Flicker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning

Song Liu[†], Karthik Pattabiraman[‡], Thomas Moscibroda[‡],
Benjamin G. Zorn[‡]
[†] Northwestern University, Evanston, IL, USA
[‡] Microsoft Research, Redmond, WA, USA

Contact: zorn@microsoft.com

Technical Report MSR-TR-2009-138
October 2009

Microsoft Research
One Microsoft Way Redmond, WA 98052, USA
http://research.microsoft.com

Flicker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning

Song Liu†, Karthik Pattabiraman, Thomas Moscibroda, Benjamin G. Zorn
Microsoft Research, †Northwestern University

Abstract

Mobile devices are left in sleep mode for long periods of time. But even while in sleep mode, the contents of DRAM memory need to be periodically refreshed, which consumes a significant fraction of power in mobile devices. This paper introduces *Flicker*, an application-level technique to reduce refresh power in DRAM memories. Flicker 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 saves energy at the cost of a modest increase in data corruption in the non-critical data. Flicker thus explores a novel and interesting trade-off between energy consumption and hardware correctness. We show that many mobile 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 Flicker can save between 20-25% of the power consumed by the memory subsystem in a mobile device, with negligible impact on application performance. Flicker is implemented almost entirely in software, and requires only modest changes to the application, operating system and hardware.

1 Introduction

Energy has become a first-class design constraint in many computer systems, particularly in mobile devices and server-farms [9, 23]. In mobile phones, saving energy can extend battery lives and enhance mobility. In the recent past, mobile phones have morphed into general computing platforms, often called smartphones, (e.g., the iPhone). Smartphones are typically used in short-bursts over extended periods of time [17], i.e., they are idle most of the time (approximately 95% of the time). Nonetheless, they are "always-on" as users expect to resume their applications in the state they last used it. 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 experience data loss. This energy drain is particularly problematic because memory is a significant contributor to the power consumption of a smart phone (30% or more according to studies of the Itsy pocket computer [11]), and the refresh operation is the dominant consumer of power in idle mode. Therefore, reducing refresh power can significantly reduce the overall power-consumption of the mobile phone and extend its battery-life.

This paper proposes *Flicker*[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 current of memory-cells follows an exponential distribution [18], 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. Flicker lowers the DRAM refresh rate in order to obtain power-reduction 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 inherently resilient to errors in all but a small portion of their data [24, 44]. We call such data *critical data*, as it is important for the overall correctness of the application. For example, in a video processing application, the data-structure containing the list of frames is more important than the output buffer (as the human eye is tolerant to mild disruptions in a frame). Previous work has shown the feasibility of identifying critical data in applications for protecting them from pointer-based memory corruption errors [5, 33].

Flicker is implemented mostly in software, with modest changes to the hardware. It enables the programmer to distinguish between critical and non-critical data in applications. At runtime, Flicker 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 Flicker to achieve power savings without degrading the application's reliability.

Our approach in Flicker fundamentally differs from existing techniques for saving energy in low-power systems. In these solutions, energy reduction is achieved by

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

appropriately trading-off performance metrics, such as throughput/latency, Quality-of-Service (QoS), or user response time, e.g. [43, 45]. In contrast, the key novelty of our approach is to leverage a so-far unexplored trade-off in system design, namely *trading-off energy efficiency for data integrity at the hardware 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.

Flicker thus falls into the category of techniques known as *better-than-worst-case (BTWC)* designs, an example of which is *Razor* [8]. BTWC techniques exploit the over-engineering of hardware to obtain performance improvement or power-savings at the cost of reliability. However, existing BTWC techniques correct the introduced errors in hardware and maintain the abstraction of correct hardware to software. In contrast, Flicker exposes hardware 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. *To the best of our knowledge, Flicker is the first software technique to intentionally introduce hardware errors for power-savings based on the characteristics of the application*.

Aspects of Flicker make it appealing for use in practice. First, Flicker 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, Flicker does not require any changes to the memory controller hardware or circuitry other than 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) [28]), 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 Flicker, because the default mode of Flicker is to consider all application data as critical, unless the application developer indicates otherwise. Further, even in non-legacy applications, developers can be conservative in identifying non-critical data in their applications.

We have evaluated Flicker both using analytical and experimental methods on five diverse applications representative of mobile workloads. The analytical model was derived based on typical mobile-phone usage patterns[17] and power-values from the Micron mobile DRAM data-sheet [28]. We use an architectural simulator to estimate the power and performance overheads of Flicker, and fault-injection experiments to evaluate application resilience under Flicker. We find that Flicker can save between 20% to 25% of the total DRAM power in mobile applications, with negligible degradation in reliability and performance (less than 1%). We also find that the effort required to deploy Flicker is small (less than half-a-day for each application considered in the paper).

This paper is organized as follows: §2 provides a brief overview of the design of Flicker. §3 and §4 present the hardware and software implementation of *Flicker* respectively. §5 and §6 present the experimental setup and the results of evaluating Flicker. Related work is discussed in §7 before §8 concludes.

2 Flicker: Design Overview

Flicker requires a modest set of simple changes to both hardware and software. Flicker 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, Flicker (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 Flicker are explained in detail in §3 and §4, respectively.

The energy-gains of Flicker are thus a result of optimizing across the hardware/software boundary. This "cross-layer" approach that involves minor changes to the hardware may seem daunting at first. However, we believe that it is justified for two reasons.

First, 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. The DRAM's task was to *ensure* that data was stored reliably whenever the memory was powered. In Flicker, we re-examine this assumption and consciously allow the DRAM to violate data integrity to a limited degree for the purpose of reducing energy consumption. In §6, we show that this *trade-off between application reliability and energy efficiency* can be exploited to achieve significant energy improvements at almost no degradation to application performance or end-to-end correctness.

