update, the progress completed thus far is retained in the NVM. Figure 6 further illustrates the high-level operational flow of ADU. Upon power resumption, the main state handler determines whether update continuation is required by inspecting the high-level state, then restores the FBs from the NVM to the VM and triggers the stage-level handler. The stage and frame-level handlers fetch the relevant progress data from the NVM and pass them into the EPD data writer to resume the update process execution from the last completed point (stage/frame/row) instead of the beginning, thus ensuring continuity.

3.2.2 Safe EPD Power OFF. Sudden power loss to the EPD can cause content distortion, reduce its performance and longevity, and cause issues with accumulative display updating as we require the EPD physical state to remain static until the next power cycle. Figure 7 shows some observed cases where abrupt power loss causes leakage energy in the CoG to produce image/optical abnormalities (e.g., unwanted vertical lines and display fading/ghosting). To address this issue, the ADU voltage monitor detects power loss (supply voltage drops below $V_C$) and immediately triggers the shutdown.
Accumulative Display Updating for Intermittent Systems

manager. The shutdown manager halts the ongoing update process, backs up the FBs to NVM and begins the EPD power OFF routine, which safely discharges the residue current in the EPD and switches off the power to the EPD.

3.2.3 Minimal ADU Example. An example ADU process and power trace are illustrated in Figure 8. When the power drops below the \( V_C \) threshold, the active update process is halted, the EPD is safely shutdown and the FBs are backed up. Upon power resumption, the update continues from the previously halted position using the tracked progress. Compared with the atomic display update examples given in Figure 4, ADU can execute across multiple short power ON cycles without having to re-execute or wait until sufficient energy for a full update is buffered, ensuring forward progress and short response times. \( V_R \) and \( V_O \), shown in Figure 8, correspond to the supply power ON/OFF thresholds (EHM unit output as introduced in Section 2.1). The duration when the EPD is powered ON and display is updating is denoted as \( T_{DO} \), and the total duration for which the EPD is powered OFF is denoted as \( T_{DF} \) (including the supply power OFF duration \( T_{OFF} \)). The supply power ON/OFF and low voltage thresholds (respectively \( V_R \), \( V_O \) and \( V_C \)) should be set reasonably apart, such that sufficient time is allowed for the EPD power ON and initialization phase and the safe EPD shutdown process to complete.

3.2.4 Design Generalizability. There are two key ways our proposed design methodology can be applied to other systems and intermittent peripheral access problems. Firstly, as our ADU design resides in the driver layer, it is independent of any specific real-time operating system (RTOS), simplifying the process of integration with other systems that have an OS/kernel. Provided that an RTOS invokes the main state handler upon each power resumption and signals the stage-level handler of new data availability, the ADU state handlers can continue the display update process intermittently without manual involvement from the application/OS layer. Secondly, the low overhead principle of identifying and tracking the minimal amount of information related to the progress of a peripheral operation (in our case, display updating), can be applied to provide state persistence of other peripherals (e.g., digital signal processors/hardware accelerators). A key requirement is to identify the sufficient progress data and use only this information to recover the execution context of the peripheral operation upon power resumption.

3.3 Context-aware Display Updating Policy to Handle Data Freshness

3.3.1 Example of Data Freshness Issue. Data freshness issues in ADU arise when the system is inaccurately updating the display with old data (across multiple power cycles) and new data becomes available but is not addressed until the current update is completed. This sequential order of processing update requests results in display content to be unsynchronized with new data. Figure 9(a) illustrates the data freshness problem via an example of the conventional updating approach applied to ADU (under intermittent power). A new data value (20) arrives at time unit $t = 15$, but only starts displaying at $t = 58$, because the ongoing update (old value 10) needs to be completed first, before a new update can be issued. This results in a total response time (i.e., the end-to-end time difference between receiving the update request and fully completing the display update) of 130, and 43 time units are wasted in updating the display with old data.

