Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies*

Aditya Agrawal, Prabhat Jain, Amin Ansari and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

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, eDRAM requires refresh. In this paper, we examine the opportunity of minimizing on-chip memory power further by intelligently refreshing on-chip eDRAM. We present Refrint*, a simple approach to perform fine-grained, intelligent refresh of on-chip eDRAM multiprocessor cache hierarchies. We introduce the Refrint algorithms and microarchitecture. We evaluate Refrint in a simulated manycore running 16-threaded parallel applications. We show that an eDRAM-based memory hierarchy with Refrint consumes only 30% of the energy of a conventional SRAM-based memory hierarchy, and induces a slowdown of only 6%. In contrast, an eDRAM-based memory hierarchy without Refrint consumes 56% of the energy of the conventional memory hierarchy, inducing a slowdown of 25%.*

1. Introduction

While CMOS technology continues to enable a 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 low voltages [4].

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 on-chip multi-level cache hierarchy of a manycore is 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 on-chip memory. They include various forms of dynamically-resizable caches with turned-off ways or sets (e.g., [23, 24]), cache decay [10], and Drowsy caches [6], among others. While these techniques can be effective, they need to be applied in a fine-grained manner to reduce the 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 an eighth or less [9]. It has the shortcoming that it needs to be refreshed, and refresh power is significant [22]. However, this fact in turn offers a new 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 for non-L1 caches.

In this paper, we examine the opportunity of power savings by intelligently refreshing an on-chip eDRAM cache hierarchy. Our goal is to refresh only the data that will be used in the near future, and only refresh it if it is really needed. The other data is invalidated and/or written back to main memory. 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 manycore with a three-level cache hierarchy, where L2 and L3 can be SRAM or eDRAM. The eDRAM-based memory hierarchy with Refrint consumes only 30% of the energy of a conventional SRAM-based memory hierarchy. In addition, Refrint's early invalidation and writeback of lines only increases the execution time of the applications by 6%. In contrast, an eDRAM-based memory hierarchy without Refrint consumes 56% of the energy of the conventional memory hierarchy, and induces an application execution slowdown of 25%.

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

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

*This work was supported in part by NSF under grants CPA 0702501 and CCF 1012759; DOE ASCR under Award Number DE-FC02-10ER2599; and DARPA under UHPC Contract Number HR0011-10-3-0007. Amin Ansari is supported by an NSF Computing Innovation Fellowship. Prabhat Jain is now with Intel. His email is prabhat.jain@intel.com.

requirements of chips. Moreover, a large fraction of the area is regularly devoted to caches — in fact, more than 70% in Niagara [11]. In addition, these chips are including extensive clock gating and ever more sophisticated management techniques for dynamic power. There is even significant interest in reducing the supply voltage of the chip and clocking it at moderate frequencies, to operate in a much more energy-efficient environment [4], with lower dynamic power.

The combination of progressively lower dynamic power and large on-chip caches points to on-chip cache leakage as one of the major contributors to present and, especially, future chip power consumption [6]. As a result, there have been proposals for new approaches and technologies to deal with on-chip SRAM leakage. These proposals include power gating (e.g., [10, 16, 23, 24, 25]), new SRAM organizations [6, 15], embedded DRAM (eDRAM), on-chip flash, and non-volatile memory technologies. Section 7 discusses this work.

One of the most interesting proposals is eDRAM, which has been used by IBM in the 32MB last level cache (LLC) of the POWER-7 processor [21]. 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 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 for non-L1 caches.

A major challenge in using eDRAM as the building cell of on-chip caches is its refresh needs. Since eDRAM is a dynamic cell, it needs to be refreshed at periodic intervals called *retention periods*, to preserve its value and prevent decay. On an access, the cell automatically gets refreshed, and stays valid for another retention period. Overall, refreshing the cells imposes a significant energy 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 energy by not refreshing the data that is not useful for the program execution anymore. The challenge is to identify such data inexpensively.

3. Refrint Architecture

3.1. Main Idea

We consider an on-chip multiprocessor cache hierarchy where L2 and L3 use eDRAM. We identify two sources of unnecessary refreshes, as summarized in Fig. 1. The first are *Cold* lines, namely those that are not being used or are being used far apart in time, but are still getting refreshed (Fig. 2). The second are *Hot* lines, which are those that are actively being used but are still getting refreshed because of the naive periodic refresh policy in eDRAMs (Fig. 3). Recall that, on a read or a write, a line is automatically refreshed, and hence there is no need to refresh it for another retention period. Cold lines are typically found in lower-level caches such as L3, while hot lines may be found in upper-level caches closest to the processor, such as L2.

$$\begin{matrix}\mathit{Unnecessary}\\ \mathit{Refreshes}\end{matrix} = \begin{matrix}\mathit{Cold\ lines}\\ (\mathit{lower\ level})\end{matrix} + \begin{matrix}\mathit{Hot\ lines}\\ (\mathit{upper\ level})\end{matrix}$$

Figure 1: Sources of unnecessary refreshes.

