# REFRINT: INTELLIGENT REFRESH TO MINIMIZE POWER IN ON-CHIP MULTIPROCESSOR CACHE HIERARCHIES

BY

PRABHAT JAIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Josep Torrellas

# ABSTRACT

As manycores use dynamic energy ever more efficiently, static power consumption becomes a major concern. In particular, in a large manycore running at a low voltage, leakage in on-chip memory modules contributes substantially to the chip's power draw. This is unfortunate given that, intuitively, the large multi-level cache hierarchy of a manycore is likely to contain a lot of useless data.

An effective way to reduce this problem is to use a low-leakage technology such as embedded DRAM (eDRAM). However, such systems require refresh. In this paper, we examine the opportunity of minimizing on-chip memory power by intelligently refreshing a full-eDRAM cache hierarchy. We present *Refrint*, a simple approach to perform fine-grained, intelligent refresh of eDRAM multiprocessor cache hierarchies. We introduce the Refrint algorithms and the microarchitecture support. We evaluate Refrint in a simulated manycore running 16-threaded parallel applications. Compared to a full-SRAM system, *Refrint*'s memory hierarchy only consumes 36% of the SRAM's memory hierarchy energy and induces a negligible slowdown. In contrast, a basic full-eDRAM memory hierarchy consumes 50% of the SRAM's memory hierarchy energy and induces a slowdown of 18%.

*To my parents, for their love and support.*

# ACKNOWLEDGMENTS

I would like to express my deep gratitude to my adviser, Prof. Josep Torrellas, for his guidance and support throughout the course of this work. I thank him for his patience, motivation, and immense knowledge. The complete freedom that he gave me enabled me to experiment with different ideas and put enough thought to the problems before deciding on a solution. Through this work, he has given me a valuable insight into research, and for this I am indebted to him.

I express my heartfelt gratitude to Aditya Agrawal, who has helped me throughout this project. I am grateful to him for all the discussions that I have had with him which have helped shaped the work. I would also like to thank Amin Ansari for having shared his knowledge and suggestions, which helped improve the work.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

While CMOS technology continues to enable higher integration of transistors on a chip, energy and power have emerged as the true constraints for more capable systems. For this reason, there is much interest in techniques for efficient use of energy in chip multiprocessors, such as lower frequencies, simpler cores with extensive clock gating, accelerators and, most recently, renewed interest in ultra low voltages [1].

As chips use dynamic energy more efficiently, however, static power becomes a major concern. For example, in a large manycore running at a low voltage, the fraction of power that is static is substantial, perhaps even dominant. In particular, since memory modules use much (if not most) of the chip area, much of the leakage comes from on-chip memories.

Intuitively, the large multi-level cache hierarchies of a chip multiprocessor are likely to contain much useless (or dead) data. Keeping such data on chip results in unnecessary leakage. For this reason, there are several proposals for power-gating techniques for on-chip memory. They include various forms of dynamically-resizable caches with turned-off ways or sets (e.g., [2, 3]), cache decay [4], and Drowsy caches [5], among others. While these techniques can be effective, they need to be applied in a fine-grained manner to reduce chip leakage substantially — and hence can be expensive.

An alternative approach to reduce on-chip memory leakage is to use a memory technology that, while compatible with a logic process, does not leak by design — or leaks much less. One obvious example is embedded DRAM (eDRAM). Roughly speaking, for the same size in Kbytes as SRAM, eDRAM reduces the leakage power to a quarter or less [6]. A shortcoming is that it needs to be refreshed, and refresh power is significant [7]. However, this fact in turn offers a novel opportunity for power savings, through fine-grained refresh management. Another shortcoming of eDRAM is its lower speed but, arguably, upcoming modest-frequency processors will be able to tolerate this issue.

In this paper, we examine the opportunity of power savings by intelligently refreshing a full-eDRAM cache hierarchy. Our goal is to refresh only the data that will be used in the near future, and invalidate and/or write back to main memory the other data. We present *Refrint*, a simple approach for fine-grained, intelligent refresh of eDRAM lines to minimize on-chip power. We introduce the Refrint algorithms and the microarchitecture support required.

Our results show that Refrint is very effective. We evaluate 16-threaded parallel applications running on a simulated chip multiprocessor with a three-level cache hierarchy. Compared to a full-SRAM system, *Refrint*'s memory hierarchy only consumes 36% of the SRAM's memory hierarchy energy (in L1, L2, L3 and DRAM). In addition, Refrint's early invalidation and write-back of lines does not increase the execution time of the applications noticeably. In contrast, a basic full-eDRAM memory hierarchy consumes 50% of the SRAM's memory hierarchy energy and induces an application execution slowdown of 18%.

This paper is organized as follows: Section 2 provides a motivation; Sections 3 and 4 present the architecture and implementation of Refrint; Sections 5 and 6 evaluate our system; and Section 7 covers related work.

# CHAPTER 2

# MOTIVATION

Current trends suggest a progressive increase in the number of cores per chip in the server market. These cores need to be relatively simple to meet the power and thermal budget requirements of chip designs. In addition, in order to enhance the performance of these cores and also mitigate thermal hot-spots, a large fraction of the chip is regularly devoted to caches — for example, more than 70% in Niagara [8]. At the same time, there is significant interest in reducing the supply voltage of the chip and cycle at more moderate frequencies, to operate in a much more energy-efficient environment [1], with lower dynamic power.

The combination of lower dynamic power and large on-chip caches points to on-chip cache leakage as one of the major contributors to present and future chip power consumption [5]. As a result, there have been multiple proposals for new approaches and technologies to deal with on-chip SRAM leakage. These proposals include new SRAM organizations, embedded DRAM (eDRAM), on-chip flash, and non-volatile memory technologies. Section 7 discusses the work in detail.

One of the most interesting proposals is eDRAM, which has been used by IBM to build the 32MB last level cache (LLCs) of the POWER-7 processor [9]. eDRAM is a capacitor-based dynamic RAM that can be integrated on the same die as the processor. Compared to the SRAM cell, eDRAM has much lower leakage power and a higher density. It also has a lower speed. However, as we explore lower supply voltages and frequencies for energy efficiency, eDRAM may be a very competitive technology.

A major challenge in implementing eDRAM as the building cell of on-chip caches is its refresh power. Since eDRAM is a dynamic cell, as opposed to SRAM, it needs to be refreshed at periodic intervals called *retention periods* to preserve its value and prevent decay. In addition, on an access, the cell automatically gets refreshed, and stays valid for another retention period. Refreshing the cells imposes a significant power consumption cost, and may hurt performance because the cells are unavailable as they are being refreshed.

It is well known that today's large last-level caches of chip multiprocessors contain a lot of useless data. There is, therefore, an opportunity to minimize the refresh power by not refreshing the data that is not useful for the program execution anymore. The challenge is to find which data should be kept alive and which data can be let to decay in an *inexpensive* manner. This is the problem addressed in this paper.

# CHAPTER 3

# REFRINT ARCHITECTURE

## 3.1 Main Idea

Our goal is to consider an on-chip multiprocessor cache hierarchy built exclusively out of eDRAM, and identify and selectively refresh only those cache lines which are expected to be used again in the short and medium term, while letting the rest decay. If our refresh policies are effective, we will only expend refresh energy on those useful lines which are going to be used in the near future, and discard the rest of the lines, by either writing them back to the lower level memory or silently invalidating them from the cache, depending on the cache line state.