Secondly, small hardware changes are not an inherent show-stopper in terms of practical deployment, given that mobile phone architectures (both hardware and software) are still in a state of flux. New hardware and software models for mobile phones are being developed and put to market regularly. For example, the Partial-Array Self-Refresh Mode (PASR) architecture for mo-

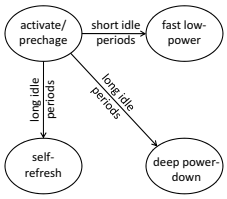


Figure 1: Simplified diagram of DRAM operating states.

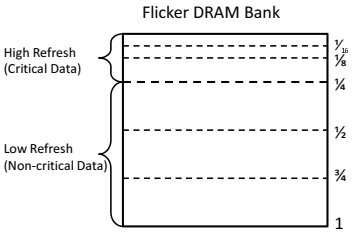


Figure 2: Flicker Bank Architecture. The DRAM bank is partitioned into two parts, the high refresh part, which contains critical data, and the low refresh part, which contains non-critical data. The high/low refresh partition can be assigned only at discrete locations (shown by the dashed lines).

bile DRAMs on which we base Flicker was introduced in the year 2003 [14].

3 Flicker Hardware

This section presents the hardware architecture of Flicker, i.e., the Flicker 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 Flicker DRAM (§3.4). This model is then used to determine the best refresh rates in our system design (§3.5), and to estimate power savings in our evaluations (§5). We find that a refresh rate of 1 second provides a near optimal energy-reliability tradeoff.

We begin by providing a brief background on DRAM systems in low-power mobile devices.

3.1 Background

Mobile phones have traditionally used SRAMs (Static Random Access Memories) for memory. However, as memory capacity increases, conventional SRAM becomes more expensive (per byte of memory), and hence, smart phones have adopted DRAM (Dynamic RAMs). A DRAM memory system consists of three major components: (1) multiple banks that store the actual data, (2) the memory controller (scheduler) that schedules commands to read/write data from/to the DRAM banks, and (3) 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.

A typical mobile DRAM chip has several modes of operation, as shown in Fig. 1. The only state in which the DRAM can be read from/written to is the active state (activate/precharge). When the DRAM is idle for short periods of time between accesses, it transitions to fast low-power states, which have short wake-up times (on the order of 10 nano-seconds or less), and less than half the power-consumption of the active state. When the system is in sleep mode, i.e., the processor is not executing any application for a long period of time, the DRAM chip transitions to the self-refresh state, which has a much lower power-consumption than the active and fast low-power states. However, the refresh operations are still carried out in the self-refresh state. Finally, when even further power-savings are desired, the DRAM chip can transition to the deep power-down state, where even refresh operations are stopped. However, the DRAM will lose its data in the deep power-down state, and hence this state is rarely used in mobile systems.

Flicker is targeted towards reducing power-consumption in the self-refresh state, i.e., when the mobile device is idle. In this state, the DRAM array is periodically refreshed even if the processor is in sleep mode and not accessing any data. Therefore, the self-refresh operation is performed by dedicated hardware on the DRAM chip without any software intervention. The Operating System (OS) needs to initiate the self-refresh process before putting the mobile device to sleep.

Partial Array Self Refresh (PASR) is an enhancement of the self-refresh low power state [28] in which only a portion of the DRAM array is refreshed in sleep-mode. The other portion is not refreshed and will lose its data. The portion of the memory array that should be refreshed can be configured by the OS before switching the device to sleep mode. The portions are typically discretized in powers-of-two fractions. For example, Micron's mobile DDR SDRAM [28] with 4 banks has five different options for PASR, namely, 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 drawback of using PASR is that it reduces the amount of memory available in self-refresh state, which can be a constraint for mobile applications.

3.2 Flicker DRAM Architecture

This section describes in detail the architecture of the Flicker DRAM. The Flicker DRAM enhances the PASR mode as follows. Instead of stopping refresh entirely to a portion of the memory array, it refreshes different por-

tions of the array at two different frequencies, one portion at the regular refresh frequency and the other portion at a much reduced frequency. Similar to PASR, the OS specifies how much of the DRAM array should be refreshed at the high frequency before putting the mobile device to sleep. This portion is also discretized to a power-of-two fraction. Unlike PASR, however, the portion of the DRAM that is refreshed at the lower rate is still usable, although it may experience errors.

Flicker DRAM: Figure 2 illustrates a Flicker DRAM bank. In the Flicker 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 or 32 milliseconds in most systems). The error rate of data in these high refresh parts is negligible and is 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 (§3.3).

Notice that we cannot simply emulate the behavior of longer refresh cycle times by having the OS periodically adjust the portions of the memory array to be self-refreshed, because the device is in sleep mode and the OS is thus unable to set any register values. Therefore, we need to modify the self-refresh hardware to refresh the DRAM at two different rates.

Hardware changes: In order to implement Flicker, small changes need to be made to mobile DRAM architectures. In particular, state-of-the-art mobile DRAMs use a self-refresh counter to remember which row to refresh next during the self-refresh operation. Flicker DRAM extends this counter by a few extra bits and adds an additional "refresh enable" output to the counter. A Flicker DRAM row is refreshed only when the refresh enable bit is set to "1". A controller sets different values to the refresh enable bit based on the higher bits of the row address and the extra bits, and thus configures the refresh rate of different DRAM rows at different values. The modifications made to the self-refresh counter by Flicker are shown in Figure 3.