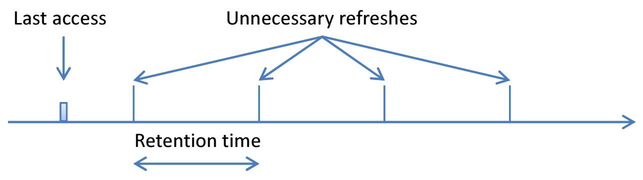

Figure 2: Access pattern of a cold line.

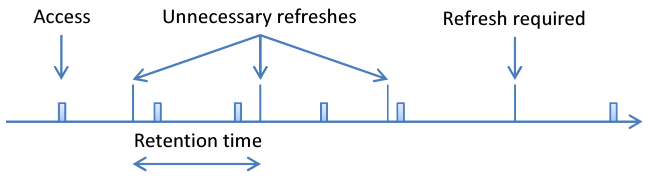

Figure 3: Access pattern of a hot line.

For the cold lines, we propose "data-based" policies, which identify and refresh only those cache lines which are expected to be used again in the near future. The rest of the lines are invalidated from the cache and not refreshed.

If the policies are too aggressive, we may end up invalidating a lot of useful lines from the caches, thereby having to access the lower level memory hierarchy to refetch lines a far higher number of times than in a conventional cache hierarchy. Also, writing back and invalidating a soon-to-be-accessed dirty line has double the penalty of invalidating a soon-to-be-accessed clean line, as it involves writing back the dirty line. Therefore, our policies need to be more conservative at handling dirty lines.

For the Hot lines, we propose "time-based" policies, which try to avoid refreshing lines after they have been accessed (and automatically refreshed). They improve over a naive periodic scheme that eagerly refreshes a line at regular periods, oblivious of when the line was last accessed.

In this paper, we focus on simple refresh policies. We do not consider line reuse predictors or similarly elaborate hardware structures. Also, we do not assume that we have information provided by the programmer or software system.

3.2. Refresh Policies Proposed

A refresh policy has a time- and a data-based component (Table 1). The time-based component decides when to refresh, while the data-based one decides what to refresh.

3.2.1. Time-Based Policy. We propose a time-based policy called *Polyphase*. Polyphase divides the retention period into a fixed number of equal intervals called *Phases*. Each cache line maintains information on which phase the line was last

Time-based policies: *When ?*	
Periodic	Refresh periodically
Polyphase	Refresh in the same phase
Data-based policies: *What ?*	
All	Refresh all lines
Valid	Refresh only Valid lines
Dirty	Refresh only Dirty lines
WB(n,m)	Refresh idle Dirty lines *n* times before writeback and refresh idle Valid Clean lines *m* times before invalidation

Table 1: Refresh policies proposed.

accessed. The phase information is updated on every read or write to the line. With Polyphase, a line is refreshed only when the same phase arrives in the next retention period. This approach reduces the number of refreshes of hot cache lines.

Fig. 4 shows an example of Polyphase, where a retention time is divided into four phases. A line is accessed in Phase 2 of the first retention time. Hence, rather than refreshing it again when the current retention period experies, it is refreshed only at the beginning of Phase 2 of the next retention period.

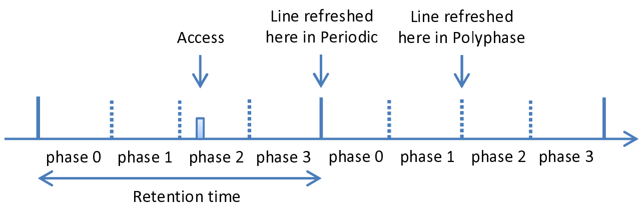

Figure 4: Polyphase with 4 phases.

In Polyphase, at the beginning of each phase, the cache controller quickly checks for lines whose phase information matches that of the controller. The phase information of each cache line is maintained in the cache controller and not in the data array. Hence, the check does not block the data array. The lines with the current phase are either refreshed or invalidated, depending on the data-policy employed. Every access to a cache line refreshes the cache line and updates its phase information. This approach performs the minimum number of refreshes to keep a particular line alive.

We also consider the trivial *Periodic* 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 phase bits; it only needs a global counter for the whole cache. However, it results in more refreshes than necessary, since it may eagerly refresh a line much before it is about to decay. Also, it may render the cache unavailable for a continuous period of time, when the lines are being refreshed.

3.2.2. Data-Based Policy. We propose the simple data-based policies shown in Table 1. They consider the state of the cache line in a multiprocessor hierarchy (valid, dirty, etc.), to decide what to refresh. Specifically, our policies are *All*, *Valid*, *Dirty*, and *WB(n,m)* (for write back). *All* refreshes every cache line, irrespective of whether it is valid or not. We evaluate this policy only for reference purposes. *Valid* and *Dirty* refresh Valid and Dirty cache lines, respectively, and invalidate the line otherwise.

