Flikker: Saving DRAM Refresh-power through Critical Data Partitioning

Song Liu
Northwestern University
sli646@eecs.northwestern.edu

Karthik Pattabiraman
University of British Columbia
karthikp@ece.ubc.ca

Thomas Moscibroda Benjamin G. Zorn
Microsoft Research
{moscitho, zorn}@microsoft.com

Abstract

Energy has become a first-class design constraint in computer systems. Memory is a significant contributor to total system power. This paper introduces *Flikker*, an application-level technique to reduce refresh power in DRAM memories. Flikker enables developers to specify critical and non-critical data in programs and the runtime system allocates this data in separate parts of memory. The portion of memory containing critical data is refreshed at the regular refresh-rate, while the portion containing non-critical data is refreshed at substantially lower rates. This partitioning saves energy at the cost of a modest increase in data corruption in the non-critical data. Flikker thus exposes and leverages an interesting trade-off between energy consumption and hardware correctness. We show that many applications are naturally tolerant to errors in the non-critical data, and in the vast majority of cases, the errors have little or no impact on the application's final outcome. We also find that Flikker can save between 20-25% of the power consumed by the memory sub-system in a mobile device, with negligible impact on application performance. Flikker is implemented almost entirely in software, and requires only modest changes to the hardware.

Categories and Subject Descriptors B.3.4 [*Hardware*]: Memory Structures-Reliability, Testing, and Fault-Tolerance; C.0 [*Computer Systems Organization*]: General-Hardware/software interfaces

General Terms Design, Management, Reliability

Keywords Power-savings, DRAM refresh, soft errors, critical data, allocation

1. Introduction

Energy has become a first-class design constraint in many computer systems, particularly in mobile devices, clusters, and server-farms [5, 11, 27]. In a mobile phone, saving energy can extend battery life and enhance mobility. Recently, mobile phones have morphed into general-purpose computing platforms, often called smartphones. Smartphones are typically used in short-bursts over extended periods of time [21], i.e., they are idle most of the time. Nonetheless, they are "always-on" as users expect to resume their applications in the state they were last used. Hence, even when the phone is not being used, application state is stored in the phone's memory to maintain responsiveness. This wastes power because Dynamic Random Access Memories (DRAMs) leak charge and need to be refreshed periodically, or else they will lose data.

Memory is a major part of overall system power in smartphones. Measures of DRAM power as a fraction of overall power range from 5-30% [6, 14] and depend on the application model, operating system, and underlying hardware. Some smartphone application programming models, such as Android's, emphasize reducing application DRAM usage in idle mode [30]. Further, the memory capacity of smartphones has been steadily increasing and, as a result, memory power consumption will be even more important in the future. Memory consumes power both when the device is active (active mode) and when it is suspended (standby mode). In standby mode, the refresh operation is the dominant consumer of power, and hence we focus on reducing refresh power in this paper.

This paper proposes *Flikker*,[1] a software technique to save energy by reducing refresh power in DRAMs. DRAM manufacturers typically set the refresh rate to be higher than the leakage rate of the fastest-leaking memory cells. However, studies have shown that the leakage distribution of memory-cells follows an exponential distribution [22], with a small fraction of the cells having significantly higher leakage rates than other cells. Hence, the vast majority of the cells will retain their values even if the refresh rate of the memory chip is significantly reduced. Flikker leverages this observation to obtain power-reduction in DRAM memories at the cost of knowingly introducing a modest number of errors in application data.

Typical smartphone applications include games, audio/video processing and productivity tasks such as email and web-browsing. These applications are insensitive to errors in all but a small portion of their data. We call such data *critical data*, as it is important for the overall correctness of the application [7, 8, 32]. For example, in a video processing application, the data-structure containing the list of frames is more important than the output buffer to which frames are rendered (as the human eye is tolerant to mild disruptions in a frame). Therefore, this data structure would be considered as critical data.

Flikker enables the programmer to distinguish between critical and non-critical data in applications. At runtime, Flikker allocates the critical and non-critical data in separate memory pages and reduces the refresh rate for pages containing non-critical data at the cost of increasing the number of errors in these pages. Pages containing critical data are refreshed at the regular rate and are hence free of errors. This differentiated allocation strategy enables Flikker to achieve power savings with only marginal degradation of the application's reliability.

[1] CRT monitors occasionally exhibited flickering, i.e., loss of resolution, when their refresh rates were lowered—hence the name.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

Our approach in Flikker fundamentally differs from existing solutions for saving energy in low-power systems. In these solutions, energy reduction is achieved by appropriately trading-off performance metrics, such as throughput/latency, Quality-of-Service (QoS), or user response time, e.g. [17, 37, 39]. In contrast, our approach explores a largeley unexplored trade-off in system design, namely *trading off energy consumption for data integrity at the application level*. By intentionally lowering hardware correctness in an application-aware manner, we show that it is possible to achieve significant power-savings at the cost of a negligible reduction in application reliability.

To the best of our knowledge, Flikker is the first software technique to intentionally introduce hardware errors for memory power-savings based on the characteristics of the application. While we focus on mobile applications in this paper, we believe that Flikker approach can also be applied to data-center applications (Section 9).

Aspects of Flikker make it appealing for use in practice. First, Flikker allows programmers to control what errors are exposed to the applications, and hence explicitly specify the trade-off between power consumption and reliability. Programmers can define what parts of the application are subject to errors, and take appropriate measures to handle the introduced errors. Second, Flikker requires only minor changes to the hardware in the form of interfaces to expose refresh rate controls to the software. Current mobile DRAMs already allow the software to specify how much of the memory should be refreshed (Partial Array Self-Refresh (PASR) [18]), and we show that it is straightforward to enhance the PASR architecture to refresh different portions of the memory at different rates. Finally, legacy applications can work unmodified with *Flikker*, as it can be selectively enabled or disabled on demand. Hence, Flikker can be incrementally deployed on new applications.

We have evaluated Flikker both using analytical and experimental methods on five diverse applications representative of mobile workloads. We find that Flikker can save between 20% to 25% of the total DRAM power in mobile applications, with negligible degradation in reliability and performance (less than 1%). Based on previous study of DRAM power as a fraction of total power consumption, this 20-25% corresponds to 1% of total power savings in a smartphone [6]. We also find that the effort required to deploy Flikker is modest (less than half-a-day per application) for the applications considered in the paper.

2. Flikker: Design Overview

We start with a brief overview of Flikker. Flikker requires a modest set of simple changes to both hardware and software. For the *hardware* portion, Flikker enhances existing DRAM architectures that allow for a partial refresh of DRAM memory by allowing different *refresh rates* for different sections in memory. For the *software* portion, Flikker (1) introduces a new programming language construct that allows application programmers to mark non-critical data, and (2) provides OS and runtime support to allocate the data to its corresponding portion in the DRAM memory. The hardware and software components of Flikker are explained in detail in Sections 3 and 4, respectively.

The changes required by Flikker are justified for several reasons. First, the software changes required to deploy Flikker can be easily made by application developers, who already distinguish between critical and non-critical data implicitly (see Section 4). Further, hardware is becoming less reliable due to trends in silicon manufacturing, and hence it is important to write software that can degrade gracefully in the presence of hardware faults [4]. The software changes to partition the application into critical and non-critical portions can also provide resilience against naturally occurring hardware faults.

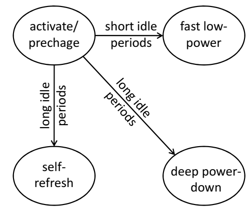


Figure 1. Simplified Diagram of DRAM Operating States.

Secondly, small hardware changes are not an inherent showstopper in terms of practical deployment, given that mobile phone architectures (both hardware and software) are still in a state of flux, with new hardware and software models being developed and put to market regularly. For example, the Partial-Array Self-Refresh Mode (PASR) mode on which we base Flikker was a relatively recent introduction in mobile DRAMs (in 2002). We show in Section 3 that the hardware changes we propose are relatively minor and only involve the memory controller.

Finally, our approach allows us to explore a novel and fundamental trade-off in system design. Traditionally, the hardware/software boundary in computer systems (and specifically the hardware memory) has provided a clear abstraction layer. The hardware was assumed to provide a resilient and "correct" substrate based on which the operating system and application can run. In Flikker, we re-examine this assumption and consciously allow the DRAM to violate data integrity (to a limited degree) in order to save energy.