If the schemes are over-aggressive, we may end up invalidating a lot of useful data from the caches, thereby having to access the lower level memory a far higher number of times than in an SRAM-based cache hierarchy. Writing back and invalidating a soon-to-be-accessed dirty line has double penalty of invalidating a clean line, as it involves writing back the dirty line and then reading it again. Therefore, our schemes need to be more conservative at handling dirty lines.

We propose *Refrint*, an alternative to a trivial periodic refresh policy of the cache lines by the cache controller. A periodic scheme ends up eagerly refreshing lines, possibly right after the line has been accessed (and automatically refreshed), leading to a higher number of refreshes than needed. In contrast, Refrint maintains a *Sentry Bit* with every cache line, such that it decays faster than the rest of the eDRAM cells in the cache line and acts as a canary, indicating when the line is about to decay and hence needs refresh. This approach minimizees the number of refreshes on the cache line and saves refresh energy.

When the Sentry bit decays, the hardware *interrupts* the cache controller, which either refreshes the line or invalidates it, depending on the state of the line and our policies. Dirty lines that are invalidated are written back to the lower level memory, while clean lines are simply discarded. In shared-memory multiprocessors, cache inclusivity is typically maintained across cache levels, for easy and efficient implementation of the coherence protocol. Consequently, the invalidation of a line in a cache also involves the invalidation of that line in its upper-level caches. This results in extra messages across the network.

In this paper, we focus on simple refresh policies. We do not consider line reuse predictors or similarly elaborate hardware structures. We do not assume that we have information provided by the programmer or software system either. Instead, we focus on refresh policies that consider the state of the line in a multiprocessor hierarchy (valid, dirty, etc.), to decide what to refresh.

## 3.2 Refresh Policies Proposed

A refresh policy has a time- and a data-based component (Table 3.1). The time-based component decides when to refresh. As indicated above, in Refrint, we use an *Interrupt*-based policy, where we associate a Sentry bit per line that decays faster than the line. When it decays, it triggers an interrupt in the cache controller, which indicates that the line is about to decay soon. This induces the refresh of the line. This approach performs the minimum number of refreshes to keep a particular line alive. Every access to a cache line refreshes both the cache line and its Sentry bit. To reduce the hardware implementation complexity, we group the interrupt wires of many individual cache lines into a single interrupt line.

As a reference, we also examine a trivial *Periodic* time-based policy. In this case, the cache controller refreshes lines at periodic intervals equal to the retention period of the eDRAM cells. This approach is cheap because it requires no Sentry bit — it only needs a global counter for the whole cache. However, it is conservative, resulting in more refreshes than necessary, as it may eagerly refresh a line much before it is about to decay. Also, it can render the cache unavailable for a continuous period of time when the lines are being refreshed. To avoid bunching up the refresh operations in the

| Time-based policies: *When ?* | |
|---|---|
| Periodic | Refresh periodically (a group of lines at a time) |
| Refrint | Refresh on Sentry bit decay (a group of lines at a time) |
| **Data-based policies: *What ?*** | |
| All | All lines are refreshed |
| Valid | Only Valid lines are refreshed |
| Dirty | Only Dirty lines are refreshed |
| WB(n,m) | Dirty lines are refreshed $n$ times before writeback, while Valid lines are refreshed $m$ times before invalidation |

Table 3.1: Refresh policies proposed.

periodic scheme, we refresh lines in groups at a time. Specifically, the refresh of a full cache is staggered across an entire retention period.

Our proposed data-based policies are shown in Table 3.1. Either of the time-based policies can be combined with any of the data-based policies, which decide what to refresh. There are many possible approaches to build these policies. In this paper, we explore very simple approaches that are based on the state of the line in the cache.

Specifically, we compare four data-based policies: *All*, *Valid*, *Dirty*, and *WB(n,m)* (write back). *All* refreshes every cache line, irrespective of whether it is valid or not. We only evaluate this policy for reference purposes. *Valid* and *Dirty* policies always refresh Valid and Dirty cache lines respectively, and invalidate the line otherwise.

The *WB* policy is associated with a tuple *(n,m)*. This policy refreshes a Dirty line (that is not being accessed) for $n$ times before writing it back and changing its state to Valid Clean; it refreshes a Valid Clean line (that is not being accessed) for $m$ times before invalidating it. We maintain a Dirty line in the cache for longer because evicting it has the additional cost of writing back the data to lower-level memory. It is, therefore, worthwhile to keep it in the cache for longer. To implement this policy, we maintain a per-line *Count*. Count is initially set to a reference value. It is then decremented every time that the Sentry bit decays and the line is refreshed. On any normal, non-refresh access to the line, Count is reset to its reference value. Note that the *Dirty* policy is equivalent to *WB($\infty$,0)*, while *Valid* is equivalent to *WB($\infty$,$\infty$)*. Finally, every policy refreshes cache lines in transient states as well.

Using cache states has the advantage that the hardware support needed is simple. A disadvantage is that the policies are unable to disambiguate lines that, within the same state, behave differently. In addition, these policies interact with the cache coherence protocol and the inclusivity requirements of multilevel caches in a non-trivial manner. For example, a Valid Clean line in a shared L3 may or may not be Dirty in upper private layers of the cache hierarchy; in either case, the algorithm will treat it in the same way. As another example already mentioned, if the policy decides to invalidate a line in L3, it has to also invalidate that line in L2 and L1.

The periodic policy can also be combined with the data-based policies. A naive eDRAM implementation periodically refreshes all the lines in the cache. Consequently, we use *Periodic All* as the baseline implementation for eDRAM caches. A slightly smarter and natural extension is the *Periodic Valid* policy.

## 3.3 Application Data Access Patterns

Intelligent refresh policies try to evict from the on-chip memory the data that is not useful — i.e., the data that will not be accessed anymore or in a long while. Intuitively, given a chip with a large on-chip memory, it is likely that there is a significant amount of such useless data. In practice, however, every application has a different access pattern, and is not easy to identify useless data.

Figure 3.1 presents a high-level categorization of applications with respect to our data-based refresh policies. It corresponds to a cache coherent multiprocessor with an inclusive cache hierarchy, where the last level of cache is shared by all the cores. This is a common design. We are interested in observing the events from the point of view of the last level cache, which is the one that matters the most in terms of the refresh energy.

We consider two axes:

**a**) The first one is the size of the application footprint relative to the size of the last level cache. Since an application can only access the data at a given maximum rate, it is likely that applications that have a large data footprint will have long time intervals between reuse of the data, if data is reused at all. Hence, such data can be safely displaced and, if reused, can be brought back again. Therefore, the best policy for such applications should be a general