The *WB* policy is associated with a tuple *(n,m)*. WB refreshes a Dirty line that is not being accessed for *n* times before writing it back and changing its state to Valid Clean; moreover, it refreshes a Valid Clean line that is not being accessed for *m* times before invalidating it. WB retains a Dirty line in the cache longer because evicting it has the additional cost of writing the line back to lower-level memory. To implement WB, we maintain a per-line *Count*. When the line is read or written, Count is set to *n* (if Dirty) or *m* (if Valid Clean). In addition, when the line is refreshed, Count is decremented. When Count reaches zero, the line is either written back or invalidated. Note that the *Dirty* policy is equivalent to *WB(∞,0)*, while *Valid* is equivalent to *WB(∞,∞)*. Finally, every policy refreshes cache lines in transient states as well.

Using cache line states for the refresh policy has the advantage that the hardware needed is simple. A disadvantage is that the policy is unable to disambiguate same-state lines that behave differently. In addition, the policy interacts with the cache coherence protocol and the inclusivity properties of multilevel caches. For example, if a policy decides to invalidate a line from L3, due to cache inclusivity, it also has to invalidate the line from L2 and L1. This results in extra network messages.

In our analysis and evaluation, we use the Periodic time-based policy and the All data-based policy as the baseline policy. A slightly smarter and natural extension is the Periodic Valid policy.

3.3. Application Categorization

We now categorize applications based on how they are affected by our time and data-based policies. We assume a cache-coherent multiprocessor with a multi-level, inclusive on-chip cache hierarchy, where the last level is shared by all the cores.

3.3.1. Time-Based Policy. Polyphase is effective in reducing the number of refreshes to a line when the interval between accesses to the line is shorter than its retention time — i.e., the frequency of accesses is higher than the refresh rate (1/Retention Time). This is shown in Fig. 5.

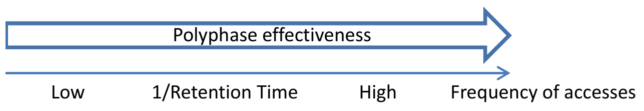

Figure 5: Application categorization according to the time-based policy.

For cache levels close to the processor, the frequency of cache accesses is high, but the fraction of energy consumed by refreshing is small. Hence, the overall impact of the time-based policy tends to be small. For cache levels far from the processor such as L3, the fraction of energy consumed by refreshing is high. However, we do not typically see repeated accesses to the same line because most of the accesses are intercepted by upper-level caches.

There are a few cases when lines in lower-level caches can observe repeated accesses. One is in applications with fine-grained sharing, where repeated coherence activity induces frequent writebacks to and reads from the shared cache. A second one is in codes with significant conflicts in the upper-level caches. A third one is in codes with accesses that are required to bypass the upper-level caches. Overall, in the application set that we evaluate in this paper, we do not find codes where any of these behaviors is dominant.

3.3.2. Data-Based Policy. We are interested in observing the effects from the point of view of the last level cache, which is the one that matters the most in the total refresh energy consumed. Fig. 6 presents an application categorization based on two axes: application footprint and visibility.

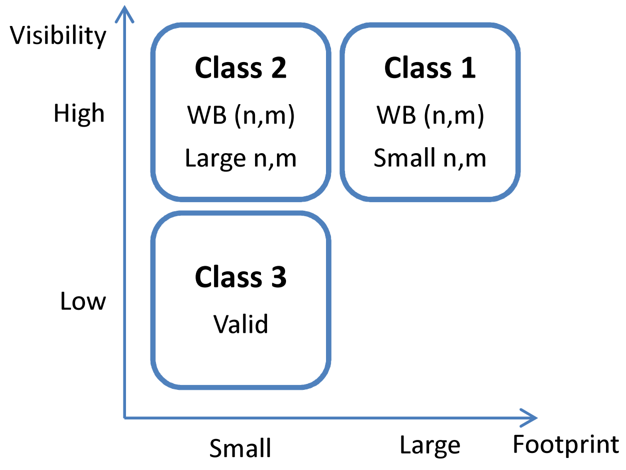

Figure 6: Application categorization according to the data-based policy.

The X axis shows 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, lines can be written back and/or invalidated; if they are reused, they can be brought back again. Therefore, the best policy for such applications should be a general one, namely WB(n,m), with a *small* (n,m). After an initial flurry of accesses, the data can soon be evicted from the cache. On the other hand, in 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.

The Y axis captures the last level cache's "visibility" on the activity of the lines in the upper levels of the cache hierarchy. For example, assume that the working set is such that: (i) it largely fits in the L1 and L2 caches (hence, there is no overflow) and (ii) there is little data sharing between processors (hence, the shared L3 cache does not see the data moving back and forth between caches and its associated state transitions between Dirty and Valid Clean). In this case, L3's 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 policy 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 between cores, such that data is frequently written by a processor and read by another, then visibility at L3 is high. There are frequent writebacks to and reads from L3. In such cases, a more specific policy such as 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 application set, we do not find any code of the type Large footprint and Low visibility; all the large-footprint applications also provide visibility to L3. This is either because they have substantial data sharing between cores, or because dirty data is often evicted from upper-level caches and written back to L3, therefore providing visibility.

4. Implementation Issues

In this section, we introduce the concepts of global and local phases, their implementation, and their use.