Example: For illustration, consider a Flicker DRAM system where $T_{low} = 16 \times T_{regular}$. In this system, the self-refresh counter requires 4 more bits than corresponding PASR implementation. 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 have a predefined value (say "1111"). For example, if 1/8 of the DRAM has been configured as high refresh, i.e. high refresh row addresses are those with highest bits of "000" (while low refresh rows addresses are the rest). When the extra bits are "0000" through "1110", the refresh enable bit is only set for high refresh row addresses. However, whenever the extra bits are "1111", the refresh enable bit is set for all row addresses. With this configuration, the low refresh rows are refreshed 16 times less frequently than the high refresh rows.

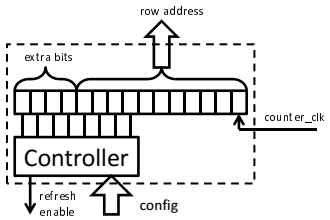


Figure 3: Self-refresh counter in the Flicker DRAM.

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]).

3.3 Flicker DRAM Error Rates

This section illustrates the dependence between error-rates and refresh cycle times in a DRAM array. DRAM cells experience vast variations in their retention times, and hence require different refresh rates. Current DRAM chips however refresh all cells at the same rate which corresponds to cells with the lowest retention times. In Flicker, we consciously lower the refresh rate for a portion of the DRAM array, and hence some fraction of cells in this portion are likely to lose their data, i.e., experience errors. We estimate this fraction of cells (error rate) as a function of the refresh cycle time of the low-refresh part.

Previous work [3, 41] has measured DRAM error rates as a function of refresh cycle times. Although these two measurements are at different granularities (per cell versus per row), their results are consistent with each other. Table 1 shows the per-cell error rates we use based on Bhalodia's measurements [3].

Note that the above error-rates correspond to a temperature of 48°C. The retention times of DRAM cells decrease with temperature and hence, for a given refresh cycle, the error-rate increases with ambient temperature. An operating temperature of 48°Cis higher than the ambient temperature of most smartphones, and hence our error-rates are likely higher than those in reality.

3.4 Flicker DRAM Power Model

We do not have a hardware implementation of the Flicker DRAM, and hence cannot directly measure its power consumption. Therefore, we derive an analytical model to estimate its power-consumption. The model is based

High Refresh Size	Self-Refresh Current [mA]			
	PASR	Flicker		
		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

* This value is derived from linear interpolation of full array (1) and half array(1/2) cases.

Table 2: Self-refresh current in different PASR and Flicker configurations (PASR current values are from [28].)

on real power-measurements in mobile DDR DRAMs with PASR from Micron's data-sheet [28].

The self-refresh power consumption $P_{Flicker}$ is calculated as follows:

$$\begin{aligned} P_{Flicker} &= P_{refresh} + P_{other} \\ &= P_{refresh\_low} + P_{refresh\_high} + P_{other} \\ &= \left(P_L \times \frac{T_{regular}}{T_{low}} \right) + (P_{refresh\_high} + P_{other}) \\ &= \left((P_{full} - P_{PASR}) \times \frac{T_{regular}}{T_{low}} \right) + P_{PASR} \end{aligned} \quad (1)$$

As shown in Equation 1, $P_{Flicker}$ has two components, $P_{refresh}$, which is the power consumed in refresh operations, and P_{other}, which is the power consumed in other parts of the DRAM (e.g., the control logic). $P_{refresh}$ is proportional to the refresh rate, while P_{other} is a constant that is independent of refresh rate. 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 of Flicker DRAM respectively (second line of Equation 1). In order to explicate the relationship between refresh power and refresh cycle time, we represent $P_{refresh\_low}$ as P_L (which is a constant) times $T_{regular}/T_{low}$ (third line in Equation 1).

In order to evaluate the value of different components of $P_{Flicker}$, we consider PASR DRAM and regular DRAM (full array refreshed at $T_{Regular}$) as two extreme cases of Flicker. We calculate P_{PASR} and P_{full} by assigning $T_{low} = \infty$ and $T_{low} = T_{regular}$ in the third line of Equation 1. Based on these two extremes cases, we rewrite the third line of Equation 1 with P_{PASR} and P_{full} (fourth line of Equation 1)

Table 2 summarizes the self-refresh current of different PASR configurations and Flicker 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 to zero even if the refresh period increases to infinity.

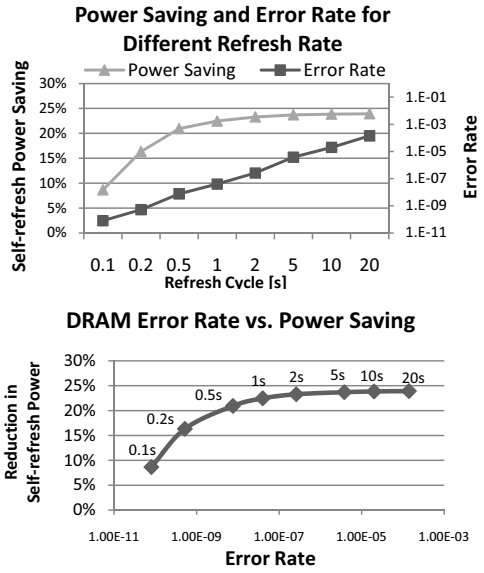


Figure 4: Error rate and power saving for different refresh cycles. The high refresh part is 1/4 of DRAM array.

3.5 Power-Reliability Trade-off

The models derived in the two previous sections are used to find a suitable refresh rate for Flicker. Figures 4 show 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 steadily 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. 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 trade-off between power savings and reliability*.