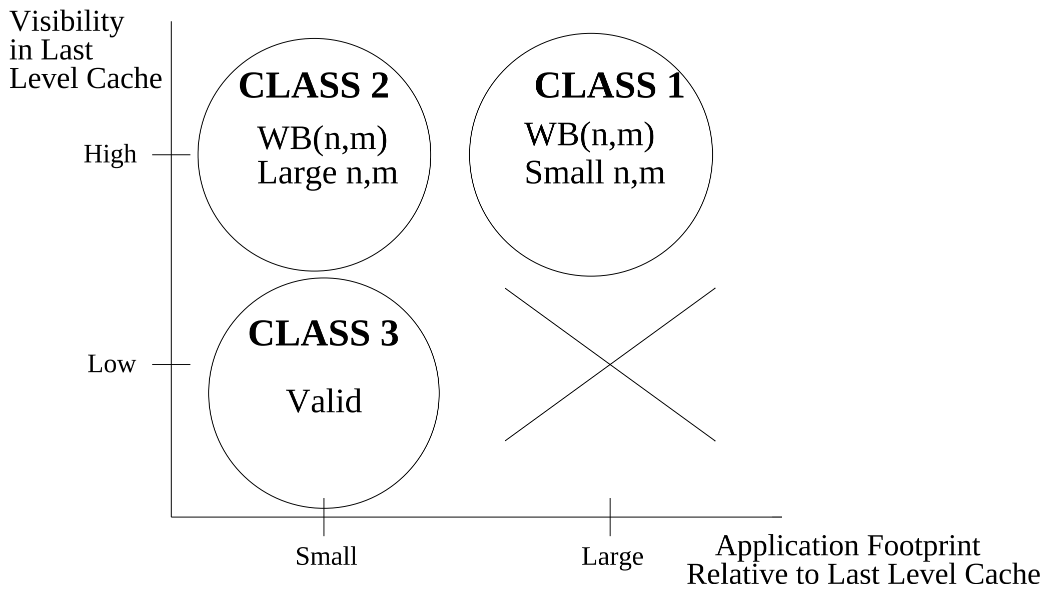
Visibility in Last Level Cache
High
Low
CLASS 2
WB(n,m)
Large n,m
CLASS 1
WB(n,m)
Small n,m
CLASS 3
Valid
Small
Large
Application Footprint Relative to Last Level Cache

Figure 3.1: Application data access patterns from the point of view of the last level of on-chip cache.

one, namely WB(n,m), with a *small* (n,m). Indeed, after an initial flurry of accesses, the data can soon be invalidated and evicted from the cache. On the other hand, for small footprint applications, the processors are likely to reuse the data more often. Therefore, a general policy such as WB(n,m), with a *large* (n,m), is likely to be useful. Data will be reused and, therefore, should be kept in the on-chip memory.
**b)** The second axis captures whether the last level cache has "visibility" on the activity of the lines in the upper levels of the cache hierarchy. For example, assume that we have the directory in the L3. Assume that the working set is such that: (i) it largely fits in the L1 and L2 caches and hence, there is no overflow and, (ii) there is little data sharing between processors and hence, the last level cache does not see the data moving back and forth between caches and its associated state transitions from dirty to shared. In this case, visibility is low. Therefore, we need to be conservative and assume that the data is being repeatedly accessed in the L1 and L2 caches. Hence, the conservative Valid scheme should be best for such applications, as it avoids invalidating data that could potentially be heavily reused in upper-level caches.

Even with a small data footprint, if there is high data sharing across upper-level caches, such that data is frequently written by a processor and read by another, then visibility at L3 is high. There are frequent writebacks due to transitions from *Dirty* to *Shared*. In such cases, the more specific WB(n,m) should do better than Valid.

We refer to the three classes of applications described as Class 1, Class 2, and Class 3 applications, respectively. In our evaluation (Section 6), we will show that the results of our experiments confirm this model. We do not find any application of the type Large footprint and Low visibility. We find that all our large-footprint applications also have substantial data sharing between the caches in the upper levels, or that dirty data is often evicted from upper level caches and written back to L3, therefore providing visibility.

# CHAPTER 4

# IMPLEMENTATION ISSUES

## 4.1 Sentry Bit Design

The Sentry bit needs to be designed conservatively, such that it decays before the rest of the cells in its corresponding cache line. This is done by implementing it as a 1T-1C eDRAM cell with a capacitance lower than that of the cells in its line, similar to the Valid bit suggested in [10]. In addition, to minimize the effects of process variation, the Sentry bit should be placed in physical proximity of its corresponding cache line, so that at least the systematic component of variation is the same. In this paper, however, we do not consider the effects of process variation.

Furthermore, the Sentry bit needs to be designed such that it decays at least as many cycles faster than the rest of the eDRAM cells in its cache line *as the maximum number of Sentry bits that can fire simultaneously.* This is needed to guarantee timely and successful refresh of every cache line even in the most pessimistic case of the maximum possible number of Sentry bits firing together. In our evaluations, we take the most conservative approach and assume that all the Sentry bits in a cache can fire in a given cycle. However, that need not be the case. If the chip fabrication process, or the post silicon testing process can guarantee/determine the timing variation amongst the Sentry bit cells, then a better bound on the Sentry bit retention period can be deduced. For example, if the difference between the retention times of the Sentry bit with the longest retention time and the Sentry bit with the shortest retention time can be determined to be, say $\Delta$ cycles, then a simple and conservative upper bound on the number of Sentry bits that can fire together would be $\Delta$, assuming that the cache supports access to only one cache line in a given cycle (single- ported cache). This could, in turn, enable further significant savings in refresh energy as a line would be refreshed fewer times in a given time period.

The retention period of the Sentry bit defines the *refresh* period of the cache line. Note that this reduces the refresh period of a cache line compared to the trivial Periodic refresh scheme. Owing to the deterministic nature of refresh in a Periodic scheme, a line need not be refreshed any sooner than its retention period. In our evaluations, where each L3 cache has 16K lines, we assume the retention period of the Sentry bit to be 16 $\mu$s (@1GHz) less than that of rest of the eDRAM cells. For a normal eDRAM cell retention time of 50 $\mu$s, we lose a significant 32% (16/50) opportunity as compared to a Periodic Scheme. However, despite this disadvantage, our Refrint schemes outperform the Periodic schemes, both in performance and energy, due to reasons cited in Section 3.1.

When the Sentry bit decays, it interrupts the cache controller triggering an action on the line. Figure 4.1 shows the algorithm used by the WB(n,m) scheme, when a Sentry bit interrupt occurs. When the line is accessed with a normal read or write, if it is dirty, Count is set to $n$ while, if it is clean, it is set to $m$. Note, in All, Valid and Dirty schemes, Count is not needed as the state of the line is sufficient to decide the action.

```
Read the line's Count
If (Count >=1)
    Refresh line and Sentry bit
    Decrement Count
Else If (Count==0 && line is Dirty)
    Write back the line, change its state to Valid Clean
    Refresh Sentry bit
    /* Writeback automatically refreshed the line */
    Reset Count to "m"
Else If (Count==0 && line is Valid Clean)
    Invalidate the line
    /* Requires invalidation in upper level caches too */
    /* Do not refresh line or Sentry bit */
```

Figure 4.1: Hardware operations in WB(n,m) when a Sentry bit interrupt occurs.

To reduce the logic complexity and the number of wires feeding into the cache controller, all the interrupt wires are first input into a *priority encoder*, the far fewer output wires of which, provide input to the cache controller (Figure 4.2). A simple encoder will not suffice as many Sentry bits can fire at the same time. A priority encoder, instead, would serialize the interrupts. The design of the interrupt handling logic, as explained later, and the time margin between the retention periods of the cache line and the Sentry bit ensure that no line is ever starved and every line is timely processed before it expires. Moreover, since the firing of Sentry bits is so spread out, no line is likely to be forced to wait for a significant amount of time before being serviced, as also shown by our simulations.