4.1. Phase Bits Design & Operation

We statically divide the retention time T into 2^N parts called *Global Phases*. For example, if $N = 2$, the retention time is divided into four global phases. The first quarter is the first phase, the second quarter is the second phase, and so on. We need N bits to encode 2^N global phases. In addition, each cache line is associated with N bits, called *Local Phase* bits. These bits record the global phase when the line was read or written. Hence, the number of global and local phase bits is the same. From our experiments, we found that values of 1 or 2 for N are good.

Assume that the system has an M bit counter ($M > N$) to support the retention time, such that 2^M clock ticks are equal to T. If we divide the retention time into 2^N phases, the N most significant bits of the counter will indicate the global phase. The counter is part of the baseline eDRAM implementation to keep track of retention time, and so there is no extra cost associated with keeping track of the global phase information.

Detecting a global phase transition is easy. Whenever the $M - N$ least significant bits of the counter become zero, we can infer a global phase transition.

For each cache line, we store the N local phase bits and a copy of the valid bit in the cache controller, in a structure called the *Phase Array*. If we assume a cache with a line size of 64 bytes and use N=2, the overhead in a line is 3 bits per 512 bits, which is less that 0.6%. If we assume a cache of size 1 MB, we have 16,384 lines. This requires a Phase Array of 16,384 X 3 bits = 49,152 bits = 6 KBytes. Such phase array can be organized as 96 rows of 64 Bytes each. Each row (line) then holds information for $\approx$171 lines of the cache.

On a normal read or write access (and hence automatic refresh) to a cache line, the global phase bits are copied to the local phase bits. This allows us to keep track of the global phase in which the line was accessed. If the same line is not accessed again, we will not refresh it for another retention period, i.e., until the same global phase in the next retention period. However, if the line is accessed before then, the local phase information is updated, and a refresh does not happen for another retention period beginning from that phase.

At the beginning of each global phase, all normal read and write requests are put on hold. The cache controller quickly scans the phase bits of the valid cache lines and, if their local phase bits match the global phase bits, schedules the line for refresh. Recall that the phase information and a copy of the valid bit is maintained inside the cache controller. From the example above, reading just one line of the array provides information for 171 lines. Thus, the cache controller can quickly find the lines that need to be refreshed, and issue back to back refresh requests. Finally, the controller releases the hold on normal accesses.

The phase array design and the associated logic is shown in the upper half of Fig. 7. The actions at the beginning of each global phase are summarized below.

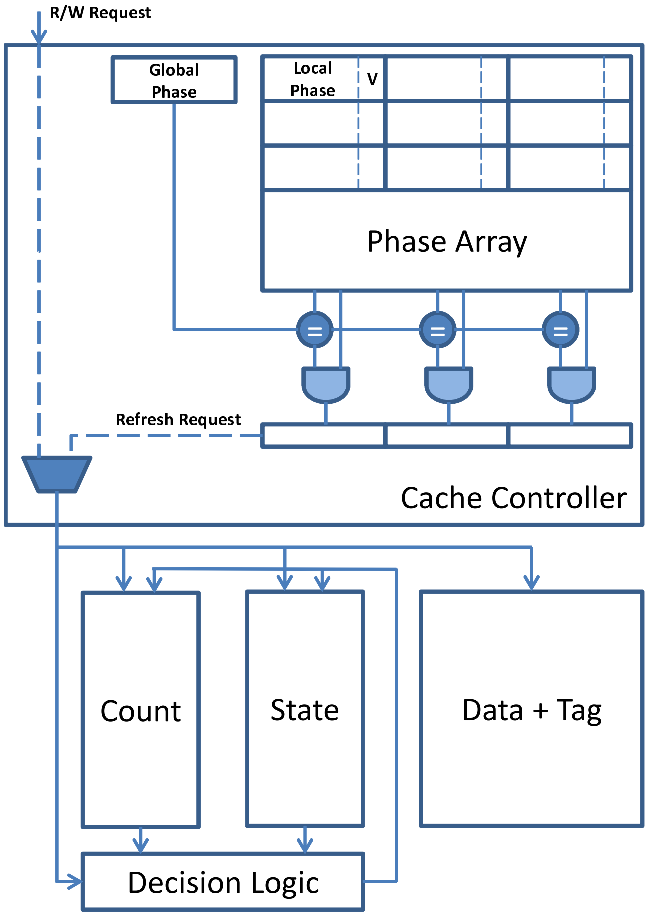

Figure 7: Phase array and logic to process refresh requests.

```
hold all read and write requests to the data array
for (all the lines of the cache) {
    if ((global phase == local phase) && (line == Valid))
        issue a refresh request for the line
}
release read and write requests
```

4.2. Processing Refresh Requests

The lower half of Fig. 7 shows the refresh processing logic that supports WB(n,m) and the other data-based policies that we consider. For each line, we have the line State bits and the Count bits. Note that the latter are not a counter but a set of bits. In our implementation, we use a 5-bit Count, which makes the Count and State overhead negligible.