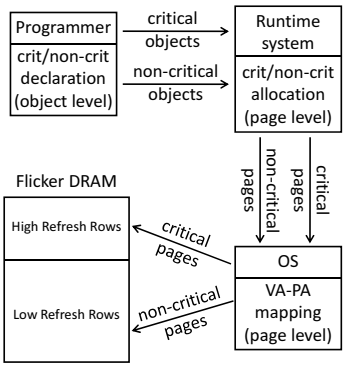


Figure 5: Flicker system diagram.

4 Flicker Software

In this section, we show how to modify existing software so that they can use the Flicker DRAM. Figure 5 shows the steps involved in the operation of Flicker. 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 Fig. 5, modifications need to be made to the application, the runtime system and the OS. We now describe how each of the above components is modified.

4.1 Application

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. Many applications already distinguish between soft-state and persistent-state, and take active measures to protect the persistent state by writing it to the file-system periodically. It has been shown that due to the natural separation among persistent- and soft-state in an application, distinguishing critical from non-critical data in applications is often straight-forward [5, 33]. Our results in this paper further confirm this observation as each application considered in the paper took us less than half-a-day to partition (including the time we spent understanding its source code).

In Flicker, 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 our system currently does not provide (see §4.2 for the implications). 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.

Note that the programmer maintains control over what data should be marked critical or non-critical in the program. In case the programmer is unsure about whether to mark a certain piece of data non-critical, she can be conservative and mark it critical. Doing so will limit the amount of power-savings Flicker provides, but will not impact the reliability of the application. Further, programmers can incrementally mark non-critical data in the application based on the amount of power-savings they wish to obtain with Flicker and the extent of reliability degradation they are willing to tolerate.

4.2 Runtime System

Flicker 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 would be partitioned into critical/non-critical parts. Our current version of Flicker does not implement partitioning of global data as this requires compiler support. However, as we will show in the experimental results (§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 Flicker, 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 Fig. 2), and non-critical pages to the low refresh part (bottom up in Fig. 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, but this may not always be possible due to discretization in the self-refresh mode (§3.4). Therefore, the OS may end up placing more DRAM rows in high-refresh state than absolutely necessary, leading to wasted power. However, as we show in §5, this does not significantly impact the power-savings of Flicker.

5 Experiment Setup

In this section, we present the applications and experimental methods used to evaluate Flicker. As mentioned in Section 3, Flicker requires minor changes to the hardware and hence it is not possible to evaluate it directly on a mobile device. Therefore, we use hardware simulation based on memory traces from real applications to evaluate the performance overheads and active power consumption of Flicker. 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 Flicker. 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 Flicker, 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 as follows.

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 [22] 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 [10] (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.

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 Flicker. (LoC = Lines of Code)

rayshade: Rayshade [20] is an extensible system for creating ray-traced images. Rayshade represents a growing class of mobile 3D applications [31]. 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 [40] 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 [40] 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 in the program. This foot-print is used to calculate power-consumption in idle-mode. These measurements were performed by enhancing the Pin dynamic-instrumentation framework [25].

Architectural simulator: We use a cycle-accurate architectural simulator for evaluating the Flicker hardware. The simulator contains a functional front-end based on

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 [28])
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.

Pin [25] and a detailed memory system model. A DRAM power model based on the system-power calculator [29] 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 smart phone with a 1GHz processor and 128 Mega-bytes of DRAM memory.

Fault-injector: We built a fault-injector based on the Pin [25] 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 (3rd 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 maintained by the injector.

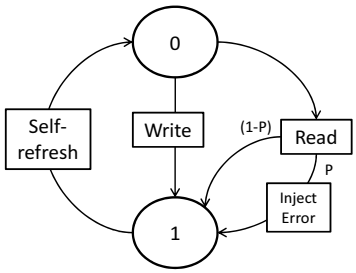


Figure 6: State transition diagram of "modified" bit in fault-injection. Error is injected to the DRAM with probability "P".

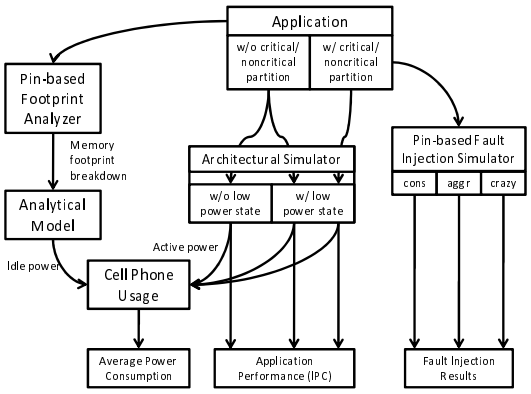


Figure 7: Evaluation Framework.

5.3 Experimental Methodology

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

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

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

Critical Data Partitioning: We modify all 5 applications to use Flicker'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

[2]Given the low error rates presented in Table 1, the probability of multiple bit flips in a single byte is very low.

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 Flicker

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 (No. of 4kB pages).

bound the performance benefit and the reliability impact of partitioning the global data respectively. We anticipate that partitioning global data yields a power-savings close to that of the aggressive configuration 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 evidence that this is indeed the case for almost all the applications considered in this paper).

In the above discussion, we assumed that stack data is placed in the 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 both the stack and critical data are also placed in low-refresh state. Table 5 summarizes the configurations used for evaluating each application.

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 in §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, as derived in §3.5. Further, the high refresh portions in each application are rounded up to discrete levels in Table 2.