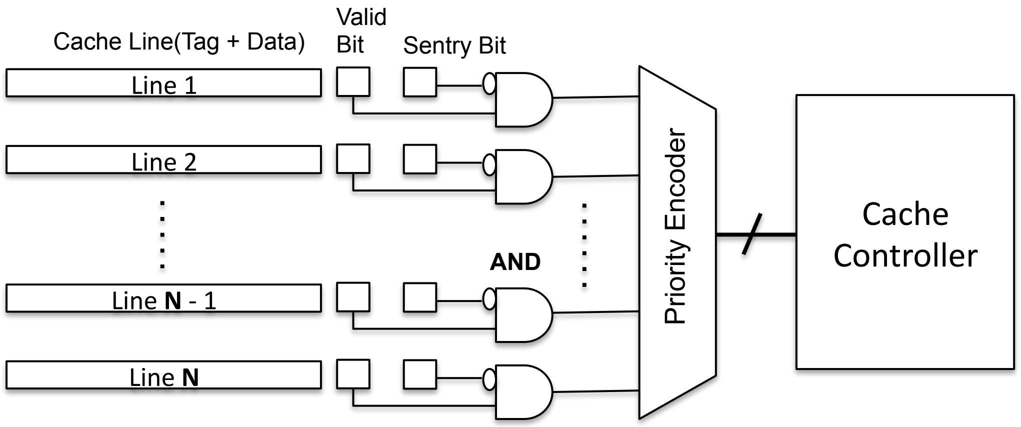

Figure 4.2: Sentry bit Interrupt logic with a priority encoder.

A priority encoder could be built in a similar fashion as a decoder used in caches to select a row. The priority logic would add some more complexity and area, but the area would be greatly dominated by the number of wires, and so it would be similar to a cache decoder. We improve and further simplify the interrupt logic design by grouping many Sentry bit wires into a single interrupt line.

Figure 4.2 represents the hardware sentry bit logic to implement Valid, Dirty, and WB(n,m) policies (not the All policy). We only evaluate the All policy for completeness and reference purposes.

## 4.2 Processing Refresh Interrupts

The Sentry bit interrupt takes priority over the normal R/W requests queued in the cache controller that are yet to be processed. Even if multiple interrupt lines go high simultaneously, they are processed sequentially by the priority encoder, and hence the cache controller. Since the cache is pipelined, a new interrupt request can be processed every cycle. No plain R/W requests are serviced in a given cycle if there are interrupt requests pending as well.

The following actions are sequentially performed by the interrupt processing logic depicted in Figure 4.3 on getting an interrupt request. In case of grouping of sentry bits, the following will be done for each and every line in the group in a pipelined way:

1. Reset the Sentry bit of the interrupting line so that the priority encoder can output another interrupting line, if any, in the next cycle.

2. Read the *Count* and *State* of the line into the "Decision Logic" and decide whether to refresh the line or invalidate it depending on the line State, Count, and the refresh policy(WB, Dirty, or Valid).

3. Based on the output of the previous step, either refresh the line, or invalidate it. Invalidation can involve sending invalidates to upper level caches and this would be initiated by the cache controller. The exact steps are shown in Figure 4.1.

The per-line Count can be maintained as a few extra memory bit cells along with the *tag* bits of the cache line. As on every Sentry bit decay of a line, the line needs to processed by the cache controller, either to be refreshed or invalidated, and hence, Count can be explicitly processed by the cache controller. There is no need for a logic based counter, which would have a significant transistor overhead. In our implementation, a 5-bit Count needs only 5 extra eDRAM cells, which has a negligible overhead compared to a 512b cache line and its associated tag and state bits.

Note, we do not introduce an additional queue for buffering the interrupt requests. The interrupt requests are automatically serialized by virtue of the priority encoder. As shown in Figure 4.3, we just need a single buffer to enable the pipelining effect.

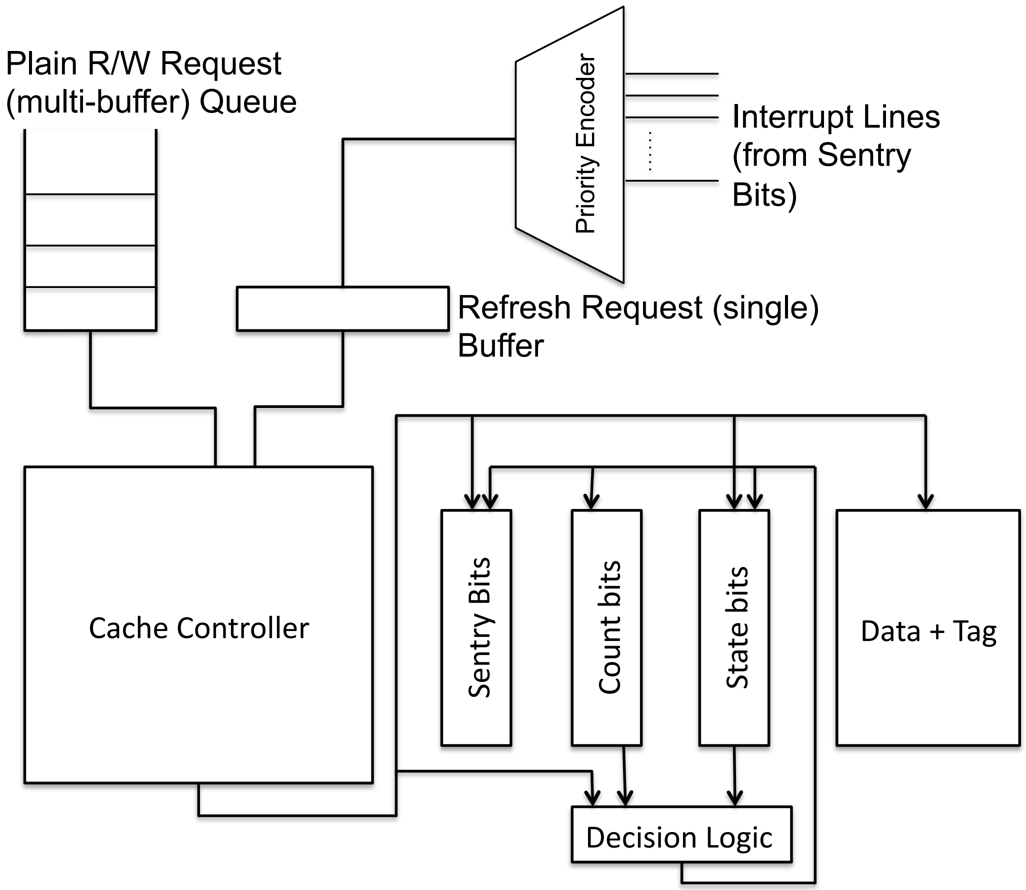
Plain R/W Request (multi-buffer) Queue
Priority Encoder
Interrupt Lines (from Sentry Bits)
Refresh Request (single) Buffer
Cache Controller
Sentry Bits
Count bits
State bits
Data + Tag
Decision Logic

Figure 4.3: Logic to process the refresh interrupts.

# CHAPTER 5

# EXPERIMENTAL SETUP

We evaluate Refrint on a 16 core chip multiprocessor (CMP) system. Each core is a dual issue, out-of-order (OOO) processor running the MIPS32 instruction set. Each core has a private instruction cache (IL1), a private data cache (DL1) and a private second level (L2) cache. The 16 cores are connected through a 4x4 torus network. A shared third level (L3) cache is divided into 16 banks and each bank is connected to a vertex of the torus network. The addresses are statically mapped to the banks of the L3. Each bank has a dedicated logic to process refresh interrupts as shown in Fig. 4.3. We employ a directory MESI coherence protocol. The directory is maintained at L3. The architectural parameters are summarized in Table 5.1.