The refresh request reaches the Decision Logic, which reads the State and Count bits. Depending on the data policy (All, Valid, Dirty or WB(n,m)) and the value of the State and Count bits, the logic may change the Count bits, refresh the line, and write back or invalidate the line. Invalidation may trigger the sending of invalidations to upper-level caches, and this is initiated by the cache controller. In case of a WB(n,m) policy, the steps are shown below.

```
read Count
if (Count >= 1)
    refresh line, decrement Count
else if (State == Dirty)
    write back and change State to Valid Clean
    /*the writeback automatically refreshes the line*/
    Count = m
else if (State == Valid Clean)
    invalidate
    /*may also invalidate upper-level caches*/
```

4.3. Array Lookup and Update Energy

At the beginning of each phase, the cache controller scans the phase array to find lines which require a refresh. We quantify the energy cost of such a lookup. Considering the cache of 1 MB from the example above, we have to read 96 lines of 64 Bytes and perform 16,384 3-bit comparisons. Therefore, the energy overhead is

$$E_{ovhd} = 96 \times E_{array\ line\ access} + 16384 \times E_{3-bit\ comparator}$$

From Synopsys synthesis tools, for a 32 nm process, we determine that $E_{array\ line\ access} = 20$ pJ and $E_{3-bit\ comparator} = 0.330$ fJ. Therefore, $E_{ovhd} = 1925.4$ pJ. From CACTI [18], the energy per access to a 1 MB cache for the same process is $\approx$100 pJ. Therefore, the energy overhead of the lookups is 19 cache accesses per phase. This is insignificant compared to the number of cache lines that are accessed in a phase.

The phase array is updated (read-modify-write) on every normal cache line read or write. As discussed above, the per access energy for a phase array line and for the cache are 20 pJ and 100 pJ, respectively. However, we will see in the evaluation that of all the energy consumed in the last level cache (which has 1 MB banks), dynamic energy accounts for very little. Hence, the overhead of updating the phase array is negligible as well. Overall, the small overhead of looking up and of updating the phase array pays off as substantial refresh energy savings.

5. Experimental Setup

We evaluate Refrint on a 16-core chip multiprocessor (CMP) system. Each core is a dual issue, out-of-order processor running the MIPS32 instruction set. Each core has a private instruction cache, a private data cache 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 L2 and each bank of L3 has dedicated logic to process refresh requests as shown in Fig. 7. We employ a MESI directory cache coherence protocol. The directory is maintained at L3. The architectural parameters are summarized in Table 2.

Architectural Parameters	
Chip	16 core CMP
Core	MIPS32, 2 issue out-of-order processor
Instruction L1	32 KB, 2 way. Access time (AT): 1 ns
Data L1	32 KB, 4 way, WT, private. AT: 1 ns
L2	256 KB, 8 way, WB, private. AT: 2 ns
L3	16 MB, 16 banks, WB, shared
L3 bank	1 MB, 8 way. AT: 4 ns
Line size	64 Bytes
DRAM	Round trip from controller: ≈80ns
Network	4 x 4 torus
Coherence	MESI directory protocol at L3
Technology Parameters	
Technology node	32 nm
Frequency	1000 MHz
Device type	LOP (Low operating power)
Temperature	330 Kelvin
Tools	
Architecture	SESC [17]
Timing & Power	McPAT [12] & CACTI [18]

Table 2: Architectural and technology parameters and tools.

We model our architecture (cores, memory subsystem, and network) with the SESC [17] cycle-level simulator. The dynamic energy and leakage power numbers for cores and network are obtained from McPAT [12], while those for memories are obtained from CACTI [18]. Even though McPAT uses CACTI internally, it does not allow for an eDRAM memory hierarchy. Hence, we have to use CACTI as a standalone tool. We experiment with 2 different values of retention times, namely 50µs and 100µs. Barth et al. [2] report a retention time of 40µs for eDRAM cells at 105°C. The retention time has an exponential dependence on temperature [3]. In this paper, we target a low-voltage, low-frequency simple core and an energy-efficient architecture for which the temperatures are significantly lower than 105°C. Hence, we conduct experiments at the above mentioned retention times. Other experimental parameters like temperature and frequency are also summarized in Table 2.

We compare a full-SRAM cache hierarchy (baseline) to one where L2 and L3 are eDRAM. To do a fair and simple comparison between the two, we have made a few simplifying decisions, which are listed in Table 3. Specifically, we assume that the L2 and L3 access times are the same in both hierarchies. We also assume that the access energies are the same. The leakage power of an eDRAM cell is 1/8th of that of an SRAM cell. In addition, for eDRAM, the time and energy to refresh a line is equal to the time and energy to access the line. Finally, a line can be refreshed in a cycle, when done in a pipelined fashion.

Parameter
eDRAM access time = SRAM access time
eDRAM access energy = SRAM access energy
eDRAM leakage power = 1/8 x SRAM leakage power
eDRAM line refresh time = eDRAM line access time
eDRAM line refresh energy = eDRAM line access energy

Table 3: Assumptions made for the memory cells in L2 and L3.