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 meaure the Instructions Per Cycle (IPC) of the system[3], and for evaluating the power consumption, we measure the total energy consumed by each DRAM bank and divide it by the simulation time. The simulations are performed with application traces consisting of 100 million instructions chosen from the approximate middle of the execution of each application (this is standard in architectural simulations where simulating the entire application may not be feasible due to prohibitive performance costs). We repeat the simulations for multiple intervals in 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 Flicker stems from the partitioning of application data, which can potentially impact locality and bank-parallelism. Therefore, the overhead of Flicker is evaluated by considering a system that employs data-partitioning (Part) with one that does not (Base)[4]. In both cases, we assume that the DRAM aggressively transitions to low-power states when not in use, as mentioned in Section 3.1.

Power-savings calculation: We assume a mobile DRAM device having a capacity of 128 Megabyte, which corresponds to the memory capacity of current smartphones (e.g., the iPhone)[5]. 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 idle mode (self-refresh state) as assumed in prior work [41].

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 smaller 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, and record them. 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

[3]The IPC defined as the average number of instructions executed in each clock-cycle of the processor

[4]The refresh rate plays no part in the measurement of active power.

[5]Future smartphones may have higher memory capacities, and hence the power savings achieved by Flicker will be even higher.

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 Flicker

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

This section discusses the results of experiments used to evaluate the power-savings, reliability and performance degradation due to the Flicker system.

6.1 Performance & Active Power

Table 7 shows the performance (IPC) 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 confirm that the IPC and active power consumption of the Base and Part scenarios are similar for all applications (both within 1% of each other). Therefore, the performance overhead of Flicker is negligible for the applications considered. Further, Flicker does not significantly increase the active power consumption of the application. Note that in some cases, the active power consumption is actually lower in the Part scenario because of increase in bank-parallelism due to the partitioning.

6.2 Power Reduction

Fig. 8 shows the reduction in DRAM standby power for different applications and the three configurations in Table 5. Fig. 9 shows the overall power reduction for different applications, which are obtained by combining the results in Fig. 8 with the active power-measurements in Table 7. We make the following observations.

- 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).

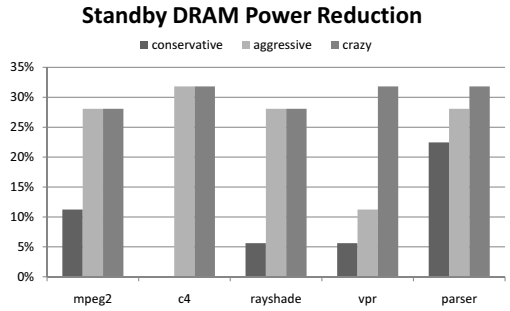


Figure 8: Standby DRAM power reduction.

- The aggressive configuration achieves significant power-savings in all other 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 when the stack and critical data 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.

6.3 Fault Injection Results

In this section, we present the results of fault-injection experiments to evaluate the reliability of Flicker. 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 One Second Refresh Period

Figure 10 shows the result of the fault-injection experiments for five applications and three configurations with

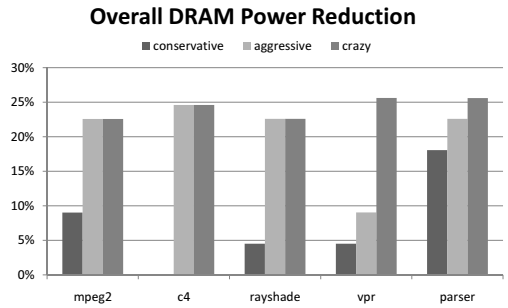


Figure 9: Overall DRAM power reduction.

an error-rate corresponding to a 1 second refresh period. Each bar in the figure represents the result of 1000 fault-injection trials. The results are normalized to 100%.

The main results from Figure 10 are:

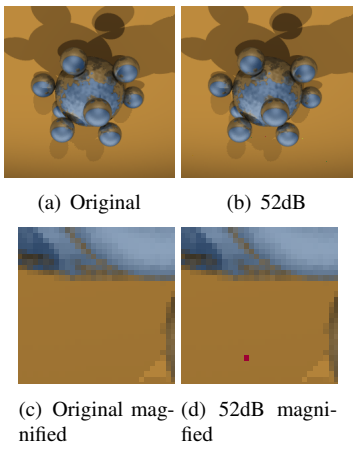
- No application exhibits failures in the conservative configuration. In fact, c4, vpr, and parser, have perfect outputs under the conservative configuration. However, mpeg2 and rayshade have a few runs with degraded results (about 33% for mpeg2 and 4% for rayshade), but as we show later in the section, the degradation is very small.
- Both aggressive and crazy configurations yield worse results than the conservative configuration for all applications, except for c4. This is because c4 has a very small proportion of critical pages, and these 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 stack and critical 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 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

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) Original magnified (d) 52dB magnified

Figure 11: Rayshade output figures with different SNRs.

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 encoded a raw video 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 11a and 11b 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 11c and 11d), 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 Injection to Critical Heap Data

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 critical data.