We model our architecture (cores, caches and network) in the publicly available cycle accurate simulator, SESC [11]. The timing, dynamic energy and leakage power numbers for cores and network were obtained from McPAT [12], while for SRAMs and eDRAMs, they were obtained from CACTI [13]. Even though McPAT uses CACTI internally, it does not allow for an eDRAM memory hierarchy. Hence, we had to use CACTI as a standalone tool. We experiment with 3 different values of retention times: 50$\mu$s, 100$\mu$s, and 200$\mu$s. Barth et al. [14] report a retention time of 40$\mu$s for eDRAM cells at 105°C. The retention time has an exponential dependence on temperature [15]. In this paper, we target a low-voltage, low-frequency, simple-core and energy-efficient futuristic architecture for which the temperatures would be significantly lower than 105°C, and hence, we conduct experiments at the above mentioned three retention times. Other experimental parameters like temperature, frequency etc. are also summarized in Table 5.1.

In this paper, we compare a full-SRAM memory hierarchy (baseline) to a full-eDRAM memory hierarchy (proposed). To do a fair and simple comparison between the two, we have made a few simplifying decisions, which are listed in Table 5.2. Since, our chip frequency is low enough (1000 MHz), we

| Architectural Parameters | |
|---|---|
| Chip | 16 core CMP |
| Core | MIPS32, 2 issue OOO processor |
| Instruction L1 | 32 KB, 2 way, private |
| | Access time: 1 ns |
| Data L1 | 32 KB, 4 way, WT, private |
| | Access time: 1 ns |
| L2 | 256 KB, 8 way, WB, private |
| | Access time: 2 ns |
| L3 | 1 MB per bank, 16 banks, 8 way, WB, shared |
| | Access time: 4 ns |
| Line size | 64 Bytes |
| DRAM | Access time: 40ns |
| Network | 4 x 4 torus |
| Coherence | Directory MESI protocol at L3 |
| **Technology Parameters** | |
| Technology node | 32 nm |
| Frequency | 1000 MHz |
| Device type | LOP (Low operating power) |
| Temperature | 330 Kelvin |
| **Tools** | |
| Architecture | SESC [11] |
| Timing & Power | McPAT [12] & CACTI [13] |

Table 5.1: Evaluation architecture & tools.

were able to achieve same latency numbers for SRAM and eDRAM for all levels of the cache hierarchy. We assume the access energies to be the same. In addition, we assume that the refresh energy of a line is equal to the access energy of the line and that a line can be refreshed in a cycle, when done in a pipelined fashion.

If, at higher frequencies, or due to other technological constraints, it is not viable to implement L1 caches with eDRAM cells, then L1s could be implemented with normal SRAM cells. Most of the energy expended in L1 is dynamic energy ($\sim$ 90 %). The fraction of refresh energy in L1 is $\sim$ 1 %. Therefore, there are minimal refresh energy savings to be leveraged from L1 caches, and so, our conclusions would still remain the same even if L1s are implemented with SRAM technology.

We evaluate Refrint by running 16-threaded parallel applications, available from the SPLASH-2 [16] and PARSEC benchmark suites. The set of applications and the problem sizes are summarized in Table 5.3. Each ap-

|  | Baseline | Proposed |
|---|---|---|
| Cell | SRAM | eDRAM |
| Access time (ratio) | 1 | 1 |
| Access energy (ratio) | 1 | 1 |
| Leakage power (ratio) | 1 | 1/4 |
| Refresh time | NA | access time |
| Refresh energy | NA | access energy |

Table 5.2: Baseline and proposed architecture.

plication was run at 3 retention times, 2 timing policies, 7 data policies and the baseline case amounting to a total of 43 (42 + 1) combinations. The parameter sweep is summarized in Table 5.4. For periodic scheme, we club lines into as many groups as the number of sub arrays per bank reported by CACTI. Therefore, for L1 we have 4 groups of 128 lines each, for L2 we have 4 groups of 1024 lines each and for L3 we have 4 groups of 4096 lines each. For Refrint, we group sentry bits so that we have a maximum of 1024 inputs to the priority encoder. We have a group size of 1 (512 inputs to the encoder) for L1, for L2 we have a group size of 4 (1024 inputs to the encoder) and for L3 we have a group size of 16 (1024 inputs to the encoder).

| SPLASH-2 [16] |  |
|---|---|
| FFT | $2^{20}$ |
| LU | 512 x 512 |
| Radix | 2M keys |
| Cholesky | tk29.O |
| Barnes | 16 K particles |
| FMM | 16 K |
| Radiosity | batch |
| Raytrace | teapot |
| **PARSEC** |  |
| Streamcluster | sim small |
| Blackscholes | sim medium |
| Fluidanimate | sim small |

Table 5.3: Applications.

| | | |
|---|---|---|
| Retention time | 50 $\mu$s, 100 $\mu$s, 200 $\mu$s | 3 |
| Timing policy | Periodic, Refrint | 2 |
| Data policy | All, Valid, Dirty | |
| | WB(4,4), WB(8,8), WB(16,16), WB(32,32) | 7 |
| | Total combinations | 42 |

Table 5.4: Parameter sweep of policies.

# CHAPTER 6

# EVALUATION

In this section, we present our evaluation of Refrint. We present the effect of our policies on memory hierarchy energy, total energy and execution time.

**Policies:** We present results for *Periodic* and *Refrint*. The two data-based policies viz. *All* and *Valid* do not create extra DRAM accesses. However, *Dirty* and *WB(n,m)* (write back) policies create extra DRAM accesses by either discarding valid data or pushing dirty data to DRAM to save on-chip refresh energy. Therefore, to do a fair comparison, we take DRAM access energy into account. In addition, we assume that at the end of the simulation all dirty data will be written back to main memory (DRAM).

**Applications:** In section 3.3 and in Fig. 3.1 we categorized applications into three classes. In the course of our evaluation we found that applications within a class responded similarly to our timing and data policies. Table 6.1 shows the binning for our set of applications. In the following sections, rather than pick one representative application from each class, we will present average numbers for the entire class.

| Category | Applications |
|---|---|
| Class 1 | FFT, FMM, Cholesky, Fluidanimate |
| Class 2 | Barnes, LU, Radix, Radiosity |
| Class 3 | Blackscholes, Streamcluster, Raytrace |

Table 6.1: Application Binning.

## 6.1 Memory Hierarchy Energy

In this section, we present the effects of our policies on the memory hierarchy energy. We split the memory energy in two different ways, the first as a sum of L1, L2, L3 and DRAM energies 6.2 and the second as a sum of on-chip dynamic, leakage, refresh and DRAM energies 6.3.

## 6.2 L1, L2, L3 & DRAM

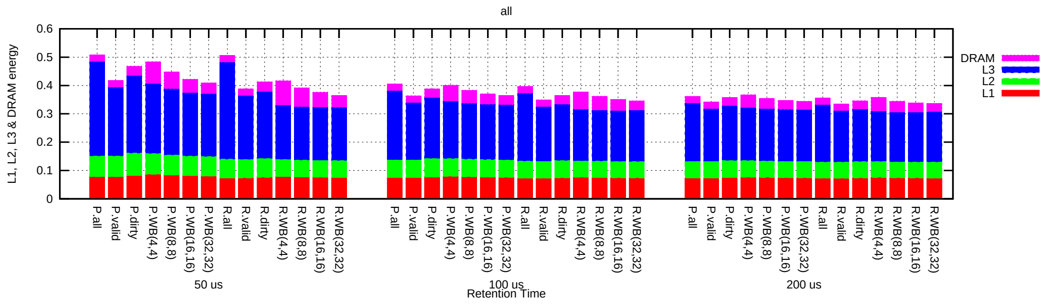

Figure 6.1: L1, L2, L3 & DRAM energy (normalized to full-SRAM memory energy).