We evaluate Refrint by running 16-threaded parallel applications from the SPLASH-2 and PARSEC benchmark suites. The set of applications and the problem sizes are summarized in Table 4. In addition, we ran a synthetic application, called *fine share*, with fine-grained producer-consumer behavior. In this application, each thread independently reads and modifies a block of data and reaches a barrier. At the barrier, the block of data is passed to the adjacent thread. This is done in a loop. The time between consecutive barriers is less than the retention time, so that the frequency of accesses to the lines in the data block in the shared cache (L3) is higher than the refresh rate.

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

Table 4: Applications and input sizes run.

Each application is run at 2 retention times, 4 timing policies, 7 data policies and the baseline case, amounting to a total of 57 combinations. The parameter sweep is summarized in Table 5.

Retention time	50 μs, 100 μs	2
Timing policy	Periodic, Polyphase (num. phases = 1, 2, 4)	4
Data policy	All, Valid, Dirty, WB(4,4), WB(8,8), WB(16,16), WB(32,32)	7
Total combinations		57

Table 5: Parameter sweep.

6. Evaluation

In this section, we present our evaluation of Refrint. We present the effect of our policies on memory hierarchy energy (Section 6.1), total energy (Section 6.2) and execution time (Section 6.3).

Policies: We present results for *Periodic* and *Polyphase*. The *All* and *Valid* data-based policies do not create extra DRAM accesses. However, the *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 [1] into account. In addition, we assume that at the end of the simulation all dirty data will be written back to main memory.

Applications: In Section 3.3 and in Fig. 6 we categorized applications into three classes based on data policies. In the course of our evaluation, we found that applications within a class responded similarly to our data policies. Table 6 shows the binning for our set of applications. In the following sections, rather than picking one representative application from each class, we will present average numbers for the entire class. To show the effectiveness of time policies we also present data from one synthetic application.

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

Table 6: Application binning.

6.1. Memory Hierarchy Energy

In Fig. 8, we have three bars. The first one shows the memory energy as the sum of L1, L2, L3 and DRAM energies, from bottom to top, normalized to the baseline memory hierarchy. The second and third bars show the L2 and L3 energies as the sum of their dynamic, leakage and refresh components, from bottom to top, normalized to the SRAM L2 and SRAM L3 energies, respectively. The data is for a naive eDRAM policy of refreshing all lines periodically at 50 μs, and have been averaged over all applications.

We observe that L3 consumes the majority ($\sim$ 60%) of the total memory energy, of which about 70% is refresh energy. L2 consumes about 10% of the total memory energy, of which about 50% is refresh energy. The aim of our policies is to shave off as much refresh energy as possible. Next, we show

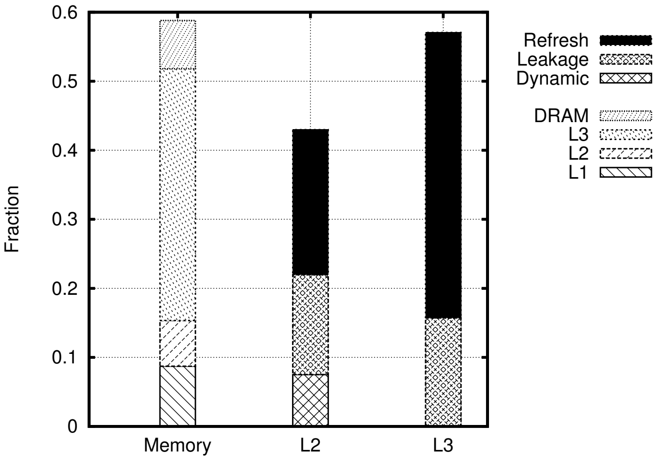

Figure 8: Memory system, L2 & L3 energy (normalized to the corresponding baseline energy).

the effectiveness of our polices in saving refresh energy in L3 and L2.

6.1.1. L3 Refresh Energy. To isolate the effectiveness of our policies in saving L3 refresh energy, we employ a Periodic timing policy and Valid data policy in L2. We do a parameter sweep on L3 refresh policies as summarized in Table 5.

In Fig. 9, we show the memory energy (averaged) for Class 1, Class 2 and Class 3 applications as the sum of on-chip dynamic, leakage, and refresh energies and DRAM energy. The fourth plot (labeled 'all') shows the average over all applications. On the X-axis, we have 2 sets of bars at retention times of 50 μs and 100 μs. Within each retention time, we have 4 time-based policies, namely Periodic (P) and Polyphase with 1, 2 and 4 phases (PP1, PP2, and PP4). For each time-based policy we have 7 data-based policies, namely *All, Valid, Dirty, WB(4,4), WB(8,8), WB(16,16)* and *WB(32,32)*. The bars are labeled 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 the sum of on-chip dynamic, leakage, and refresh energies and DRAM energy, normalized to the baseline memory hierarchy energy. The policy P.all represents the naive eDRAM policy of refreshing all lines periodically.

In all classes of applications, the amount of dynamic energy remains almost the same across retention times and across policies because the amount of work done is the same. The amount of leakage energy varies because of the effect of the policies on execution time (Section 6.3). The main variation is in the amount 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, whose energy consumption has been taken into account.