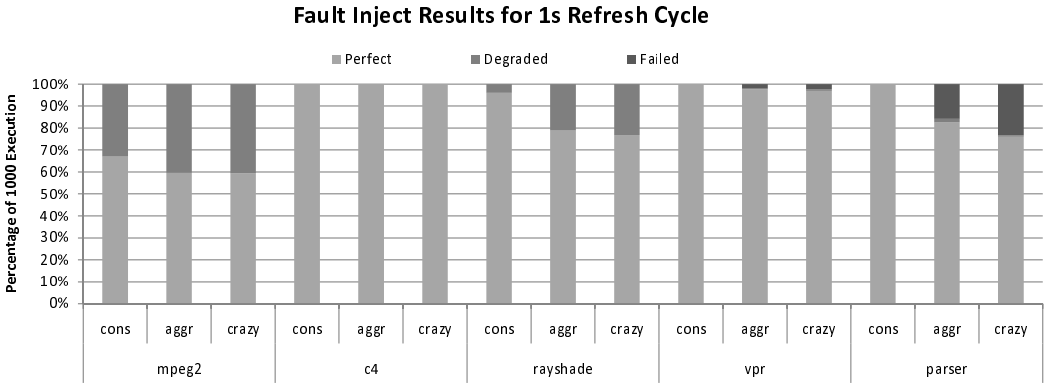


Figure 10: Fault-injection result for systems with low refresh rate of 1 second.

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.

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 data-partitioning) is unnecessary for 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 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[6]. 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 much 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

We now combine the fault-injection results (Fig. 10) with the power-savings results (Figs. 8 and 9) to find the optimal configuration in terms of the power-reliability tradeoff for each application. The main results are:

- 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 mostly non-critical. Hence, crazy is the optimal configuration for vpr.

6.5 Power vs. Output Sensitivity Analysis

In order to validate the analytical model in Section 3.4, we study the sensitivity of the power/reliabiluty results to the refresh rate. Due to space constraints, we focus on the mpeg2 application for these experiments. We repeat each experiment for 100 trials for refresh cycles ranging from 1 to 20 seconds for the conservative and aggressive configurations. Figure 12 shows the percentage of power saving and output quality (in decibels) for the mpeg2 application at different refresh cycles.

From the figure, we see that the output quality of the conservative configuration is always better than that of

[6]Therefore, the injections into c4 will yield 100% degraded outputs.

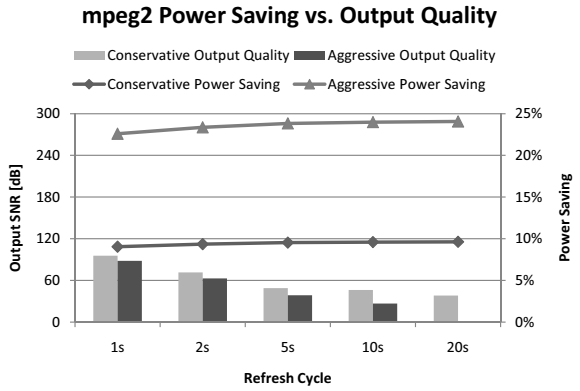


Figure 12: Sensitivity Analysis of mpeg2 in terms of power saving and output SNR. The primary y-axis shows the output video SNR in dBs. The secondary y-axis is the overall DRAM power saving in percentage. An infinite SNR is plotted as 100dB.

aggressive. However, the difference in output quality between conservative and aggressive increases as the refresh period increases. For example, with a 1 second refresh-period, both conservative and aggressive yield high-quality outputs (over 88 decibels). However, when the refresh period is increased to 20 seconds, the conservative configuration yields output quality of over 38 decibels; while the aggressive configuration's output quality drops to 0 decibels.

The power-savings obtained when increasing the refresh rate from 1 to 20 seconds is relatively small for both the conservative (0.5%) and aggressive (1.6%) configurations. This confirms our earlier result from the analytical model (Section 3.4) that reducing the refresh rate beyond 1 per second does not provide significantly higher power savings. Hence a refresh period of 1 second provides a near optimal power-reliability trade-off for most applications.

7 Related Work

This section discusses related work in the areas of critical data protection, error-resilience of applications, better-than-worst-case designs and DRAM refresh power reduction.

7.1 Critical-data protection

Critical data is defined as data whose integrity is important to the overall correctness of the application. Checkpointing and rollback-based systems periodically store the application's critical data to disk, and restore the state of the critical data in the case of an application failure. However, these mechanisms assume that the state of the critical data is uncorrupted during roll-back. This assumption may not always be true as shown by Chandra and Chen[4]. Hence, we need robust mechanisms to protect critical data in applications.

Samurai [33] is a general-purpose mechanism to protect critical data in applications from errant pointer writes due to type-unsafe code. Similar to Flicker, Samurai modifies the memory allocator to isolate and protect the application's critical heap data. However, there are two important differences between Samurai and Flicker. First, Samurai requires the application developer to manually identify the places in the application (i.e., source lines) where the critical data is legitimately updated. This can be tedious for developers as it requires significant modifications to the source code. The second difference is that Samurai employs in-process replication of critical data in order to ensure that at most any one copy of the data can be corrupted due to a memory error. This can be expensive if the amount of critical data to be protected is large or if memory space is limited.

7.2 Error-resilience of applications

A number of studies have showed that applications are resilient to low-level hardware errors [6, 13]. Broadly, there are two reasons for the resilience. First, each layer of the system stack masks a subset of errors that occur at lower layers of the stack. Hence, the number of errors that are exposed to the application (i.e., impact some element of the application's state) become progressively lower with the level of origin of the error. For example, only a small fraction of errors (less than 15% [37, 42] that arise at the lowest level of the system stack, namely circuit errors impact the application. This phenomenon is called *error derating*, and arises due to myriad circuit and architectural characteristics [7, 46].

The second reason for the resilience of applications is that many applications are inherently tolerant of small-deviations in their state, and can produce acceptable outputs even if there are faults in the architecture-visible state. This phenomenon has been shown for soft-computing applications such as probabilistic-inference [44]. Surprisingly, it has also been observed in general-purpose applications [24, 30]. This is because applications may perform many computations that are ultimately discarded (e.g., sub-optimal solutions in optimization algorithms) and hence errors in these computations are unlikely to affect application state. Further, applications may employ error-correcting schemes (e.g., checksums in gzip) to automatically correct single errors in their data.