3. Flikker Hardware

This section presents the hardware architecture of Flikker, i.e., the Flikker DRAM (§3.2) and the impact of lowering the refresh rate on DRAM error-rates (§3.3). We present an analytical model to evaluate the power-savings of the Flikker DRAM in §3.4. This model is used to determine the best refresh rates in our system design (§3.5), and to estimate power savings in our evaluations (§5).

3.1 Background on mobile DRAMs

Mobile phones have traditionally used SRAMs (Static Random Access Memories) for memory. However, as memory capacity increases, conventional SRAM becomes more expensive, and hence, smartphones have adopted DRAM (Dynamic RAMs). A DRAM memory system consists of three major components: (1) multiple DRAM banks that store the actual data, (2) the DRAM controller (scheduler) that schedules commands to read/write data from/to the DRAM banks, and (3) DRAM address/data/command buses that connect banks and the controller. The organization into multiple banks allows memory requests issued to different banks to be serviced in parallel. Each DRAM bank has a two-dimensional structure, consisting of multiple rows and columns. The usual memory mapping is for consecutive addresses in memory to be located in consecutive columns in the same row, and consecutive memory rows to be located in different banks. The size of a row varies between 1 to 32 kilobytes in commodity DRAMs. In mobile DRAMs, the row size varies from 1 to 4 kilobyes.

Figure 1 shows a simplified diagram of DRAM operating states. DRAM could only be accessed in active states (activate/precharge). Beside active states, several low-power states are used in different system scenarios. Fast low-power states, which have good responsiveness (short wakeup time), are used in idle period during application executions. In systems with light DRAM traffic, utilizing these fast low-power states is usually a good power-performance tradeoff. On the other hand, self-refresh state and deep power-down state are used when the whole system is in standby mode. Com-

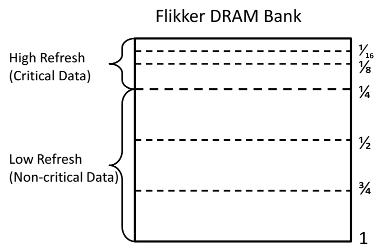


Figure 2. Flikker Bank Architecture. The DRAM bank is partitioned into two parts, the high refresh part, which contains critical data, and the low refresh part, which contains non-critical data. The high refresh / low refresh partition can be assigned at discrete locations, as shown by the dashed lines. The curly brackets on the left show a partition with 1/4 high refresh rows and 3/4 low refresh rows.

pared with fast low-power states, self-refresh and deep power-down has lower power consumption as well as longer wakeup time. Unlike the self-refresh, the deep power-down mode will stop all refresh operations and hence the DRAM will lose data in deep power-down state. In the following, we will discuss details of these low-power states.

Self-refresh: *Self-refresh* is a feature of low power mobile DRAMs in which the DRAM array is periodically refreshed even if the processor is in sleep mode. The self-refresh operation is performed by dedicated hardware on the DRAM chip. The self-refresh mode is activated only when the mobile device is in standby and the processor is put to sleep as it incurs considerable latencies to transition in and out of self-refresh mode. The Operating System (OS) needs to activate self-refresh before putting the mobile device to sleep.

Partial Array Self Refresh (PASR) is an enhancement of the self-refresh low power state [18] in which only a portion of the DRAM array is refreshed. DRAM cells that are not refreshed will lose data in PASR. In a system with PASR, before switching to self-refresh mode, the OS needs to specify the portion of the memory array to refresh. In Micron's mobile DDR SDRAM [18] with 4 banks, there are five different options for PASR, full array (4 banks), half array (2 banks), quarter array (1 bank), 1/8 array (1/2 bank), and 1/16 array (1/4 bank).

The main difference between Flikker DRAM and PASR is that instead of discarding the data in a part of the memory array, Flikker lowers the reliability of the data. As a result, Flikker is able to achieve similar levels of power savings as PASR, without compromising on the amount of memory available to applications.

Fast Low-power States: Mobile DRAMs also employ low-power states during active mode to conserve power. These low-power modes are activated even when there are brief periods of no DRAM traffic as the latency of transitioning out of these states is only around 10 nano-seconds. The transition to/from these modes is performed by the DRAM controller and does not need OS intervention. The power consumption in these low-power states is typically less than half of the DRAM power-consumption without any memory accesses.

3.2 Flikker DRAM Architecture

Figure 2 illustrates a Flikker DRAM bank. In Flikker DRAM, each bank is partitioned into two different parts, the *high refresh fault-free part* and the *low refresh faulty part*. DRAM rows in the high refresh part are refreshed at a regular refresh cycle time $T_{regular}$ (64 milliseconds or 32 milliseconds in most systems). The error rate of data in these high refresh parts is negligible (similar to data in state-of-the-art DRAM chips). On the other hand, the low refresh part is refreshed at a much lower rate (longer refresh cycle time T_{low}) and its error rate is a function of the refresh cycle time (see Section 3.3).

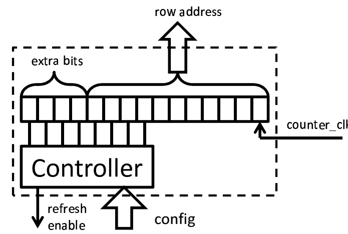


Figure 3. Self-refresh counter in the Flikker DRAM.

Mobile DRAMs use a hardware counter during the self-refresh operation to remember which row to refresh next, known as the self-refresh counter. Flikker DRAM extends this counter by a few extra bits (see Figure 3.2) in order to support two refresh rates. The Flikker self-refresh counter also has an additional "refresh enable" output. The DRAM row is refreshed only when the refresh enable bit is set to "1". A configurable controller sets different values to refresh enable bit based on higher bits of the row address and the extra bits, and thus control the refresh rate of different DRAM rows.

The number of additional bits required in the self-refresh counter is given by the ratio of T_{low} to $T_{regular}$. For example, in a system where $T_{low} = 16 \times T_{regular}$, the Flikker self-refresh counter requires 4 extra bits. The refresh enable bit is always set to "1" when the row address is a high refresh row. For low refresh rows, the refresh enable bit is set to "1" only when the extra bits has a predefined value (say "1111"). In the case of 1/8 high refresh, when the extra bits are "0000" through "1110", the refresh enable bits is only set for row addresses with highest three bits of "000". When the extra bits are "1111". The refresh enable bits is set for all row addresses. With this configuration, the low refresh rows (rows with "001" through "111" in highest bits of row address) are refreshed 16 times less frequently than the high refresh rows.

3.3 Flikker DRAM Error Rates

Previous work [3, 36] has measured DRAM error rate as a function of refresh cycle time. Bhalodia presents the per cell DRAM error rate under different temperatures and different refresh cycles [3]. Venkatesan et al. measure the percentage of DRAM rows that are free from errors with a low refresh rate [36]. Although these two measurements are at different granularity (per cell versus per row), their results are consistent with each other.

Table 1 shows the DRAM error rates used in our experiments which are based on Bhalodia's measurements [3]. The retention time of DRAM cells decrease with temperature. Therefore, under a given refresh cycle, the DRAM error rate increases with ambient temperature. We assume an operating temperature of 48°C, which is higher than the operating temperatures of most smartphones, and hence our error-rates are higher than those likely to be experienced under real conditions.

Note that the above error-rates are only a function of the refresh period and temperature. In particular, *the error-rates do not depend on the duration of low refresh mode.* This is because errors in DRAM cells are primarily caused by manufacturing variations among their retaining capacities. Thus, under a given temperature and refresh rate, a fraction of DRAM cells loses their charge, and this fraction is independent of how long the refresh rate is applied.

3.4 Flikker DRAM Power Model

We use an analytical model to estimate the power consumption of the Flikker DRAM. The model is based on real power measurements in mobile DDR DRAMs with PASR [18]. The self-refresh power consumption is calculated as follows:

Refresh Cycle [s]	Error Rate	Bit Flips per Byte
1	4.0×10^{-8}	3.2×10^{-7}
2	2.6×10^{-7}	2.1×10^{-6}
5	3.8×10^{-6}	3.0×10^{-5}
10	2.0×10^{-5}	1.6×10^{-4}
20	1.3×10^{-4}	1.0×10^{-3}

Table 1. Error rate under different refresh cycle (under 48°C, data derived from [3]).