Alternatively, in the process shown in Figure 9(b), the current update is first halted abruptly, then the new image frame buffer (FB2) is updated (as per the general update process shown in Figure 3) and a new update is started. Although this approach results in a shorter response time of displaying the new data, abruptly issuing a new display update, using invalid old image FB (FB1) contents and invalid EPD physical state leads to non-uniform movement of the EPD pigment particles, resulting in long-term contrast/ghosting issues.
Algorithm 1: ADU+F: context-aware data freshness handling policy

```plaintext
/* INPUTS */
S: current stage
bND: new data received signal, set in the event ISR

if bND=TRUE then
    /* Refresh FB2 with new data, then proceed update */
    if (S=1) or (S=2) then
        Update FB2={new data}
        Reset bND=FALSE
        Continue display update process
    else if (S=3) or (S=4) then
        Update FB1={WHITE}, FB2={new data}
        Reset bND=FALSE
        Restart display update process
    else
        Start display update process
    end
end
```

3.3.2 Exploiting Stage-level Context. We propose an approach (denoted ADU+F) to handle data freshness by exploiting stage-level context information. The key idea is to inject the new data into an active update pipeline without re-issuing a new display command, if the timing permits; else, to effectively restart the update process without affecting the display long-term performance. Algorithm 1 shows the proposed ADU+F data freshness handling policy. Recall that the new image frame buffer (FB2) is only accessed in Stages 3 and 4 (as described in Section 2.2 and illustrated in Figure 3). Hence, if new data is received in Stages 1 or 2, the FB2 is still unaccessed in the update process and can be re-populated with the new data and the process can continue seamlessly to display the new value, without issuing a new display update command (lines 4-7 of Algorithm 1).

This policy reduces the new value display response time and reduces time wasted in updating old data. An example of this approach is shown in Figure 9(c), where the ongoing update (value 10) is temporarily preempted when a new update request is received at $t=15$ with value 20 and the FB2 is immediately updated. This allows the new data display update to complete much earlier at $t=61$, compared with $t=145$ in the case of the conventional approach (Figure 9(a)).

When new data is received during Stages 3 or 4 of an ongoing update, we set FB1 to WHITE, update FB2 with the new data and then restart the update process (lines 8-11 of Algorithm 1). Setting FB1 to WHITE helps to construct the appropriate stage flow: (BLACK$\rightarrow$WHITE$\rightarrow$Inverse New Image$\rightarrow$New Image), such that the display is effectively erased before drawing a new image, thus reducing long-term display contrast/ghosting issues [18]. A representative example of our approach is shown in Figure 9(d), where a new update request is received at $t=15$, and immediately both FBs are refreshed as specified above and a new display update process is initiated. Alternatively, if the approach in Figure 9(b) is used in this scenario, the display would incur unwanted contrast/ghosting issues in the long term (weeks/months), as the partially completed image is not cleared before a new update is started. In comparison, our approach helps to maintain display contrast longevity and reduces ghosting, at the marginal expense of additional delay taken to set FB1 to WHITE ($\approx 3$ ms). Note that in both instances, ADU+F will drop the old data value in the ongoing update process (i.e., either display incompletely or entirely discard), to handle the new display update requests with fresh data.
The data freshness handling functionality is integrated into the stage/frame-level handlers. As shown in Figure 5, upon receiving a new display update request with fresh data (via an interrupt), the respective interrupt service routine (ISR) will signal the stage-level handler to preempt the active update process. Depending on the stage, the handler will either refresh FB1 and continue or refresh both FB1 and FB2 and restart the update process.

3.4 Implementation Details
We realized our ADU design on a lightweight platform, namely the Texas Instruments (TI) MSP-EXP430FR5994 LaunchPad [33], integrated with the 1.44 inch (128x96 pixels) eTC-Mb EPD module [26] with EXT2 mounting board [27], produced by Pervasive Displays (PDi). The EPD module is directly powered via the LaunchPad using the same power source, but the EPD can also be powered down via software. The EPD module was interfaced via SPI with a baudrate of 16 Mbps. The module offers a fast display update feature but was not used as it provides poorer contrast compared to the full EPD update cycle and ghosting when used continuously for long periods. We used 5 frames per stage drawing, which provides an acceptable contrast level at room temperature (20 °C). Two functions related to stage/frame-level EPD data transfer in the EPD driver [25] was extended to provide the ADU functionality. The stage/frame-level ADU handlers act as a wrapper around the EPD data transfer function in the EPD driver by fetching and saving the progress data to implement update state continuity. Overall, our implementation consists of about 1200 lines of C code contained in 3 new C modules and 1 EPD driver file (extended to support ADU).

3.4.1 Voltage Monitoring. The COMP_E analog comparator in the MSP430FR5994 was configured to trigger a system interrupt when the $V_C$ threshold is detected. Apart from 2 resistors used for voltage division, using COMP_E removes the need for external voltage monitoring hardware. The minimum operational voltage of the EPD is 2.3 V; hence, $V_C=2.7$ V was selected to give sufficient time for safe EPD shutdown and FB checkpointing. Furthermore, the EPD is only powered ON if the voltage goes above the safe $V_C$ threshold.