7.3 Better-Than-Worst-Case Designs

Traditional hardware-design techniques consider the worst-case behavior of the hardware circuit and over-provision for that behavior. However, in common us-

age, the worst-case behavior rarely occurs, and this approach is wasteful. In contrast, *Better-Than-Worst-Case* (BTWC) design approaches provision for the common-case behavior and handle worst-case behavior as exceptions [1]. A well known example of the BTWC paradigm is Razor [8], which reduces the energy consumption of processors by progressively lowering their voltage until the processor starts to experience errors due to timing violations. Razor corrects the introduced errors by flushing the processor's pipeline.

RAPID [41] is a software technique that applies the BTWC principle to DRAM refresh-power reduction. RAPID characterizes the leakage behavior of each physical page and partitions the pages into different classes based on their leakage. 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). RAPID benefits from applications' slack in memory usage, a condition that often does not hold in smart-phone applications which are memory-constrained. Further, RAPID is orthogonal to Flicker and the two techniques may be combined to achieve even higher power-savings.

Fault-tolerant refresh reduction is a circuit-level BTWC technique [18] that uses ECC memory. Similar to Flicker, this assume that the majority of memory cells are likely to retain their charge even at reduced refresh rates and use ECC to correct the errors that they introduce. Flicker provides similar benefits to the above approach without requiring expensive ECC memory.

A number of other techniques modify the memory controller hardware to reduce unnecessary or redundant refreshes of DRAM cells. *Block-based refresh* [19] assigns different refresh rates to different memory blocks to avoid refreshing the entire DRAM at the frequency of the fastest-leaking cells. *Smart refresh* [12] avoids refreshing DRAM rows that have recently been read/written as these have had their charge restored by the read/write operation. *Value-based refresh* [32] leverages the observation that memory consists predominantly of zeroes and selectively disables refresh of clusters of zeroes. All of the above techniques require substantial changes to the hardware and are orthogonal to Flicker.

In very recent work, ESKIMO [15] saves DRAM power using knowledge of application semantics. The main idea is to reduce both active power and refresh power based on application knowledge of unused memory areas. Similar to Flicker, ESKIMO modifies the allocator to expose details of the application's allocation patterns to the hardware. In terms of refresh-power reduction, ESKIMO differs from Flicker in two ways: First, ESKIMO reduces the refresh power of unused memory areas, while Flicker reduces the refresh power of the used memory areas. Second, ESKIMO does not trade-off reliability as Flicker does, and hence provides only limited refresh-power savings (6 to 10%).

7.4 Energy–Reliability Trade-off

Recently, a technique called Fluid-NMR [38] 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. While Fluid-NMR and Flicker have similar goals , there are important differences. First, Fluid-NMR reduces processor power in active mode while Flicker reduces memory power in standby mode. Second, Flicker does not require replication of processes and can be deployed even on single-core processors.

Rinard [36] proposes to discard application tasks that experience errors in a computation, provided the distortion in the application's output is within specified bounds. Similar to Flicker, this approach leverages the slack in application's reliability, but their focus is on performance improvement rather than power-reduction.

The Eon programming language and runtime system allows programmers to annotate paths in the program (i.e., flows) to trade-off output fidelity for power-savings [39]. However, Eon requires programmers to re-structure their program in the form of flows by using Eon's programming language. In contrast, Flicker only requires that programmers identify critical data in existing programs through annotations or minor code changes to use a different allocator.

Green [2] trades off Quality-of-Service (QoS) for energy efficiency in server applications by allowing 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 marked region to reduce processor power. Similar to Green, SpeedGuard [35] performs approximations of code-regions without requiring programmer intervention. These systems share similar goals as Flicker, but there are important differences between them. First, they are code-centric, while Flicker is data-centric. Second, they target processor power while Flicker targets memory power.

Clumsy Packet Processors (CPPs) [26] are deliberately over-clocked processors used in network processing which make occasional mistakes in computation (due to over-clocking). Network applications are inherently error resilient as they have to deal with many natural causes of failures. CPPs leverage this error-resilience to obtain power-savings in the processor. CPPs differ from Flicker in that they do not protect important state in the application, and may hence fail catastropically.

A number of network systems have explored the trade-

off between energy savings and reliability. These systems have typically focused on reducing the power-consumption of fault-tolerance mechanisms such as replication [16] or checkpointing [27, 34].

7.5 Alternatives to DRAM

Today, most smart-phones use DRAM as the main memory and Flash memory as secondary storage. Unlike DRAMs, Flash memory is durable and does not need to be periodically refreshed. Hence it can be used to store critical data before the smart-phone transitions into sleep mode. However, Flash memory read and write times are an order of magnitude higher than DRAMs, with the result that it may take a long time to restart the phone from sleep mode, and hence make it less responsive. While it is possible to accelerate the process by writing out only selected memory state, the challenges are similar to those faced by Flicker, i.e., identifying critical data in the application. Further, Flicker requires only that the program be able to tolerate a small number of errors in the non-critical data, rather than make the entire non-critical data unavailable to the application.

Phase-Change Memory (PCM) is an emerging technology that offers unlimited data endurance while providing higher read/write speeds than Flash. Recent proposals have called for the replacement of DRAMs with PCMs [21], however this replacement incurs significant performance and power costs. Flicker may be able to ease this transition by storing the critical data on PCMs and the non-critical data on regular DRAM.