In Fig. 6.1 we show memory energy as the sum of L1, L2, L3 and DRAM energies (averaged over all applications). On the X-axis we have 3 sets of bars, each at retention times of 50 $\mu$s, 100 $\mu$s and 200 $\mu$s respectively. Within each retention time we have 2 time-based policies viz. periodic (P) and Refrint (R). For each time-based policy we have 7 data-based policies viz. *All, Valid, Dirty, WB(4,4), WB(8,8), WB(16,16)* and *WB(32,32)*. The bars are labelled as 'time-policy.data-policy' e.g., P.WB(4,4) stands for periodic and WB(4,4) policy. The Y-axis represents total memory energy as a sum of L1, L2, L3 and DRAM energies respectively, from bottom to top.

Across all retention times and policies, we find that L3 consumes the majority ($\sim$ 60 %) of the on-chip memory energy. This was expected, and so we focussed our evaluation on the effect of our policies in saving L3 energy. In each of the 42 combinations, all on-chip memories (L1, L2 and L3) have the same timing policies i.e. either Periodic or Refrint. However, L1 and L2 were always run at the *Valid* data-policy, which refreshes all the valid lines in the cache. This was done because L1 and L2 caches have very high dynamic energy and small leakage and refresh energy components. Therefore, there are minimal refresh energy savings to be gained. In addition, any line which is not being used is quickly replaced by the normal cache replacement policies. Even with this decision of applying intelligent refresh only at L3, we are able to save significant memory energy, as can be seen from the plot.

To show the effectivness of our polices in saving on chip refresh energy, we present the same data (memory energy), but now as a sum of on-chip dynamic, leakage, refresh and DRAM energies.

## 6.3 On-Chip Dynamic, Leakage, Refresh & DRAM

In Fig. 6.2 we show the memory energy (averaged) for Class 1, Class 2 and Class 3 applications as the sum of on-chip dynamic, leakage, refresh energies and DRAM energy. The fourth plot (labelled 'all') shows the average distribution over all applications. The X-axis of the plots are the same as in section 6.2. The Y-axis represents total memory energy as a sum of on-chip dynamic, leakage, refresh energies and DRAM energy respectively, from bottom to top.

In all classes of applications, the fraction of dynamic energy remains almost the same across retention times and across policies because the amount of work done is the same. The fraction of leakage energy varies because of the effect of the policies on execution time (Sec. 6.5). The main variation is in the fraction of refresh energy, which is the focus of this paper. The reduction in on-chip refresh energy (as a result of policies) comes at the cost of extra DRAM accesses, and has been taken into account.

**Retention Time:** As the retention time increases, the lines have to be refreshed less often and hence the fraction of refresh energy reduces. The effect of the policies (timing and data) are most pronounced at smaller retention times.

**Timing Policies:** Refrint policies always do better than Periodic. This is because the refreshing of lines is highly staggered and is done only when the line truly needs one. Periodic schemes, on the other hand, block the cache more often leading to increased execution times, hence leakage and refresh the line even if not required.

**Data Policies:** The effect of data policies is different in the three classes of applications. In Class 1 applications (high footprint, high visibility), *WB(n,m)* policies even at small values of (n,m) are very effective and significantly reduce the refresh component and the total memory energy in comparison to *All, Valid* and *Dirty* policies. In Class 2 applications (low footprint, high visibility), *WB(n,m)* policies are still effective but only at high values of (n,m). *Valid* scheme does equally well for such applications. In Class 3 applications (low footprint, low visibility), any policy *Dirty* or *WB(n,m)* which attempts to reduce refresh energy pays a penalty in terms of leakage energy (due to increased execution time) or DRAM energy. *Valid* scheme does best for this class of applications. Our observations are in line with our hypothesis presented in Sec. 3.3.

Across all applications, Refrint schemes do better than Periodic schemes. On average, *WB(32,32)* policy does better than all other policies. At 50 $\mu$s, on average, the base refresh policy for eDRAM (Periodic All) consumes 50 % energy compared to a full-SRAM memory hierarchy. Our Refrint WB(32,32) policy reduces the memory hierarchy energy to 36 % compared to a full-SRAM memory hierarchy.

## 6.4 Total Energy

In Fig. 6.3 we show the normalized total system energy (cores, caches, network and DRAM energy), averaged over Class 1 applications and over all applications. The X-axis of the plots is the same as in section 6.2. The Y-axis represents the total system energy. For Class 1 applications, Refrint *WB(32,32)* policy does the best. On average, across all applications, Refrint *WB(32,32)* still does the best although, Refrint *Valid* policy comes close enough. For emerging classes of high footprint applications, we expect Refrint *WB(32,32)* policy to bring in the most energy savings. At 50 $\mu$s, on average, the base refresh policy for eDRAM (Periodic All) consumes 72 % energy compared to a full-SRAM system. Our Refrint *WB(32,32)* policy reduces the system energy to 61 % compared to a full-SRAM system.