High Refresh Size	Self-Refresh Current [mA]			
	PASR	Flikker		
		1s	10s	100s
1	0.5	0.5	0.5	0.5
3/4	0.47*	0.4719	0.4702	0.4700
1/2	0.44	0.4438	0.4404	0.4400
1/4	0.38	0.3877	0.3807	0.3801
1/8	0.35	0.3596	0.3509	0.3501
1/16	0.33	0.3409	0.3310	0.3301

Table 2. Self-refresh current in different PASR and Flikker configurations (PASR current values are from [18]). * This value is derived from linear interpolation of full array (1) and half array(1/2) cases.

$$
\begin{aligned}
P_{Flikker} =& P_{refresh} + P_{other} \\
=& P_{refresh\_low} + P_{refresh\_high} + P_{other} \\
=& \underline{P_L \times \frac{T_{regular}}{T_{low}}} + \underline{\underline{P_{refresh\_high} + P_{other}}} \qquad (1) \\
=& \underline{(P_{full} - P_{PASR}) \times \frac{T_{regular}}{T_{low}}} + \underline{\underline{P_{PASR}}}
\end{aligned}
$$

As shown in Eq. 1, $P_{Flikker}$ has two components, $P_{refresh}$, which is the power consumed by refresh operations, and P_{other}, which is the power consumed by other parts of the DRAM (e.g. the control logic) in standby. $P_{refresh}$ is proportional to the refresh rate; while P_{other} is independent of the refresh rate and is constant. Then we divide $P_{refresh}$ into $P_{refresh\_high}$ and $P_{refresh\_low}$, which correspond to the refresh power consumed by the high and low refresh parts respectively (as shown in second line of Eq. 1). We further explicate the relationship between refresh power and refresh cycle time by representing $P_{refresh\_low}$ as P_L (which is a constant) times $T_{regular}/T_{low}$ (third line in Eq. 1).

In order to evaluate $P_{Flikker}$, we consider the DRAM with PASR and DRAM with full array refreshed (i.e., regular DRAM) as two extreme cases of Flikker. We calculate P_{PASR} and P_{full} by assigning $T_{low} = \infty$ and $T_{low} = T_{regular}$ in the third line of Eq. 1. With these two extremes cases, we rewrite the third line of Eq. 1 with P_{PASR} and P_{full} (as shown in the fourth line of Eq. 1). The underlined and double-underlined parts of the third line in Eq. 1 are equal to the corresponding parts in the fourth line.

Table 2 summarizes the self-refresh current of different PASR configurations and Flikker DRAM with different refresh cycle times for the low refresh part. The self-refresh power is calculated as self-refresh current times power supply voltage (1.8V in our experiments). It is important to understand that the self-refresh power comprises the power consumed in refreshing the DRAM array, and the power consumed to control the refresh operations of the DRAM chip. The former is proportional to the refresh rate, while the latter is a constant. Therefore, the self-refresh power does not decrease linearly with the refresh rate, but decreases and saturates at about 33%.

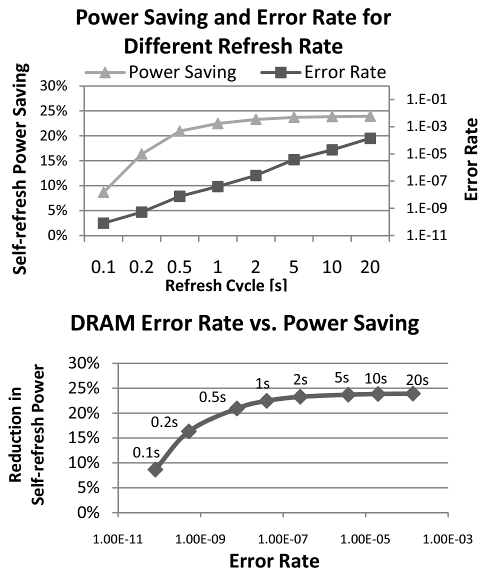


Figure 4. Error rate and power saving for different refresh cycles. The high refresh part is 1/4 of DRAM array .

3.5 Power-Reliability Trade-off

The models derived in the two previous sections are used to derive a suitable refresh rate for Flikker. Figure 4 shows the self-refresh power saving and DRAM error rate of different refresh cycles in a system with 1/4 of the memory array at the high refresh rate. In Figure 4 (top), the X-axis represents the refresh cycle time, the Y-axis on the left represents the power-savings in self-refresh mode, while the Y-axis on the right represents the error-rate on a logarithmic scale. It can be observed that the DRAM error rate increases exponentially with the DRAM refresh cycle. However, the self-refresh power saving saturates to about 25% at a refresh cycle time of about 1 second.

Increasing the refresh cycle beyond 1 second leads to significant increase in the error rates (the graph is draw to log-scale). For example, from 1 to 20 seconds, the error rate increases over 3000 times, from 4.0×10^{-8} to 1.3×10^{-4}. However, the improvement in power saving corresponding to the refresh cycle increase is small (22.5% to 23.9%). On the other hand, reducing refresh cycle time from 1 second to 0.5 seconds leads to a steep decrease in power saving. This finding is also substantiated in Figure 4 (bottom), which shows the power-savings as a function of the error-rate (in log scale). Therefore, we believe that *a refresh cycle of 1 second is near-optimal, as it achieves a desirable tradeoff between power savings and reliability*. This is the value we use in our experiments.

4. Flikker Software

In this section, we describe the changes that need to be made to software so that it can use the Flikker DRAM. Figure 5 shows the steps involved in the operation of Flikker. First, the programmer marks application data as critical or non-critical. Second, the runtime system allocates critical and non-critical data to separate pages in memory, and places the pages in separate regions of memory (i.e., high-refresh and low-refresh respectively). Third, the Operating System (OS) configures the DRAM self-refresh counter before switching to the self-refresh mode. Finally, the self-refresh controller refreshes different rows of the DRAM bank at different rates depending on the OS-specified parameters. Based on Figure 5, modifications need to be made to the application, the runtime system and the Operating System (OS).

4.1 Application Changes

Critical data is defined as any data that if corrupted, leads to a catastrophic failure of the application. It includes any data that cannot be easily recreated or regenerated and has a significant impact on the output. The concept of critical data has been used in prior work [7, 8, 32] on error recovery. However, to the best of our knowledge, our work is the first to leverage critical data in applications for power savings.

Earlier studies have shown that it is intuitive for developers to identify critical data in applications [8, 32]. This is consistent with our observations in this paper as each application considered in the paper took us less than a day to partition (including the time we spent understanding it).

Reason for ease of critical data separation: We posit that application developers make a natural distinction between critical and non-critical data. Such a distinction is important for three reasons. First, application developers typically partition their code into modules based on functionality. Some modules may be responsible for the application's core functionality and separating the data manipulated by such modules from the rest of the application's data makes it easier to reason about their correctness. Secondly, software fails due to a variety of different reasons (in production use), and application developers may provide recovery and restart mechanisms for such failures. While the operating system may provide limited support, it is typically up to the application developer to store the important state of the application periodically and to restart the application from the stored state upon a failure [7]. Finally, programs typically separate the data structures storing different parts of their input/output space. For example, a video application will likely store its non-critical video data separate from the critical meta-data describing the video file.

How to identify critical data: In Flikker, the programmer marks program variables as "critical" or "non-critical" through type-annotations in the program's source code. We assume that the default type of a variable is critical, so that we can run an unmodified (legacy) application safely. An application's memory footprint has four components, code, stack, heap, and global data. Errors in the code or the stack are likely to crash the application and hence, we place code and stack data on the critical pages. Global data and heap data, on the other hand, contain both critical and non-critical parts. For global data, the programmer uses special keywords to designate the non-critical part. This requires support from the compiler and linker, which we currently do not have (we emulate such support). For heap data, the programmer allocates non-critical objects with a custom allocator, which involves modifying malloc calls in the program where non-critical data objects are allocated.