8 Conclusion and Future Work

We present Flicker, a novel technique to save refresh power in mobile DRAMs. Flicker 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 mobile applications. Results show that Flicker saves between 20-25% of total DRAM power in memory systems with very little performance degradation (less than 1%) and no loss in application reliability. Flicker represents a novel tradeoff in systems design, namely trading off hardware reliability for power-savings, as hardware only needs to be as reliable as the software requires.

We plan to extend Flicker for saving energy in data-centers. Similar to mobile applications, data-center applications (1) have considerable periods of inactivity due to workload variations, (2) consume significant power in idle-mode, and (3) are inherently error-resilient as they do not have to be 100% accurate. Therefore, data-centers are particularly well-suited to a Flicker-like approach.

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.

References

[1] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Proc. Conference on Asia South Pacific Design Automation*, 2005.

[2] W. Baek and T. Chilimbi, "Green: A System for Supporting Energy-Conscious Programming using Principled Approximation," Microsoft Research, Tech. Rep., 2009.

[3] V. Bhalodia, "SCALE DRAM subsystem power analysis," Master's thesis, Massachusetts Institute of technology (MIT), 2005.

[4] S. Chandra and P. Chen, "How fail-stop are faulty programs?" in *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1998, pp. 240–249.

[5] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving sensor-networks software faults," in *SOSP*, 2009.

[6] G. Choi, R. Iyer, and V. Carreno, "Simulated fault injection: a methodology to evaluate fault-tolerant microprocessor architectures," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 486–491, 1990.

[7] J. Cook and C. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 482–491.

[8] D. Ernst et al., "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO*, 2003.

[9] C. Ellis, A. Lebeck, and A. Vahdat, "System support for energy management in mobile and embedded workloads: A white paper," Duke University, Tech. Rep., 1999.

[10] M. Fierz, "4 in a row." [Online]. Available: http://www.fierz.ch/4inarow.htm

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

[12] M. Ghosh and H. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D Die-Stacked DRAMs," in *MICRO*, 2007, pp. 134–145.

[13] J. Gray and D. Siewiorek, "High-availability computer systems," *IEEE Computer*, vol. 24, no. 9, pp. 39–48, 1991.

[14] H. Hwang, J. Choi, and H. Jang, "System and method for performing partial array self-refresh operation in a semiconductor memory device," Jun 2003, US Patent App. 10/452,176.

[15] C. Isen and L. John, "ESKIMO - energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *In MICRO*, 2009.

[16] N. Joukov and J. Sipek, "GreenFS: Making enterprise computers greener by protecting them better," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. ACM New York, NY, USA, 2008, pp. 69–80.

[17] A. Karlson, B. Meyers, A. Jacobs, P. Johns, and S. Kane, "Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker," in *PERVASIVE*, 2009.

[18] Y. Katayama, E. 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)*, vol. 311, 1999, p. 318.

[19] J. Kim and M. Papaefthymiou, "Block-based multiperiod dynamic memory design for low data-retention power," *IEEE*

Transactions on Very Large Scale Integration (VLSI) Systems, vol. 11, no. 6, pp. 1006–1018, 2003.

[20] C. Kolb, "Rayshade graphics program." [Online]. Available: http://graphics.stanford.edu/\~cek/rayshade

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *International Symposium on Computer Architecture (ISCA)*, pp. 2–13, 2009.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO*, 1997.

[23] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller, "Energy management for commercial servers," *Computer*, pp. 39–48, 2003.

[24] X. Li and D. Yeung, "Application-Level Correctness and its Impact on Fault Tolerance," in *HPCA*, 2007.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[26] A. Mallik and G. Memik, "A case for clumsy packet processors," in *MICRO*, 2004.

[27] R. Melhem, D. Mossé, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 217–231, 2004.

[28] Micron Technology Inc., "1Gb Mobile LPDDR: MT46H32M32LFCG-5 IT." [Online]. Available: http://www.micron.com/products/partdetail?part=MT46H32M32LFCG-5IT

[29] ——, "System power calculator." [Online]. Available: http://www.micron.com/support/designsupport/tools/powercalc/powercalc.

[30] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 544–553.

[31] ngmoco, "Star defense." [Online]. Available: http://blog.ngmoco.com

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

[33] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: protecting critical data in unsafe languages," in *EuroSys*, 2008.

[34] P. Pop, K. Poulsen, V. Izosimov, and P. Eles, "Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis(CODESS)*. ACM New York, NY, USA, 2007, pp. 233–238.

[35] M. Rinard, A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," Massachusets Institute of Technology (MIT), Tech. Rep., 2009.

[36] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *ICS*, 2006.

[37] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer, "An experimental study of soft errors in microprocessors," *IEEE MICRO*, pp. 30–39, 2005.

[38] J. Satori, J. Sloan, and R. Kumar, "Fluid NMR - performing power/reliability tradeoffs for applications with error tolerance," *Workshop on Power Aware Computing and Systems*, 2009.

[39] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. Corner, and E. Berger, "Eon: a language and runtime system for perpetual systems," in *Conference on Embedded networked sensor systems (SenSys)*. ACM, 2007, pp. 161–174.

[40] SPEC, "SPEC CPU2000." [Online]. Available: http://www.spec.org/cpu2000/

[41] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM," in *HPCA*, 2006.

[42] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks(DSN)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 61.

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

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

[45] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, "GRACE-1: Cross-layer adaptation for multimedia quality and battery energy," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 799–815, 2006.

[46] M. Zhang and N. Shanbhag, "Soft-error-rate-analysis (SERA) methodology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 25, no. 10, pp. 2140–2155, 2006.