3.4.2 Overhead. We reserve 8 bytes in the NVM for progress data (four 16 bit integers) and 3072 bytes for backing up the 2 frame buffers. We track only 4 data elements in NVM at runtime, related to stage/frame/row-level completion, resulting in $\approx 2.5$ ms total tracking time spent per display update; thus, the overhead of our design is negligible. The FB backup and restore respectively take 1.9 ms and 3.7 ms, which is significantly lower than a complete EPD update operation ($\approx 2360$ ms).

3.4.3 Timing. We use a 16 MHz system clock in our implementation, resulting in $\approx 2.36$ s to complete the full EPD update process. The frame buffer updating requires 12 ms. The EPD power ON and initialization process takes 345 ms, which sets the minimum requirement of the system power ON duration. Each update stage takes on average 375 ms to complete (5 frames per stage with an average of 76 ms per frame). The EPD power OFF procedure takes 515 ms, which is taken into account when we set a suitable value for $V_C$ (low voltage threshold).

3.4.4 Energy usage. Based on the aforementioned system configuration, we obtained energy consumption measurements of EPD updating under a stable power supply. The energy measurements were obtained using the EnergyTrace software [32], also produced by Texas Instruments. The full EPD display update cycle with the integrated ADU functionality consumes $\approx 45.5$ mJ with the majority of the energy consumed by the multi-stage update process (on average 7.3 mJ per stage). The EPD power ON and initialization and the EPD power OFF processes consume 9.6 mJ and 6.4 mJ, respectively. Updating the FB consumes 0.01 mJ and the FB backup/restore procedures consume 0.02 mJ each. The ADU/ADU+F functionality requires $\approx 4.5\%$ additional energy overhead.
when the power is stable. However, as shown in the experimental evaluations (Section 4), ADU and ADU+F significantly outperform a conventional atomic display updating approach in terms of energy utilization when the power supply is intermittent.

4 Performance Evaluation

4.1 Experimental Setup

To evaluate and better understand the performance of the proposed ADU design with data freshness handling capability, we conducted a series of experiments on the aforementioned TI MSP-EXP430FR5994 Launchpad platform attached with the PDi 1.44 inch eTC-Mb EPD module. Table 1 details the experimental specifications and the experiment environment is shown in Figure 10. As shown in Figure 10 (lower right), we use a second MSP-EXP430FR5994 device physically connected via GPIO to the primary device with EPD (Figure 10 - lower left) to produce and trigger display updates with new data. To emulate execution patterns of intermittent systems, we used a Keithley 2280S DC programmable power supply unit (Figure 10 - upper left) to power the platform with various power traces. Both the primary and secondary devices in the experimental platform were powered by the same power source. Debug and measurement logging are performed via a laptop (Figure 10 - upper right) connected to the TI Launchpad platform.

Typical IoT devices with displays can receive new data (generally numerical values, e.g., temperature, humidity, etc.) from sensors or wirelessly from a remote server, or they can even produce a display refresh as a response to a user input. To closely represent such a practical scenario and to carry out controlled and reproducible experiments, we setup the secondary device (data producer) to send a 16-bit positive integer value to the primary device, which then attempts to display the numerical value on the EPD. To avoid overloading the system with many overlapping new data display requests, the data producer was configured to only trigger new data at most once per power cycle at any point during the power ON period. The probability at which a new data display request will be generated in the current power cycle is referred to as data input rate, and we evaluate the efficacy of our design to handle data freshness at 3 increasing data input rates (low=25%, medium=50% and high=75%).

Fig. 10. Experiment environment
Table 1. Specification of the experimental platform

<table>
<thead>
<tr>
<th>Intermittent Computing System specification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU</td>
<td>16-bit RISC TI-EXP430FR5994</td>
</tr>
<tr>
<td>System clock</td>
<td>16 MHz</td>
</tr>
<tr>
<td>Memory</td>
<td>8KB SRAM, 256 KB FRAM</td>
</tr>
<tr>
<td>EPD Module</td>
<td>1.44 inch eTC-Mb (128x96) &amp; EXT2 extension board</td>
</tr>
<tr>
<td>Voltage threshold settings</td>
<td></td>
</tr>
<tr>
<td>Power ON ($V_R$)</td>
<td>3.3 V</td>
</tr>
<tr>
<td>EPD Safe shutdown ($V_C$)</td>
<td>2.7 V</td>
</tr>
<tr>
<td>Power cut-off ($V_O$)</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Power trace settings</td>
<td></td>
</tr>
<tr>
<td>EPD Power ON duration ($T_{DO}$)</td>
<td>1, 2, 4 (seconds) denoted as pow1, pow2, pow4 respectively</td>
</tr>
<tr>
<td>EPD Power ON/OFF duty cycle</td>
<td>50%</td>
</tr>
</tbody>
</table>