Limitations: Failure to identify critical data correctly may lead to corrupted states of applications and/or loss of data. As a result, Flikker may not be suitable for safety critical applications, such as bio-medical devices. However, there are a large class of applications that do not require such stringent correctness guarantees. To protect data during application failures, programmers of many applications already identify and protect critical data. Specifically, some applications periodically dump critical data to persistent storage (e.g., a file), while other applications introduce redundancy in the critical data. For these applications, Flikker may not require substantial effort from programmers. However, in many other applications, the separation between critical and non-critical data may not be as obvious. In particular, applications in which the critical state it tightly intertwined with the non-critical state are not good candidates for Flikker. In such applications, there are two difficulties with deploying Flikker. One is that the programmer may miss identifying critical data, leading to corruption of critical state and failure of the application. The second difficulty is that the programmer may over-annotate critical data, thereby marking non-critical data as critical. This leads to lost opportunities for power savings, though the application's reliability is not impacted. Identifying the most effective way for the programmer to partition critical data is outside the scope of this paper and is a topic of future work.

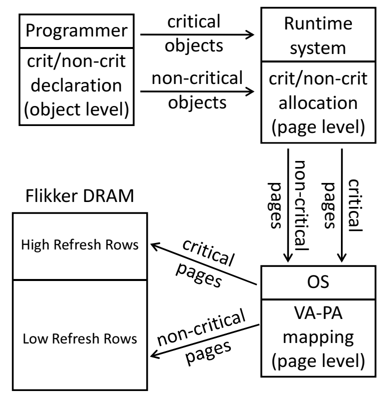


Figure 5. Flikker system diagram.

4.2 Runtime System Support

Flikker utilizes a custom allocator that allocates critical and non-critical heap data on different pages. The allocator marks pages containing non-critical data using a special bit in the page-table entry. The allocator also ensures that either all the data in a page is critical or all of it is non-critical, i.e., there is not mixing of critical and non-critical data within a page.

Ideally, both heap and global data should be partitioned into critical/non-critical parts. Our current version of Flikker does not implement partitioning of global data as this requires compiler support. However, as we will show in the experimental results (Section 6), there is strong evidence that global data has similar characteristics as heap data in terms of the relative proportion of critical to non-critical data.

4.3 Operating System Support

In a system with Flikker, the OS is responsible for managing critical and non-critical pages. A "criticality bit" is added to the page table entry of each page. This bit is set by the custom allocator when allocating any data from the page, unless the data has been designated as non-critical by the programmer. Based on the criticality bit, the OS maps critical pages to the high refresh part of the bank (top down in Figure 2), and non-critical pages to the low refresh part (bottom up in Figure 2). Before switching to the self-refresh mode, the OS configures DRAM registers that control the self-refresh controller based on the amount of critical data.

Ideally, the high refresh rate portion in the bank covers only pages containing critical data. However, this may not always be possible due to discretization in the self-refresh mode (Section 3.4). Therefore, the OS may end up placing more DRAM banks in high-refresh state than absolutely necessary, leading to wasted power. We show in Section 5 that this discretization does not significantly impact the power-savings of Flikker.

5. Experimental Setup

In this section, we present the applications and experimental methods used to evaluate Flikker. As mentioned in Section 3, Flikker requires minor changes to the hardware and hence we do not evaluate it directly on a mobile device. Instead, we use hardware simulation based on memory traces from real applications to evaluate the performance overheads and active power consumption of Flikker. Further, we evaluate the error-resilience of these applications by injecting representative faults in the applications' memory with an error-rate corresponding to the expected rate of errors from Section 3.3. The fault-injection experiments are carried out during the execution of each application (to completion) on a real system. We inject thousands of faults in each application and observe their final outputs in order to evaluate the reliability degradation due to Flikker. Finally, we evaluate the total power consumed by combining the active power consumption with the idle power consumption from the analytical model.

5.1 Selected Applications

We choose a diverse range of applications to evaluate Flikker, based on typical application categories for smartphones. Each application's output is evaluated using custom metrics based on its characteristics. For each application, we describe the application, the choice of critical data and the metrics for evaluating its output.

mpeg2: Multimedia applications are important for smartphones. Many multimedia applications utilize lossy compression / decompression algorithms, which are naturally error resilient. We select mpeg2 decoder from MediaBench [26] to represent multimedia applications. We mark the input file pointer, video information, and output file name as critical because corrupting these objects will cause unrecoverable failures in the application. We use the Signal-to-Noise-Ratio (SNR) to evaluate the output of the mpeg2 application, which is a commonly-used measure of video/audio fidelity in multimedia applications.

c4: Computer games constitute an important class of smartphone applications. Games usually have a save mechanism to store their state to files. Since the game can be recovered entirely from the saved files, the data stored to these files constitute the critical data. We select c4 [13] (known as connect 4 or four-in-a-row), which is a turn-based game similar to chess. c4 stores its moves in a heap-allocated array, which we mark as critical. We modify c4 to save its moves at the end of each game, and use the saved moves to check its output.

rayshade: Rayshade [24] is an extensible system for creating ray-traced images. Rayshade represents a growing class of mobile 3D applications. In rayshade, objects that model articles in the scene are marked critical as errors in these objects impact large ranges of the output figure. As was the case with mpeg2, we use the SNR to evaluate the output of rayshade.

vpr: Optimization algorithms may be executed on mobile phones for a variety of common tasks, e.g., calculating driving directions. We select vpr from SPEC2000 [35] to represent these algorithms, as it employs a graph routing algorithm for optimizing the design of Field-Programmable Gate Arrays (FPGAs). We choose the graph data structure as critical because any error in this structure will crash the program. We evaluate the output of vpr by perfoming a byte-by-byte comparison with the fault-free outputs.

parser: Natural language parsing is used in applications such as word processing and translation. The parser application from SPEC2000 [35] is chosen to represent this class of applications. Parser translates the input file into the output file based on a dictionary. Errors in the dictionary data are likely to affect multiple lines in the output and hence the dictionary is marked critical. Similar to vpr, we evaluate the results of parser by comparing the output file with the fault-free output.

Table 3 summarizes the characteristics and evaluation metrics of these applications.

5.2 Experimental Framework

We introduce the main components of our experimental infrastructure in this section.

Memory Footprint Analyzer: We analyze the memory footprint of each application in order to calculate the proportion of critical data. This foot-print is used to calculate power-consumption in idle-mode. These measurements were performed by enhancing the Pin dynamic-instrumentation framework [29].

Application	LoC	Input	Metric
mpeg2	10,000	mei16v2.m2v	output SNR
c4	6100	N/A	saved moves
rayshade	24,200	balls.ray	output SNR
vpr	24,600	ref/test	output file
parser	11,500	ref/test	output file

Table 3. Application characterizations and the output criteria used for evaluating Flikker. (LoC = Lines of Code)

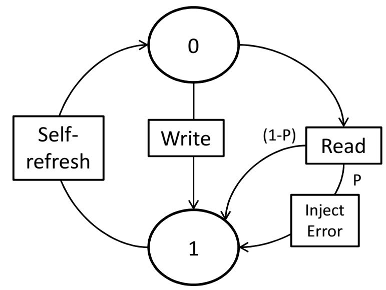


Figure 6. State transition diagram of "modified" bit in fault-injection. Error is injected to the DRAM with probability "P".

Architectural simulator: We use a cycle-accurate architectural simulator for evaluating the Flikker hardware. The simulator contains a functional front-end based on Pin [29] and a detailed memory system model. A DRAM power model based on the system-power calculator [19] is incorporated into the simulator. We do not specify the physical allocation of pages among different banks in the simulator—this is implicitly assigned depending on whether the page is critical. The simulator takes instruction traces as inputs, and produces as outputs estimates of the total power consumed and the total number of processor cycles and instructions executed in the trace. Table 4 shows the main processor and DRAM parameters used by the simulator. These parameters are chosen to model a typical smartphone with a 1GHz processor and 128 Mega-bytes of DRAM memory.

Fault-injector: We built a fault-injector based on the Pin [29] dynamic instrumentation framework. The injector starts the application and executes it for an initial period. No errors are injected during this period. Then a self-refresh period is inserted, after which errors are injected to the non-critical memory pages to emulate the effect of lowering their refresh rate. In order to keep track of the errors injected during the self-refresh period, the injector maintains a "modified" bit for each byte in the low refresh pages denoting whether this byte has been accessed after the self-refresh period. Before a low refresh byte is read, the corresponding modified bit is checked. If it is "0", meaning that the byte has not been accessed after self-refresh, a single bit is flipped in the byte with a pre-computed probability (third column of Table 1).[2] Modified bits that correspond to target bytes of memory read or write operations are set to "1" to prevent future injections into these bytes. Figure 6 shows the state transition diagram of the "modified" bit.