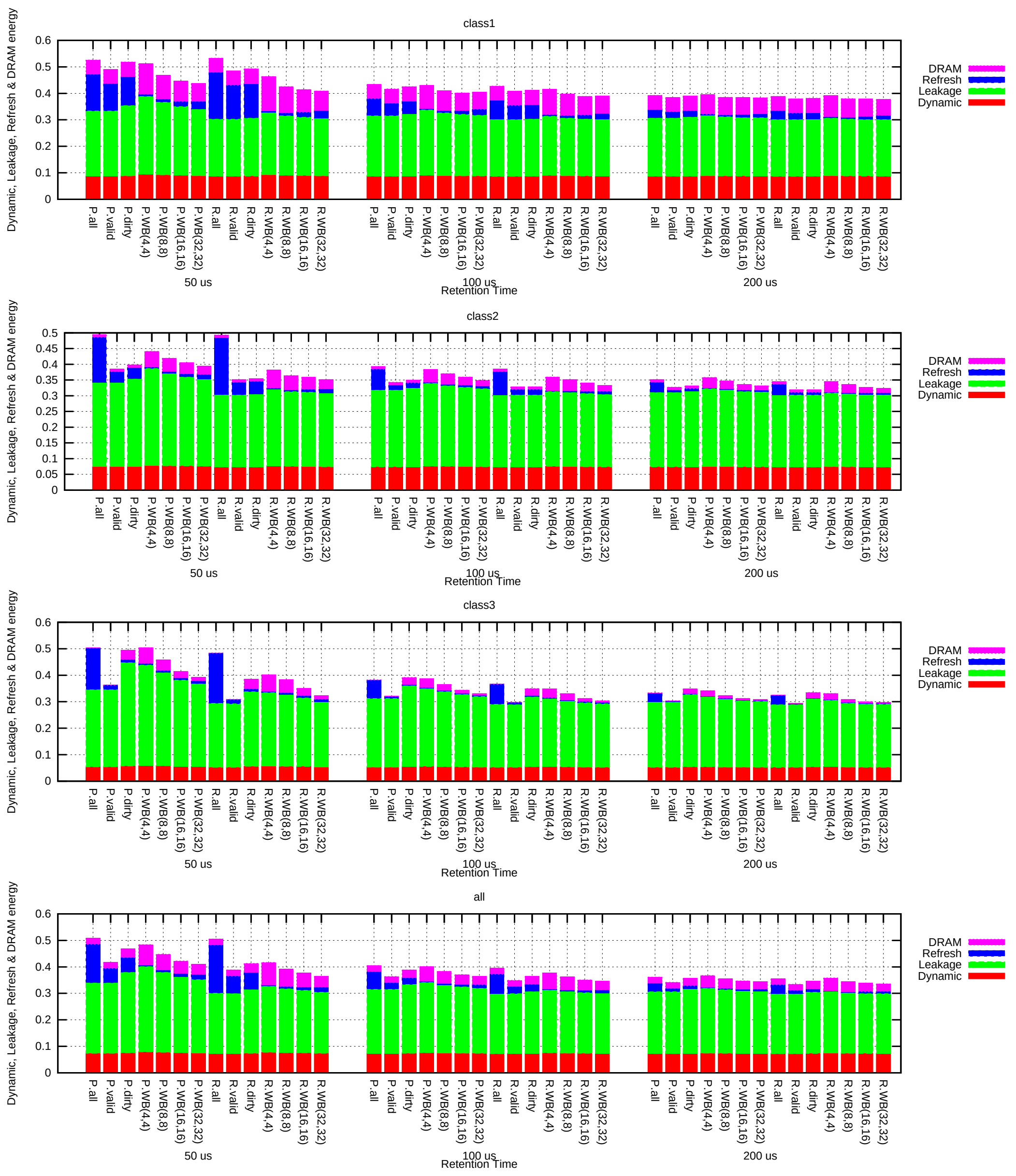

Figure 6.2: On chip dynamic, leakage, refresh & DRAM energy (normalized to full-SRAM memory energy).

## 6.5 Execution Time

In Fig. 6.4 we show the normalized execution time averaged over Class 1 applications and over all applications. The X-axis of the plots are the same as in section 6.2. The Y-axis represents total execution time. All applications have the same trend across all retention times and policies. On average, with increasing retention times, the performance penalty reduces.

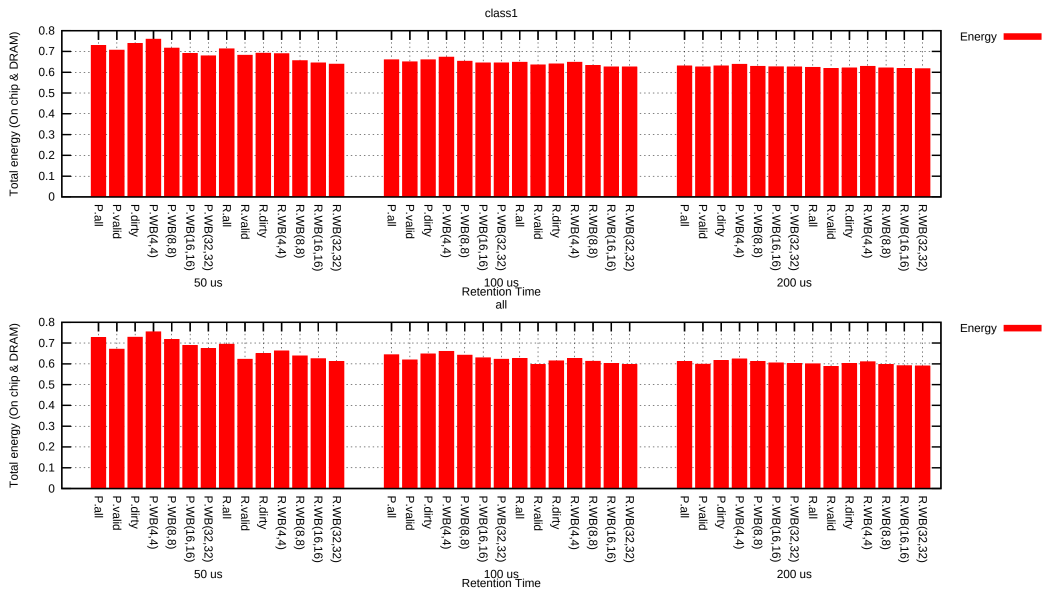

Figure 6.3: Total energy (normalized to full-SRAM system energy).

**Timing Policies:** Periodic schemes do much worse in comparison to Refrint schemes. This is because periodic schemes block the cache while the lines are being refreshed. Also, by their nature of being periodic they refresh the lines more often than needed. Refrint schemes, on the other hand, refresh lines in a very staggered fashion and do not cause significant performance degradation.

**Data Policies:** *All* and *Valid* data policies are the best w.r.t. execution time as they keep all data in the cache do not create any extra invalidations or writebacks compared to full-SRAM memory hierarchy. *Dirty* and *WB(n,m)* policies incur a performance overhead, due to an increase in miss rates caused by extra invalidations and writebacks. The performance penalty for *WB(n,m)* policy goes down as $(n, m)$ grow. This is obvious, as the data is kept around for a longer time.

At 50 $\mu$s, on average, the base refresh policy for eDRAM (Periodic All) suffers a slowdown of 18 % compared to a full-SRAM system. Our Refrint *WB(32,32)* policy suffers a slow down of only 2 % compared to a full-SRAM system.

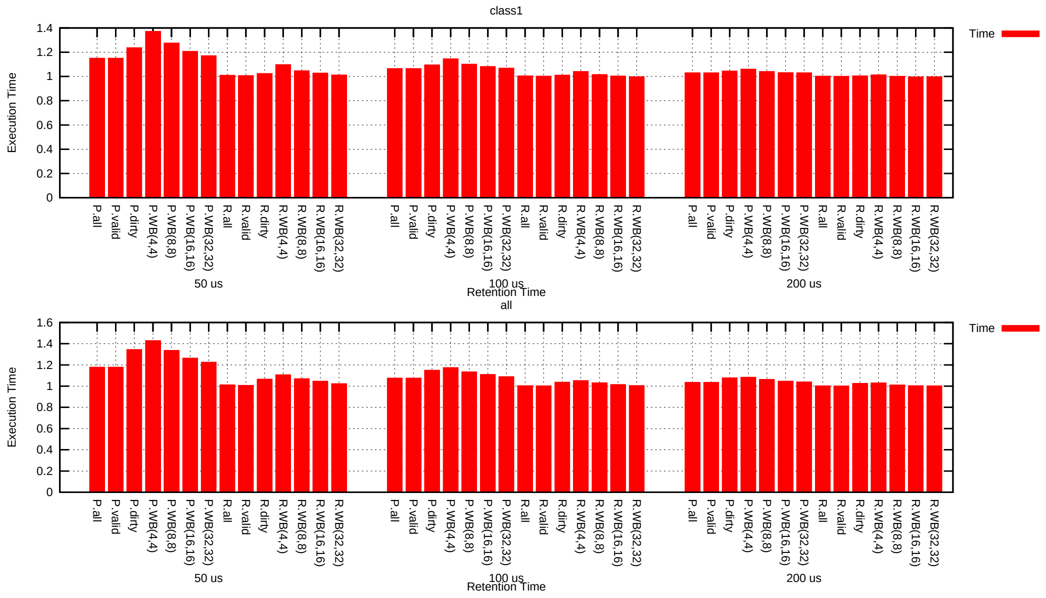

Figure 6.4: Execution time (normalized to full-SRAM execution time).