Three step-wise power traces of 800 seconds with a periodic power ON/OFF profile were manufactured as shown in Figure 8 to emulate different patterns of intermittent executions while making the experiments reproducible (i.e., sufficient to mitigate experimental variances). The supply voltage was configured to vary step-wise between 3.3 V, 1.8 V (min. operational voltage of the platform) and 0V, resulting in periodic power failures and resumptions. The 3 power traces (denoted as pow1, pow2 and pow4) have periodic EPD power ON durations ($T_{DO}$) of 1, 2 and 4 seconds, respectively representing short, medium and long power pulses. We maintain a 50% duty cycle between system power ON/OFF durations. All three traces have equal amounts of total power ON time with 200, 100 and 50 power cycles for the pow1, pow2 and pow4 traces, respectively. Note that power failures are assumed to be unpredictable, although we used power traces with regular patterns in the experiments, for ease of experimental analysis and reproducibility.

Our ADU+F design is compared against a baseline display updating technique (termed DU), which requires atomicity guarantees and repeatedly retries the update process from the beginning until successful completion (similar to re-execution based atomic I/O in [5, 11]). DU has the same hardware as ADU, but does not contain the proposed accumulative display updating functionality. Functionally, DU and energy-buffered atomic I/O approaches such as in [12, 17] have similar atomic constraints, but the latter requires additional hardware/software and accurate energy estimation to operate; therefore, we evaluate against the former. To evaluate the advantage of data freshness handling, we also compare against a basic version of accumulative display updating (ADU), which does not contain the stage-level context aware updating policy proposed in Section 3.3. To ensure correct and safe EPD operation, all evaluated techniques only update the display when the supply voltage is between 3.3 V and 2.4 V.

We use accurate forward progress and display update response time as primary metrics to measure the effectiveness of our approach, both directly perceived by the user. A display update request with new data received by the system can be completed accurately, inaccurately, dropped/incompleted or retried. We define accurate forward progress as when the system is able to complete a display update with the displayed data (and respective FB2 contents) accurately matching the newest data received by the system, and inversely a completed update is inaccurate when the display shows old data. Often, displaying old data is not crucial (e.g., obsolete sensor readings) and it can instead be logged/aggregated. Therefore, an update request can be dropped (discarded) or become incomplete to handle a newly received request. In the case of DU, an update request can be retried (re-executed) when the process did not complete successfully in the previous power cycle. We define display update response time as the end-to-end time difference between receiving the update request and fully completing the display update (including the duration when the system has lost power). To
provide display *data freshness*, an updating technique should have a high level of accurate forward progress as well as low response time. We measure the *time wasted*, which is the time spent on updating the display with obsolete data instead of potentially using that time to update the display with new data. This also gives insight into the cause behind longer response times and inaccuracy. In addition to the time wasted, we also evaluate the corresponding *energy wastage* related to updating the display with old data.

### 4.2 Experimental Results

![Figure 11: Accurate forward progress achieved by display update techniques](image)

#### 4.2.1 Accurate Forward Progress

Figure 11 shows the level of accurate forward progress achieved by the different evaluated techniques (i.e., DU, ADU and ADU+F) across increasing data input rates (low=25%, medium=50% and high=75%) and varying EPD power ON durations (short: pow1 = 1 s, medium: pow2 = 2 s and long: pow4 = 4 s power pulses). Recall from Section 4.1 that a display update request can be accurately/inaccurately completed, dropped or retried and the number of updates accurately completed denotes the level of accurate forward progress. We can see that DU is unable to fully complete any display updates in short/medium power ON conditions, as it requires atomic power supply guarantees for the full duration of the display update (≈2.36 s) and will retry in the next power cycle. This results in a high number of retries (re-executions) in short and medium power pulses. DU can still fail to complete an update in long power ON conditions (e.g., pow4), if the update request is received towards the end of the power cycle, leaving insufficient time to complete the update. In all cases, ADU shows a higher amount of accurate forward progress than DU, as it relaxes the display update atomicity constraints.