5.3 Experimental Methodology

We evaluate the performance overhead, power savings, and reliability degradation due to Flikker. Figure 7 demonstrates our overall evaluation methodology. The main steps are as follows:

[2] Given the low error rates in Table 1, the probability of multiple bit flips in each byte is extremely low, and are hence ignored.

Parameter	Value
Processor	single core, 1GHz
Cache	32KB IL1 and 32KB DL1, 4-way set associative, 32-byte block, 1-cycle latency
DRAM	1Gb, 4 banks, 200MHz (see [18])
Low power scheme	precharge row buffer after 100ns idle; switch to fast low power state 100ns after precharge
Cache miss delay	row-buffer-hit: 40ns, row-buffer-close: 60ns, row-buffer-conflict: 80ns

Table 4. Major architectural simulation parameters.

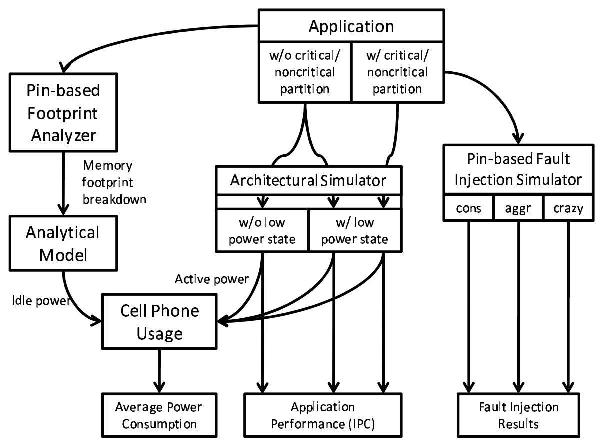


Figure 7. Evaluation Framework.

1. First partition each application's data into critical and non-critical (top box of Fig. 7).
2. Obtain the memory footprint of each application and use the analytical model to calculate the idle DRAM power consumption with and without Flikker (left portion of Fig. 7).
3. Apply architectural simulation for measuring the performance degradation and active DRAM power consumed by the application (middle portion of Fig. 7).
4. Calculate average DRAM power consumption and the total DRAM power saving achieved by Flikker (bottom left portion of Fig. 7).
5. Use fault-injection to evaluate the application's reliability under Flikker (right portion of Fig. 7).

In the following, we describe each of the above steps in detail.

Critical Data Partitioning: We modify all 5 applications to use Flikker's custom allocator for allocating heap data. Our experimental infrastructure does not allow us to partition the global data into critical and non-critical parts. To understand the impact of global data partitioning, we consider two configurations: "conservative", in which all global data is critical, and "aggressive", in which all global data is non-critical. The configurations bound the performance benefit and the reliability impact of partitioning the global data. We anticipate that partitioning global data yields a power-savings close to that of aggressive and has reliability impact close to that of the conservative configuration, provided that the critical data is a small fraction of all global data (in Section 6.4, we present experimental evidence that this is indeed the case).

In the above discussion, we assumed that stack data is placed in high-refresh state. However, in some applications, the stack data may also be amenable to being partitioned into critical and non-critical. To emulate this condition, we consider a third-configuration "crazy", where the stack and critical data are also placed in low-refresh state. Table 5 summarizes the configurations used for evaluating each application.

Configuration	High Refresh	Low Refresh
conservative	Code, Stack Crit-Heap, Global	Noncrit-Heap
aggressive	Code, Stack Crit-Heap	Global Noncrit-Heap
crazy	Code	Stack, Global Crit/Noncrit-Heap

Table 5. Configurations used to evaluate Flikker

App.	Code	Stack	Global	Crit Heap	Noncrit Heap
mpeg2	79	31	181	1	618
c4	473	21	10062	1	0
rayshade	97	10	603	2	541
vpr	114	713	4271	1739	2888
parser	88	544	1570	27	7688

Table 6. Memory footprint breakdown (number of 4kB pages).

Memory foot-print and idle-power calculator: Table 6 summarizes the memory footprint break down for code, stack, global data, critical, and non-critical heap pages. For stack and heap data, we report the maximum number of pages used during the execution. Hence, these measurements form an upper-bound on the total memory foot-print of the application.

We calculate the power consumed by the system in idle mode based on the analytical model derived in Section 3.4 and the data presented in Table 6. The refresh cycle in the low refresh portion of memory is assumed to be 1 second. This calculation is based on the results of the analytical model in Section 3.4. Further, the high refresh portions in each application are rounded up to the discrete levels in Table 2 to emulate their real-world behavior.

Architectural Simulation: We evaluate the performance and power consumption in active mode using the hardware simulator described in Section 5.2. For evaluating performance, we measure the Instructions Per Cycle (IPC) of the system, and for evaluating the power consumption, we measure the total energy consumed by each DRAM bank and divide it by the simulation time.

All 5 applications are compiled with Microsoft Visual Studio 2008. The simulations are performed with application traces consisting of 100 million instructions chosen from the approximate middle of the execution of each application. For vpr and parser, we use the SPEC ref inputs in architectural simulations, while for the other applications, we choose inputs representative of typical usage scenarios.

The main source of performance overhead due to Flikker stems from the partitioning of application data, which can potentially impact locality and bank-parallelism. Therefore, the overhead of Flikker is evaluated by considering a system that employs data partitioning (Part) with one that does not (Base). Note that the refresh rate of Flikker plays no part in the measurement of active power. In both cases, we assume that the DRAM aggressively transitions to low-power states when not in use, as mentioned in Section 3.1.

Application	Scenario	IPC	Active Power [mW]
mpeg2	Base	1.462	4.17
	Part	1.462	4.18
c4	Base	1.057	5.06
	Part	1.068	5.03
rayshade	Base	1.734	4.15
	Part	1.734	4.15
vpr	Base	1.772	4.14
	Part	1.772	4.14
parser	Base	1.694	4.17
	Part	1.695	4.16

Table 7. Performance (IPC) and Active Power Consumption of Flikker

Power-savings calculation: We assume a mobile DRAM device having a capacity of 128 megabytes, which is conservative compared to the memory capacity of current smartphones (e.g., the iPhone). [3] Most of the selected applications will use far less RAM than this space. However, in a realistic scenario, multiple applications will share the RAM space and hence it is important to account for power-savings on a per-application basis. Therefore, we compute the proportion of critical and non-critical data for the application, and scale it to the size of the entire DRAM. This allows us to emulate the multiple-application scenario while considering only one application at a time. In order to evaluate overall DRAM power reduction, we assume that the cell phone usage profile is 5% busy versus 95% in standby mode (self-refresh state) as assumed in prior work [36].

Fault-injection: The fault-injection experiments are performed using the fault-injector described in Section 5.2. Note that the inputs used for each application during fault-injection are the same as those used for performance evaluation and active-power calculation (the only exceptions are vpr and parser, where we use the SPEC test inputs for fault-injection due to the large number of trials performed). When performing the fault-injection experiments, we monitor the applications for failures, i.e., crashes and hangs. If the application does not fail, its final output is evaluated using application-specific metrics shown in Table 3. We classify the fault-injection results into three categories as follows, (1) perfect (the output is identical to an error-free execution), (2) degraded (program finishes successfully with different output), and (3) failed (program crashes or hangs).

6. Experimental Results

We now discuss the results of experiments used to evaluate the power savings, reliability and performance degradation with Flikker.

6.1 Performance & Active Power

Table 7 (column 2) shows the performance and active power consumption of the Base and Part system scenarios. Recall that Base represents the non-partitioned version of the application, while Part represents the partitioned version. The results in Table 7 show that the IPC of the Base and Part scenarios are similar for all applications (both within 1% of each other). Therefore, the performance overhead of Flikker is negligible for the applications considered. Further, Flikker does not significantly increase the active power consumption of the application. In some cases, the active power consumption is actually reduced due to the partitioning because it increases the bank-parallelism by laying out memory differently.

[3] The higher the memory capacity, the greater the power savings achieved by Flikker.

6.2 Power Reduction

Figure 8 shows the reduction in DRAM standby power for different applications and the three configurations in Table 5. Figure 9 shows the overall power reduction for different applications, which are obtained by combining the results in Figure 8 with the active power measurements in Table 7. The following trends may be observed from Figures 8 and 9.