# CHAPTER 7

# RELATED WORK

To reduce leakage power in the conventional SRAM caches, two main approaches have been proposed. They differ in their ability to preseve state. Gated-Vdd [17] and Cache Decay [4, 18] turn off cache lines that are not likely to be accessed in the near future, and thereby save leakage power. Drowsy Caches [5, 19], periodically move inactive lines to a low power mode in which they cannot be read or written.

Nevertheless, both approaches targetting SRAMs, require design changes, power gating/voltage biasing circuitry and have a non-negligible hardware overhead.

Emerging memory technologies like eDRAM, the focus of our work, offer great leakage reduction and area benefits, and are a promising memory technology of the future.

eDRAM 1T-1C cells can be logic-based or DRAM based. Logic-based eDRAM operates faster but it is more expensive as it complicates the logic manufacturing process. Logic-based eDRAM, as a feasible alternative to on-chip SRAMs, has been proposed in [20]. To make eDRAM characteristics closer to SRAM in terms of delay and to simplify the process technology, Liang et al. [21] proposed the 3T-1D eDRAM cell for L1 caches. The proposed cell consists of three transistors and a diode which loses its charge over time, thereby requiring refresh. Juang et al. [22] proposed a dynamic cell with 4 transistors by removing two transistors that restore the charge loss in a conventional 6T SRAM cell. The 4T cell requires less area compared to 6T SRAM while achieving almost the same performance. However, the data access is slower and destructive, which can be solved by refreshing the data.

Hybrid memory cells have also been proposed to take advantage of the different features that different memory cells offer. Valero et al. [10] introduced a macro-cell that combines SRAM and eDRAM at cell-level. They implement a N-way set-associative cache with these macro-cells consisting of one SRAM cell, N-1 eDRAM cells, and a transistor that acts as a bridge to move data from the static cell to the dynamic ones. Although applicable to first-level caches, this approach is not effective for large shared caches, since the access patterns are not so predictable and the data access characteristics at L1 caches do not hold true at lower level caches.

In deep sub-micron technology nodes, when implementing an eDRAM-based on-chip cache, power consumption and the performance overhead of refreshing the eDRAM cells become the main bottlenecks. An interesting approach is introduced by Venkatesan, et al. [23] which is a software-based mechanism that allocates blocks with longer retention time before allocating the ones with a shorter retention time. Using this technique, the refresh period of the whole cache is determined only by the portion instead of the entire cache. Ghosh et al. [24] proposed SmartRefresh that reduces refresh power by adding timeout counters per line. This avoids unnecessary refreshes of the lines which were recently read or written.

Usage of error-correction codes (ECC) is another technique to reduce the refresh power [25]. ECC can tolerate some failures and hence, allows setting the global refresh time irrespective of the weakest cells. This means that employing a stronger ECC can increase the refresh period and reduce the refresh energy. Nonetheless, strong codes come with high overheads in terms of storage, encoding/decoding power, area, and complexity.

# CHAPTER 8

# CONCLUSIONS

To reduce the power consumed by large chip multiprocessors in the cache hierarchy, this paper considered a low-leakage technology (eDRAM) and examined intelligently refreshing it for power savings. Our goal was to refresh the data that will be used in the near future, and invalidate and/or write back to memory the other data. We presented *Refrint*, a simple approach for fine-grained, intelligent refresh of eDRAM lines to minimize on-chip power. We introduced the Refrint algorithms and the microarchitecture support required.

We evaluated 16-threaded parallel applications running on a representative chip multiprocessor with a three-level cache hierarchy. Our results showed that Refrint is very effective. Compared to a full-SRAM system, a basic full-eDRAM system consumes 50 % memory (L1, L2, L3 and DRAM) energy and 72 % system energy. In comparison, Refrint consumes 36 % memory energy and 61 % system energy. In addition, in the energy remaining, the contribution of refreshes is negligible. Finally, Refrint's early invalidation and write back of lines does not increase the execution time of the applications noticeably. The basic full-eDRAM suffers a slowdown of 18 %, while Refrint slows down performance by only 2 %

# REFERENCES

[1] R. G. Dreslinski et al., "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proc. of the IEEE*, no. 2, pp. 253–266, Feb. 2010.

[2] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," in *HPCA*, 2001, pp. 147 –157.

[3] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *HPCA*, Feb. 2002, pp. 151 – 161.

[4] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *ISCA*, 2001, pp. 240–251.

[5] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA*, 2002, pp. 148–157.

[6] S. S. Iyer et al., "Embedded dram: Technology platform for the blue gene/l chip," *IBM Journal of Research and Development*, vol. 49, no. 2-3, pp. 333–350, 2005.

[7] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-L. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ISCA*, 2010, pp. 83–93.

[8] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[9] D. F. Wendel et al., "Power7: A highly parallel, scalable multi-core high end server processor," *J. Solid-State Circuits*, vol. 46, no. 1, pp. 145–161, 2011.

[10] A. Valero, J. Sahuquillo, S. Petit, V. Lorente, R. Canal, P. López, and J. Duato, "An hybrid edram/sram macrocell to implement first-level data caches," in *MICRO*, 2009, pp. 213–221.

[11] J. Renau et al., “SESC simulator,” January 2005, http://sesc.sourceforge.net.

[12] S. Li et al., “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *MICRO*, 2009, pp. 469–480.

[13] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi, “CACTI 5.1. Technical Report,” Hewlett Packard Labs, Tech. Rep. HPL-2008-20, Apr. 2008.

[14] J. Barth et al., “A 500 mhz random cycle, 1.5 ns latency, soi embedded dram macro featuring a three-transistor micro sense amplifier,” *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 86 –95, Jan. 2008.

[15] K. C. Chun, W. Zhang, P. Jain, and C. Kim, “A 700mhz 2t1c embedded dram macro in a generic logic process with no boosted supplies,” in *ISSCC*, Feb. 2011, pp. 506 –507.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” in *ISCA*, 1995, pp. 24–36.

[17] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. Vijaykumar, “Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories,” in *Low Power Electronics and Design, 2000. ISLPED*, 2000, pp. 90 – 95.

[18] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, “Adaptive mode control: A static-power-efficient cache design,” in *IEEE PACT*, 2001, pp. 61–70.

[19] S. Petit, J. Sahuquillo, J. M. Such, and D. Kaeli, “Exploiting temporal locality in drowsy cache policies,” in *Computing frontiers*, 2005, pp. 371–377.

[20] R. E. Matick and S. Schuster, “Logic-based edram: Origins and rationale for use,” *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 145–166, 2005.

[21] X. Liang, R. Canal, G.-Y. Wei, and D. Brooks, “Process variation tolerant 3t1d-based cache architectures,” in *MICRO*, 2007, pp. 15–26.

[22] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. W. Clark, “Managing leakage for transient data: decay and quasi-static 4t memory cells,” in *ISLPED*. ACM, 2002, pp. 52–55.

[23] C. Isen and L. John, “Eskimo: Energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem,” in *MICRO*, 2009, pp. 337–346.

[24] M. Ghosh and H.-H. Lee, “Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams,” in *MICRO*, Dec. 2007, pp. 134 –145.

[25] P. Emma, W. Reohr, and M. Meterelliyoz, “Rethinking refresh: Increasing availability and reducing power in dram for cache applications,” *Micro, IEEE*, vol. 28, no. 6, pp. 47 –56, nov.-dec. 2008.