The system can be actively processing one update request and at the same time receive another with new data, resulting in a display update request *overlap*. The amount of overlap is proportional to the data input rate and inversely proportional to the power ON duration. Under short power ON/high data input rate conditions (i.e., higher overlap), the accurate forward progress of ADU worsens, as ADU processes the ongoing update until completion before sequentially handling new requests. In such cases, ADU+F significantly improves the accurate forward progress of ADU by dropping old updates to immediately handle new requests with fresher data. When the power ON duration is long, the forward progress accuracy of ADU+F is equivalent to ADU, due to very few overlapping requests. Given a long power ON with low data input rate condition, both ADU and ADU+F achieve accurate forward progress of all handled update requests as there is no overlap. Overall, the results confirm that, unlike DU, by relaxing the atomicity constraints, ADU is able to
complete display updates and make accurate forward progress even in short power ON conditions and provides a 25 to 31% improvement even when the power ON duration is relatively long. In situations with more update request overlaps, ADU can effectively further improve the accurate forward progress by 8 to 22%, depending on the amount of overlap.

4.2.2 Response Time. Figure 12 shows the average end-to-end response time of completing display updates under different power ON durations and data input rates, for all 3 evaluated update techniques. The percentage values shown in the figure represent the relative response time increase against ADU+F. As explained in Section 4.2.1, DU is unable to complete any updates in the short/medium power ON durations; hence, the average response time of DU is shown as infinite. Under long power ON durations (i.e., pow4), ADU has a marginally lower average response time (∼4-5% lower) compared to DU, because in such scenarios the number of updates that require re-execution by DU is low. However, we observed that even in long power pulse conditions, if the update request is received towards the end of the power cycle, re-execution overhead can result in DU having a significantly higher maximum response time than ADU. In all cases, ADU+F has a lower average response time than ADU and DU. ADU+F shows higher improvement compared to ADU in shorter power ON durations, as ADU can spend several short power cycles to complete a single old update request before proceeding to handle a new request, which leads to longer update response times. When there is no display update overlap (i.e., in the pow4 and data input rate = 25% case), the average response time of ADU and ADU+F is similar. In summary, ADU+F can significantly reduce the average response time for display updating compared to ADU, resulting in an improvement of 28.4 to 47.4% in short power ON periods and of 0.1 to 7.7% in long power ON periods, depending on the data input rate.

4.2.3 Time and Energy Wasted. The time wasted in updating the display with obsolete data when new data is already available directly impacts the response time and accuracy of display updating. Time wastage also directly reflects inefficient utilization of the available energy and ideally should be minimized such that the latest data can be displayed with minimum latency and energy consumption. Table 2 shows the total time wasted in an experimental run, by each of the update techniques in the different power and data input rate conditions. The values in brackets show the comparative difference between the respective DU/ADU approaches compared to ADU+F. When the power ON duration is short (i.e., pow1), DU shows a lower total time wastage than ADU. In all other cases, ADU has significantly lower time wastage than DU because, unlike ADU which processes
old update requests until completion, DU picks up any new available data during re-execution of a display update in the next power cycle. Therefore, when power cycles are short, DU has a higher chance of addressing new update requests, thus reducing the time wasted on old updates. However, recall that DU does not successfully complete any updates in the short/medium power ON conditions (Figure 11). All evaluated techniques show a higher amount of total wasted time when the data input rate is high because higher amount of update requests are handled. Although ADU+F shows significantly lower time wastage compared to DU and ADU, a small amount of time is still wasted because new data availability checks are only performed at the stage/frame-level. Therefore, in the worst-case, ADU+F may completely process an old frame before addressing new data. If a new update request was received in Stage 3 or 4, ADU+F will restart a new update process, which may require several power cycles to complete, leading to relatively longer response times than if the request was received in Stages 1 or 2. Therefore, the total time wastage reductions given by ADU+F are much larger than the average response time improvement seen in Figure 12. ADU+F effectively wastes very little time processing ongoing updates to quickly address new update requests. Depending on the amount of overlap, ADU+F significantly reduces the total time wasted by 2.7 to 32.5 times compared to DU and 7.5 to 17.6 times compared to ADU.