- Both the standby and overall power consumed vary with the application and the configuration. For all applications, the crazy configuration achieves the highest power savings (25-32% standby and 20-25% overall), followed by the aggressive configuration (10-32 % standby and 9-25% overall) and finally the conservative configuration (0-25% standby and 0-17% overall).
- The aggressive configuration achieves significant power savings in all applications except vpr. This is because the applications' memory foot-print is dominated by global and non-critical data, whereas in vpr the stack, code and critical data pages constitute a sizable fraction of the total memory pages (over 35% according to Table 6). However, the crazy configuration achieves significant power savings for vpr, as the stack and critical pages are placed in the low-refresh state.
- For mpeg2, c4 and rayshade, the aggressive and crazy configurations yield identical power savings (both standby and overall) as these applications have very few stack and critical data pages.
- Among all applications in the conservative configuration, parser exhibits the maximum reduction in both standby and overall power consumption (22% and 17% respectively). This is because parser has the largest proportion of non-critical heap data among the applications considered, and this data is placed in low-refresh state in the conservative configuration.
- The power savings for the c4 application in the conservative configuration is 0% as its memory footprint is dominated by global data pages (according to Table 6), which are placed in high-refresh mode in the conservative configuration.

From Figure 8 and 9, Flikker achieves substantial DRAM power savings. The actual reduction in the overall system power consumption depends on the relative fraction of memory power to total system power. Previous work [6] shows that DRAM contributes about 4% of overall power consumption of the Openmoko Neo Freerunner (revision A6) mobile phone. In this case, Flikker would only yield 1% reduction of total system energy consumption. Nevertheless, as DRAM power is and will continue to be a significant component of computer systems, Flikker savings can be obtained across the spectrum of systems, ranging from the very small (mobile devices) to the very large (datacenters).

6.3 Fault Injection Results

In this section, we present the results of fault-injection experiments to evaluate the reliability of Flikker. We first present overall results corresponding to the error-rate for a low-refresh period of one second, which we showed represents the optimal refresh period for power-reliability trade-off in Section 3.4. We further evaluate the output degradation for each application under faults. Finally, we demonstrate the importance of protecting critical data by performing targeted fault-injections into the critical heap data.

6.3.1 Injections in both critical and non-critical data

Figure 10 shows the result of the fault-injection experiments for five applications and three configurations with an error-rate corresponding to a 1 second refresh period. Each bar in the figure represents

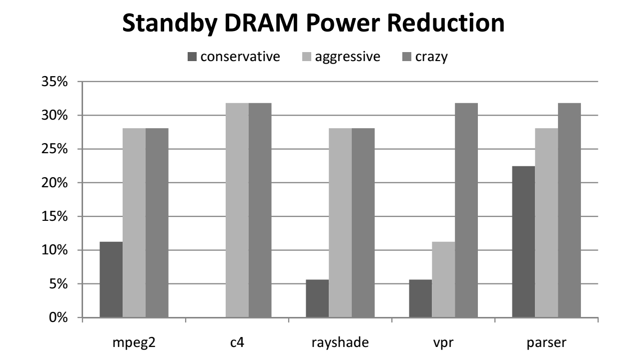


Figure 8. Standby DRAM power reduction for different applications.

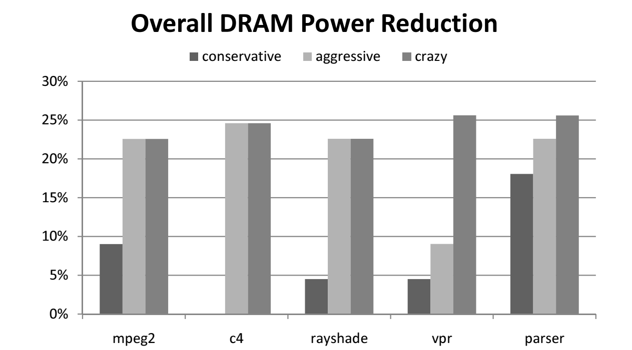


Figure 9. Overall DRAM power reduction for different applications.

the result of 1000 fault-injection trials. The results are normalized to 100% for ease of comparison.

The main results from Figure 10 are summarized as follows:

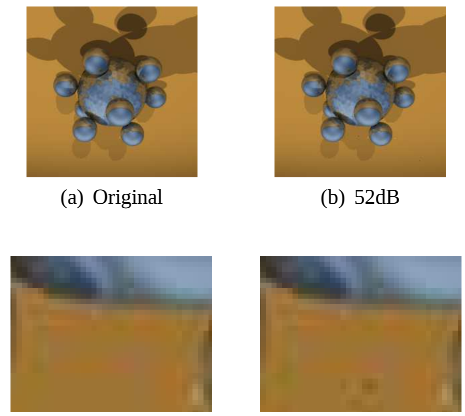
- No application exhibits failures in the conservative configuration. In fact, c4, vpr, and parser, have perfect outputs in the conservative configuration. However, mpeg2 and rayshade have a few runs with degraded results (about 33% for mpeg2 and 4% for rayshade), but as we show later in the section, the degradation is marginal. The degradation accurs because mpeg2 and rayshade maintain a large output buffer in DRAM, which is likely to accumulate errors during the self-refresh period.
- Both aggressive and crazy configurations yield worse results than conservative for all applications. The only exception is c4, which has a very small proportion of critical pages. These pages are unlikely to get corrupted given the relatively low error rate corresponding to the 1 second refresh period.
- The difference between the aggressive and crazy configurations is small, with aggressive having slightly fewer failures and degraded outputs. This is because the proportion of critical heap and stack pages is relatively small, and hence the probability of corrupting objects in these pages is very low.
- Finally, the aggressive configuration exhibits a very small number of failures across applications (except parser). This confirms our earlier intuition (see Section 5.3) that global data is likely to contain a very small proportion of critical data.

As mentioned above, the conservative configuration yields degraded outputs in about 33% of mpeg2 executions and in about 4% of rayshade executions. The aggressive and crazy configurations also yield degraded output in about 40% and 20% of mpeg2 executions and 21% and 23% of rayshade executions respectively.

To further understand the extent of output degradation, we measure the quality of the video or image using measures such as the

Configuration	mpeg2	rayshade
conservative	95	101
aggressive	88	72
crazy	88	73

Table 8. Average SNR of degraded output for mpeg2 and rayshade [dB]. Larger values indicate better output quality.



(a) Original (b) 52dB

(c) Magnified orig. (d) Magnified 52dB

Figure 11. Rayshade output figures with different SNRs.

Signal-to-Noise Ratio (SNR). Table 8 shows the average SNR measurements for the outputs averaged across all trials exhibiting degraded outputs. Note that SNR is measured in decibels (dB), a logarithmic unit of measurement. As can be seen from the table, the conservative configuration yields over 95 decibels of output quality for mpeg2 and over 100 decibels for rayshade on average. The aggressive and crazy configurations both yield SNRs of over 80 decibels for mpeg2 and over 70 decibels for rayshade.

In order to understand better the qualitative impact of output degradation in mpeg2, we take a raw video, encode it with the mpeg2 encoder, and decoded the result with the mpeg2 decoder. Compared with the original video, the final output video has an SNR of 35 decibels. This demonstrates that an SNR of 80 or above in fact represents a video of high-quality, which we believe is acceptable for a mobile smartphone with a limited display resolution.

For rayshade, we attempt to understand the output degradation by studying the rendered images. Figures 11 a and 11 b show the original image and the corresponding degraded image (with a SNR of 52 decibels). The latter is generated during a faulty execution of rayshade. These figures are shown with a scale factor of 0.25. As can be seen from the figure, it is almost impossible to tell the difference between the original image and the degraded image. However, when we magnify the images to a factor of two of the original (Figures 11 c and 11 d), small differences among the pixels become discernible. Therefore, even for a significantly degraded image with SNR considerably below 70 decibels, the differences become discernible only at high resolutions.

6.3.2 Injections into critical data only

Based on the results presented in the previous section, one may ask whether it is indeed necessary to partition applications in order to prevent errors in the non-critical data. We attempt to answer this question by performing targeted injections into the critical data. If we do not observe any failures in these experiments, then we can conclude that preventing errors in the critical data (and hence partitioning of data) is unnecessary for achieving high reliability.