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

Timing Policies: Polyphase policies (PP1, PP2, and PP4) always do better than Periodic because of two reasons. First, in

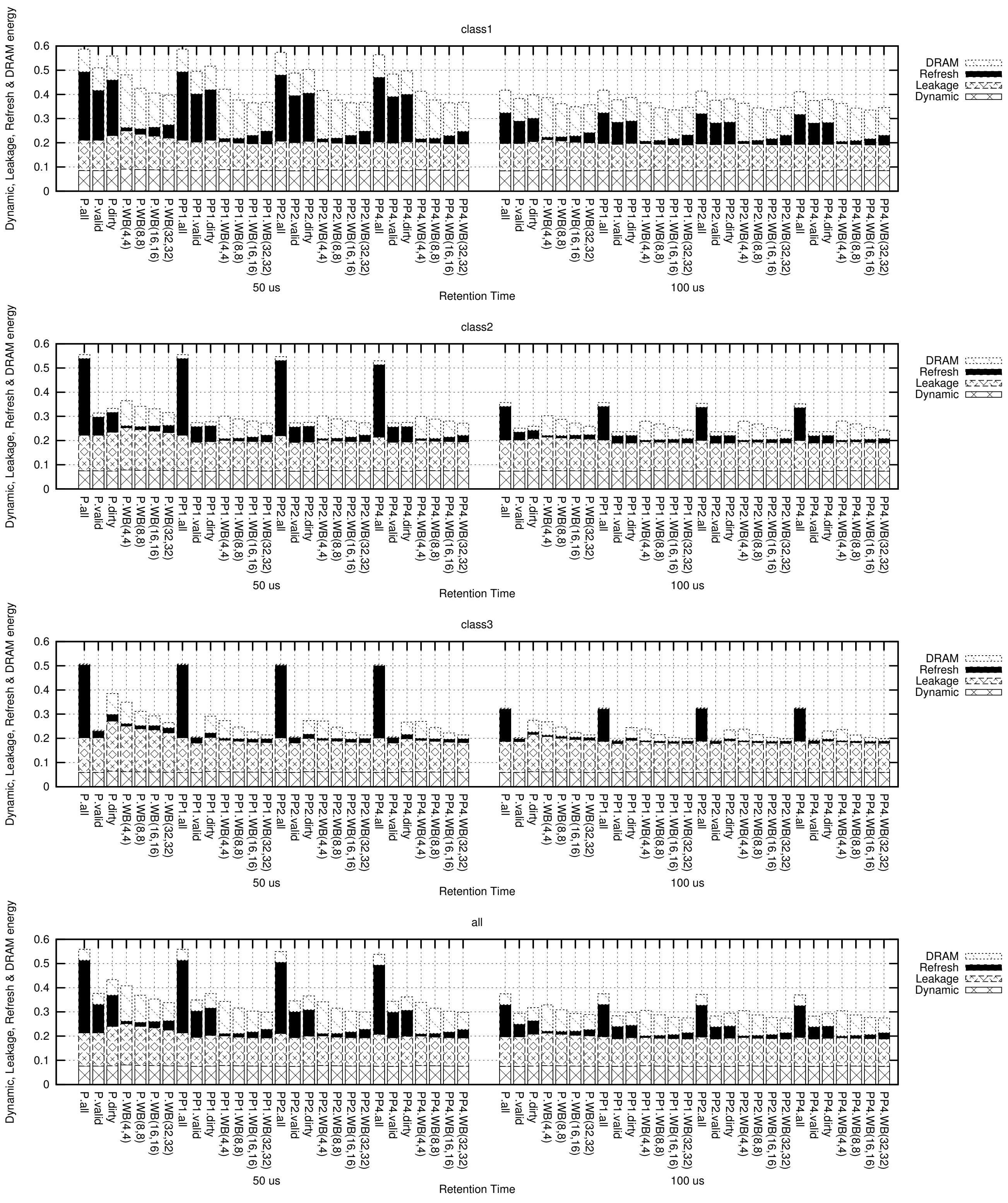

Figure 9: On-chip dynamic, leakage, refresh & DRAM energy (normalized to the baseline memory hierarchy energy).

Periodic, a cache cycle is spent for every line (valid or invalid) to determine if a refresh needs to be scheduled. In Polyphase, thanks to the phase array, this check is performed for multiple lines at once, and cache cycles are consumed only for issued refresh requests. Second, PP2 and PP4 can save refreshes compared to Periodic and PP1 by exploiting recently-accessed lines. Unfortunately, for the applications considered, we do not see much benefit in increasing the number of phases in a retention period to PP2 or PP4. This is because the number of read/write accesses to L3 is very small in comparison to