Table 3 shows the respective total energy wasted by updating the display with old data for each of the evaluated techniques. These results were derived by combining the stage-level energy and timing analysis (provided in Sections 3.4.4 and 3.4.3), with the total time wasted measurements given in Table 2. In all cases, ADU+F outperforms DU and ADU, showing that ADU+F utilizes energy efficiently by minimizing the amount of time spent on processing old display update requests. The energy wastage is directly related to the amount of time wasted (Table 2). Therefore, the comparative differences between ADU+F and the other approaches show a similar trend. When DU receives new data towards the end of a power cycle, it will attempt to display the data but will fail and retry again in the next power cycle, leading to higher energy wastage in long power pulses than ADU. On the other hand, the improvement of ADU+F over ADU and DU is generally higher in medium to long power cycles as ADU+F has sufficient time to react to new display requests before encountering a power outage. To sum up, under medium power cycle conditions, ADU+F can reduce the energy wastage by 16.5 to 30.8 times compared to DU and 9.5 to 17.7 times compared to ADU.
Table 3. Total energy wasted updating the display with old data (mJ)

<table>
<thead>
<tr>
<th>Data input rate = 25%</th>
<th>DU (vs.)</th>
<th>ADU (vs.)</th>
<th>ADU+F (vs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow1</td>
<td>136.60 (3.3x)</td>
<td>448.10 (10.8x)</td>
<td>41.58 (10.8x)</td>
</tr>
<tr>
<td>pow2</td>
<td>238.27 (26.0x)</td>
<td>86.96 (9.5x)</td>
<td>9.18 (9.5x)</td>
</tr>
<tr>
<td>pow4</td>
<td>5.89 -</td>
<td>0.00 -</td>
<td>0.00 -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data input rate = 50%</th>
<th>DU (vs.)</th>
<th>ADU (vs.)</th>
<th>ADU+F (vs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow1</td>
<td>270.14 (3.4x)</td>
<td>1273.34 (16.2x)</td>
<td>78.80 (16.2x)</td>
</tr>
<tr>
<td>pow2</td>
<td>520.72 (30.8x)</td>
<td>298.88 (17.7x)</td>
<td>16.88 (17.7x)</td>
</tr>
<tr>
<td>pow4</td>
<td>65.99 (21.4x)</td>
<td>29.71 (9.6x)</td>
<td>3.08 (9.6x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data input rate = 75%</th>
<th>DU (vs.)</th>
<th>ADU (vs.)</th>
<th>ADU+F (vs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow1</td>
<td>420.32 (2.5x)</td>
<td>1534.82 (9.2x)</td>
<td>166.48 (9.2x)</td>
</tr>
<tr>
<td>pow2</td>
<td>838.09 (16.5x)</td>
<td>700.34 (13.8x)</td>
<td>50.86 (13.8x)</td>
</tr>
<tr>
<td>pow4</td>
<td>93.65 (30.4x)</td>
<td>23.54 (7.6x)</td>
<td>3.09 (7.6x)</td>
</tr>
</tbody>
</table>

5 Concluding Remarks

This paper presents the concept of accumulative display updating for intermittently-powered systems. Unlike existing self-powered EPD updating techniques, our design relaxes the atomicticy constraints to allow the EPD update to operate across multiple power cycles without the need for re-execution or energy buffering for a full update. To realize this concept, we track the minimum amount of update progress information at the driver layer, to enable continuity after power resumption rather than start the process from the beginning, and we also safely shutdown the EPD before power failure to maintain the EPD physical state and long-term operability. To address data freshness issues that arise from enabling accumulative display updating, we propose a stage-level context-aware updating policy which aims to show the latest data with low latency. We implemented the proposed design and conducted experiments on the Texas Instruments MSP-EXP430FR5994 LaunchPad with an 1.44 inch EPD by Pervasive Displays Inc. Compared to an atomic display updating approach that uses re-execution based atomic I/O operation [5, 11], accumulative display updating makes accurate forward progress even in power cycles shorter than the complete update duration (e.g., smaller energy buffers or unstable power sources with frequent power loss can produce short power pulses). Moreover, our ADU+F update policy efficiently handles data freshness to further improve accurate forward progress by 8 to 22% and reduce the average response time by 5.8 to 47.4%, depending on the amount of update request overlap experienced by the system. The improvements by ADU+F are primarily due to a 2.7 to 32.5 times reduction in time wastage on old update requests, which subsequently also leads to a 2.5 to 30.8 times reduction in energy wastage, depending on the amount of update request overlap. Future research will seek to further improve the latency of ADU+F by exploring accumulative partial screen updating and adapting update process characteristics, as well as taking environmental conditions into account.

References