In these experiments, we inject a single error into the critical data during each trial because the proportion of critical data in each

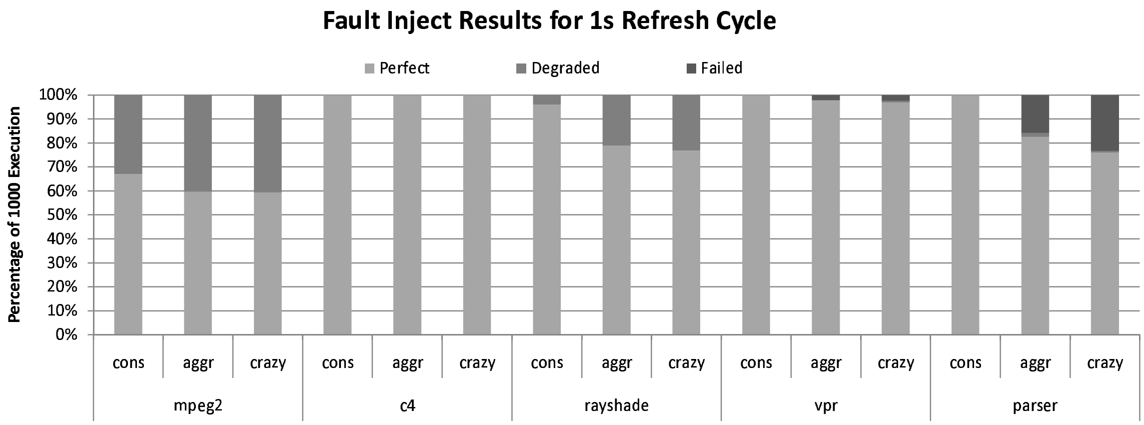


Figure 10. Fault-injection result for systems with low refresh rate of 1 second.

Application	Perfect	Degraded	Failed	SNR
mpeg2	0%	0%	100%	N/A
rayshade	42%	58%	0%	39.37dB
vpr	7%	0%	93%	N/A
parser	52%	10%	38%	N/A

Table 9. Results of injecting a single error in the critical heap data.

application is relatively small. Further, we perform fewer trials (50-100) than previous experiments as we obtained converging results even within these trials.

Table 9 shows the results of these experiments normalized to 100%. We exclude c4 from the experiments, because its only critical heap data is the game record, and this is precisely the output used for comparison. Therefore, all injections into the critical data of c4 will result in failures.

From Table 9, mpeg2 always fails (crashes) due to the injected errors because its output path or file pointer gets corrupted. On the other hand, rayshade does not fail but its output quality with even a single error in the critical data is 39 decibels on average, which is considerably worse than the quality with errors in non-critical data (over 70 decibels). Both parser and vpr experience high failure rates due to a single error in the critical data - vpr even more so than parser. The above results illustrate the importance of protecting critical data in applications and underline the need for data partitioning to prevent reliability degradation due to lowering of refresh rates.

6.4 Optimal Configurations

In this section, we combine the fault-injection results (Figure 10) with the power-savings results (Figures 8 and 9) to find the optimal configuration in terms of the power-reliability trade-off for each application. The main results are as follows:

- mpeg2, c4 and rayshade exhibit high overall power savings (20-25%) and no failures in the aggressive configuration. Further, the output quality is high (measured in SNR) for both rayshade and mpeg2 in the aggressive configuration. Hence, the best configuration for these applications is aggressive, suggesting that they have a large proportion of non-critical global data (see Section 5.3).
- For parser, the best results are achieved in the conservative configuration. This is because parser has a large proportion of non-critical data pages, and hence significant power savings (about 25%) can be achieved by putting these pages in the low-refresh mode. Further, parser experiences quite a few failures in the aggressive configuration, which suggests that it has a sizable chunk of critical global data.
- Finally, for vpr, the crazy configuration achieves the best overall power savings (nearly 25%) compared to the other two configurations. Further, even under the crazy configuration, the number of failures in vpr is marginal (less than 3%). This is because vpr has a significant proportion of stack data due to recursive calls, which is not critical to its correct execution.

7. Related Work

This section discusses related work in the areas of both hardware and software techniques for power reduction.

Hardware Techniques: Traditionally, hardware design techniques over-provision for the worst-case behavior of the platform. However, in the majority of common usage scenarios, the worst-case behavior is rarely exhibited, and the approach is often wasteful. Therefore, a new class of techniques have emerged that provision for the average case and treat the worst case behavior as an exception. This paradigm is referred to as *Better-Than-Worst-Case* (BTWC) [1]. Razor is one of the best known examples of the BTWC paradigm [12]. Razor reduces the energy consumption of processors by progressively lowering their voltage until such a point that the processor starts to experience errors due to timing violations.

At a high-level, Flikker is also a BTWC technique. However, unlike Razor and other BTWC technique which attempt to correct the introduced errors in hardware, Flikker exposes the errors all the way up the system stack to the application, thereby leveraging power-saving opportunities that were unexposed or infeasible at the architectural level alone. This is because many applications are naturally resilient to errors [28, 38], and this resilience can be exploited for power-savings through application-level techniques such a Flikker.

RAPID [36] is a hardware-software technique that applies the BTWC principle for DRAM refresh-power reduction. The main idea is to characterize the leakage behavior of each physical page and partition the pages into different classes based on their DRAM leakage characteristics. Applications preferentially use pages from the leakage class with the lowest leakage rate and the overall refresh rate is set based on the highest leakage class of pages allocated by the application (thereby preserving data integrity). In order for RAPID to be effective, applications must have substantial slack in memory usage. However, this assumption often does not hold for smartphone applications which are memory-constrained and typically operate near their peak memory capacities.

A number of other techniques modify the memory controller hardware to reduce unnecessary or redundant refreshes of DRAM cells [15, 23, 31]. These techniques however, require substantial changes to the memory controller's hardware compared to Flikker.

Finally, ESKIMO [20] is a hardware mechanism to save DRAM power using knowledge of application semantics. Similar to Flikker, ESKIMO modifies the memory allocator to expose details of the application's allocation patterns to the hardware. However, in terms of refresh-power reduction, ESKIMO differs from Flikker in two ways. First, ESKIMO focuses on reducing the refresh power of unused memory areas, while Flikker focuses on reducing the refresh power of the used memory areas. Second, ESKIMO attempts to preserve data integrity in the allocated areas, and hence has only limited opportunities for saving refresh-power (6 to 10%).

Software Techniques: Recently, a number of software-based techniques have been proposed that trade-off reliability for energy savings [2, 10, 16, 34]. These techniques share the same goal as Flikker, namely to reduce hardware reliability in an application-specific manner in order to achieve power savings. We discuss the techniques further in this section and then discuss the differences with Flikker.

Fluid-NMR [34] performs N-way replication of applications in a multi-core processors for tolerating errors due to reductions in voltage-levels of processors. The parameter N is varied based on the application's ability to tolerate errors. Relax is a technique to save computational power by exposing hardware errors to software in specified regions of code [10]. Relax allows programmers to mark certain regions of the code as "relaxed", and lowers the processor's voltage and frequency below the critical threshold when executing such regions, thereby allowing errors to occur during computation. Green [2] trades off Quality-of-Service (QoS) for energy efficiency in server and high-performance applications respectively. Green allows programmers to specify regions of code in which the application can tolerate reduced precision. Based on this information, the Green system attempts to compute a principled approximation of the code-region (loop or function body) to reduce processor power. Code perforation [16] is similar to Green, except that it attempts to infer the approximation code regions based on acceptance criteria provided by the user. Further, code perforation monitors the application at runtime and adapts the inference mechanism based on the application's behavior.

The above techniques are very similar to Flikker in their overall objectives. However, they differ from Flikker in two ways. First, they are task-centric and/or code-centric whereas Flikker is data-centric. In other words, the techniques require programmers to identify regions of code where errors are allowed (code perforation infers such regions automatically [16]), while Flikker requires programmers to identify data items where errors are allowed, i.e., non-critical data. We believe that it is more intuitive for programmers to identify non-critical data as data items often map directly to applications' outputs. Second, the above techniques target processor power reduction, while Flikker targets memory power reduction, which involves a different set of trade-offs and is hence orthogonal to the techniques.

In work submitted concurrently with our own, Salajegheh et al. propose "Half-wits" to save Flash power consumption by operating Flash chips at a lower voltage level and correcting errors with software techniques [33]. However, Half-wits fail to exploit the full potential of power reduction because it provides same level of reliability to both critical and non-critical data. The combination of Half-wits and Flikker will achieve more power savings.

8. Alternatives to Flikker

In this section, we consider alternative technologies to Flikker and qualitatively discuss the relative costs and benefits of the Flikker technique vis-a-vis these technologies.

Flash memory is predominantly used in smartphones as secondary storage. Flash memory is durable and does not need to be periodically refreshed. Hence it can be used to store the application's data before the smartphone transitions into sleep mode. Unfortunately, Flash memory read and write times are an order of magnitude higher than DRAM's, with the result that it is considerably slower read/write the contents of entire DRAM to memory. For a smartphone with 128 megabytes of DRAM and 16 megabytes per second effective Flash bandwidth, paging the whole main memory from Flash requires 8 seconds. Since memory capacity scales faster than bandwidth, this delay is likely to increase in future smartphones. While it is possible to accelerate the process by writing out only selected portions of the memory state, the challenges in doing so are similar to those faced by Flikker. In particular, the programmer must identify critical data in the application and be prepared to restore the application based only on the critical data.

Phase-Change Memory (PCM) is an emerging technology that offers better write performance and longer lifetime than Flash. Some recent proposals [25] have called for the partial replacement of DRAMs with PCMs in mobile devices. However, due to its overhead in dynamic power and access latency, PCM is not expected to completely replace DRAMs, but instead to be used for high-endurance but infrequently accessed data [25]. Flikker can also be applied in this context by storing the critical data only on PCM memory. This will allow the refresh rate of the entire DRAM array to be reduced rather than partition the array into the high-refresh and low-refresh part as done by Flikker. This is an avenue for future investigation.

ECC memory is widely used in systems that require extreme relibility. To enable error correction, ECC memory employs both extra storage and logic, which consumes power. As a result, ECC memory is not suitable for power-constrained systems. To the best of our knowledge, fault-tolerant refresh reduction [22] is the only technique that utilizes ECC memory for refresh power reduction. However, this technique targets the system scenario where only idle power consumption is considered. Hence, dynamic power consumed in the logic circuit is eliminated from the power overhead of ECC in this technique. An additional factor to consider is the increased cost of ECC memory, which may be a significant bottleneck to its adoption in commodity systems. Studies have shown that this cost may be as high as 25% in some systems [9].

9. Conclusion and Future Work

We present Flikker, a novel technique to save refresh power in DRAMs. Flikker enables programmers to partition the application data based on its criticality and lowers the refresh rate of the part of memory containing the non-critical data to save power. This separation introduces a modest amount of data corruption in the non-critical data, which is tolerated by the natural error-resilience of many applications. We prototyped the Flikker approach on a mobile device (using simulation), and find that it saves between 20-25% of total DRAM power in memory systems with less than 1% performance degradation and almost no loss in application reliability. Flikker represents a novel tradeoff in systems design, namely trading off hardware reliability for power-savings in an application-aware manner, as hardware only needs to be as reliable as the software requires.

Flikker can also be applied to data-center applications because they, (1) exhibit high variations in workloads and have considerable periods of inactivity, (2) consume significant power in idle-mode due to over-engineering, and (3) are inherently error-resilient and do not have to be 100% accurate [2]. Understanding the benefits of Flikker in this domain is a direction for future research.

Acknowledgements: We thank Emery Berger, Martin Burtscher, Shuo Chen, Trishul Chilimbi, John Douceur, Erez Petrank, Martin

Rinard, Karin Strauss, Nikhil Swamy and David Walker for useful comments and discussions about this work. We also thank our shepherd, Luis Ceze, and the anonymous reviewers for their constructive feedback.

References

[1] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the Asia South Pacific design automation conference*, pages 2–7, 2005.

[2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '10, pages 198–209, 2010.

[3] Vimal Bhalodia. SCALE DRAM subsystem power analysis. Master's thesis, Massachusetts Institute of Technology, 2005.

[4] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2004.

[5] Kirk W. Cameron, Rong Ge, and Xizhou Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38:40–47, 2005.

[6] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. *Usenix Annual Technical Conference (ATC)*, 2010.

[7] S. Chandra and P.M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *International Symposium on Software Reliability Engineering (ISSRE)*, page 91, 2002.

[8] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor-networks software faults. In *Proceedings of the International Symposium on Operating Systems Design (SOSP)*. ACM New York, NY, USA, 2009.

[9] Tezzaron corporation. Soft errors in electronic memory (white paper), 2005. Available from `http://www.tezzaron.com/about/papers/Papers.htm`.

[10] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. *International Symposium on Computer Architecture (ISCA)*, pages 497–508, 2010.

[11] C. Ellis, A. Lebeck, and A. Vahdat. System support for energy management in mobile and embedded workloads (white paper). Technical report, Duke University, 1999.

[12] D. et al. Ernst. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, 2003.

[13] Martin Fierz. 4 in a row. Available from `http://www.fierz.ch/4inarow.htm`.

[14] J. Flinn, K.I. Farkas, and J. Anderson. Power and energy characterization of the Itsy pocket computer (version 1.5). *Compaq Western Research Laboratory, Tech. Rep*, 2000.

[15] M. Ghosh and H.H.S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture(MICRO)*.

[16] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, Technical Report Massacchusets Institute for Technology, 2009.

[17] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, March 2011.

[18] Micron Technology Inc. 1gb mobile LPDDR. Available from `http://www.micron.com/products/partdetail?part=MT46H32M32LFCG-5IT`.

[19] Micron Technology Inc. System power calculator. Available from `http://www.micron.com/support/designsupport/tools/powercalc/powercalc.`

[20] Ciji Isen and Lizy John. Eskimo - energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *Proceedings of the International Conference on Microarchitecture (MICRO)*. IEEE, 2009.

[21] A.K. Karlson, B.R. Meyers, A. Jacobs, P. Johns, and S.K. Kane. Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker. In *Pervasive Computing: 7th International Conference, Pervasive 2009, May 11-14, 2009, Proceedings*, page 398. Springer, 2009.

[22] Y. Katayama, E.J. Stuckey, S. Morioka, and Z. Wu. Fault-Tolerant Refresh Power Reduction of DRAMs for Quasi-Nonvolatile Data Retention. In *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT-VLSI)*, volume 311, page 318, 1999.

[23] J. Kim and MC Papaefthymiou. Block-based multiperiod dynamic memory design for low data-retention power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):1006–1018, 2003.

[24] Craig Kolb. Rayshade graphics program. Available from `http://www-graphics.stanford.edu/~cek/rayshade`.

[25] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. *International Symposium on Computer Architecture (ISCA)*, 2009.

[26] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[27] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy management for commercial servers. *Computer*.

[28] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. *International Symposium on High-Performance Computer Architecture (HPCA)*, 0:181–192, 2007.

[29] Chi-Keung et. al. Luk. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language design and implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005. ACM.

[30] M. Murphy. *Beginning Android*. Apress, 2009.

[31] K. Patel, E. Macii, M. Poncino, and L. Benini. Energy-Efficient Value Based Selective Refresh for Embedded DRAMS. *Lecture Notes in Computer Science (LNCS)*, 3728:466, 2005.

[32] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: protecting critical data in unsafe languages. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008.

[33] Mastooreh Salajegheh, Yue Wang, Kevin Fu, Anxiao (Andrew) Jiang, and Erik Learned-Miller. Exploiting half-wits: Smarter storage for low-power devices. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, February 2011.

[34] John Satori, Joseph Sloan, and Rakesh Kumar. Fluid NMR-performing power/reliability tradeoffs for applications with error tolerance. *Workshop on Power Aware Computing and Systems (HotPower'09)*, 2009.

[35] SPEC. SPEC CPU2000. Available from `http://www.spec.org/cpu2000`.

[36] Ravi K. Venkatesan, Stephen Herr, and Eric Rotenberg. Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[37] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. *Kluwer International Series in Engineering and Computer Science*, pages 449–472, 1996.

[38] V. Wong and M. Horowitz. Soft Error Resilience of Probabilistic Inference Applications. *SELSE II*, 2006.

[39] W. Yuan, K. Nahrstedt, SV Adve, DL Jones, and RH Kravets. GRACE-1: Cross-layer adaptation for multimedia quality and battery energy. *IEEE Transactions on Mobile Computing*, 5(7).