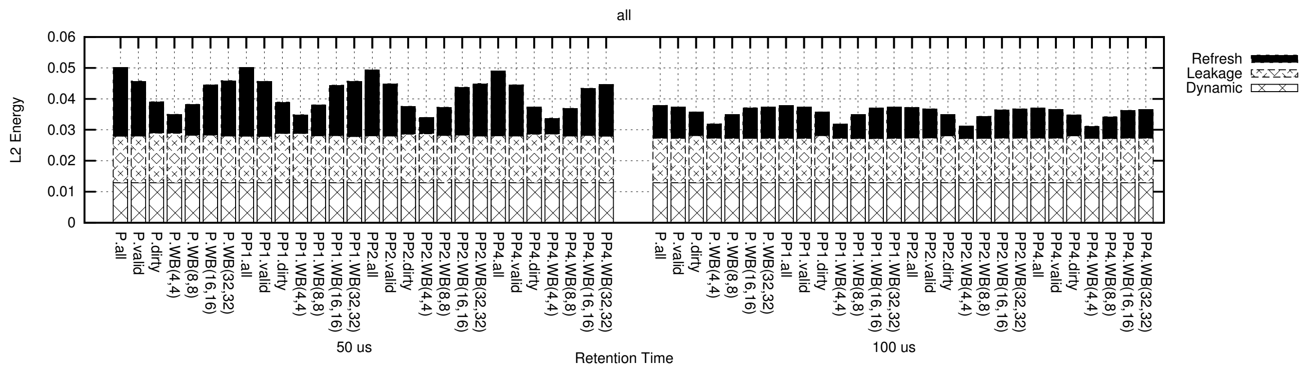

Figure 10: L2 dynamic, leakage and refresh energy (normalized to the baseline memory hierarchy energy).

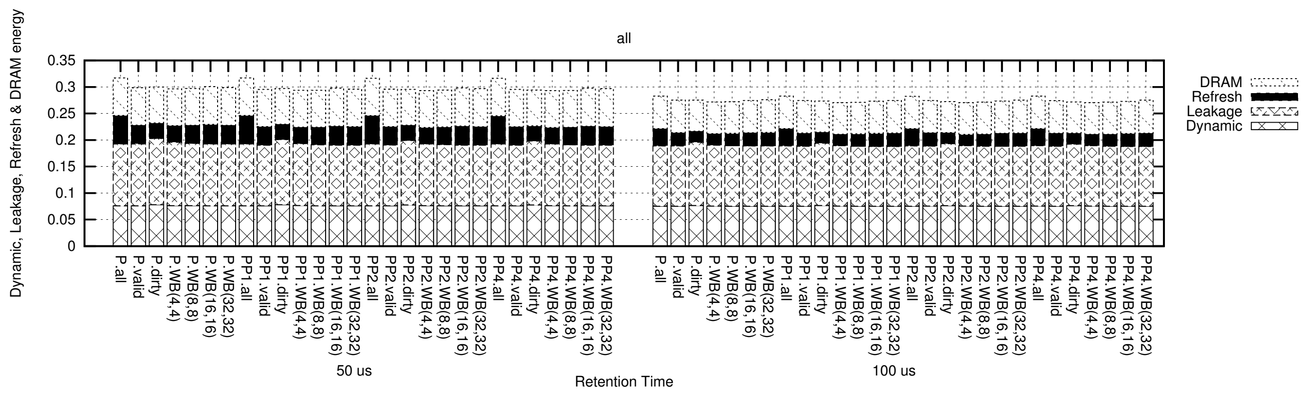

Figure 11: On chip dynamic, leakage, refresh & DRAM energy (normalized to the baseline memory hierarchy energy).

the number of refreshes. Hence, adding more phases is not noticeably effective at eliminating refreshes.

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

On average across all applications, Polyphase with 1 phase (PP1) does better than Periodic schemes. Also, the *WB(32,32)* policy does better than all other policies. At 50 μs, on average, the base refresh policy for eDRAM (Periodic All) consumes 56% of the energy of the conventional SRAM-based memory hierarchy. Our Polyphase WB(32,32) policy consumes only 30% of the energy of the conventional SRAM-based memory hierarchy.

6.1.2. L2 Refresh Energy. In the section above, we found that on average the Polyphase timing policy with 1 phase and the WB(32,32) data policy perform well for L3. Now, to isolate the effectiveness of our policies in saving L2 refresh energy, we freeze the L3 refresh policy to Polyphase with 1 phase and WB(32,32) and do a parameter sweep on L2 refresh policies. Since the L2 energy is a small fraction of the total memory energy, in Fig. 10 we show the impact of refresh policies only on L2 energy. Fig. 11 shows the impact of L2 policies on the total memory energy, averaged over all applications. In both plots, the X-axis is the same as in Section 6.1.1.

Retention Time: In both plots, as the retention time increases, the lines have to be refreshed less often and hence the amount of refresh energy reduces.

Timing Policies: Though not so significant, increasing the number of phases (keeping the data policy constant) shaves off about 5% refresh energy, as can be seen in Fig. 10.

Data Policies: In Fig. 10, we notice that the WB(4,4) policy is highly effective at saving L2 energy. At 50 μs, on average across all applications, WB(4,4) saves 30% of the L2 energy compared to P.all. However, in Fig. 11, which shows the total memory hierarchy energy, we observe that the energy benefits of policies on L2 have leveled out. This is because the L2 refresh energy represents a small fraction of the total memory energy.

Therefore, to reduce implementation complexity, a periodic-valid refresh policy (P.valid) is the best choice for L2.

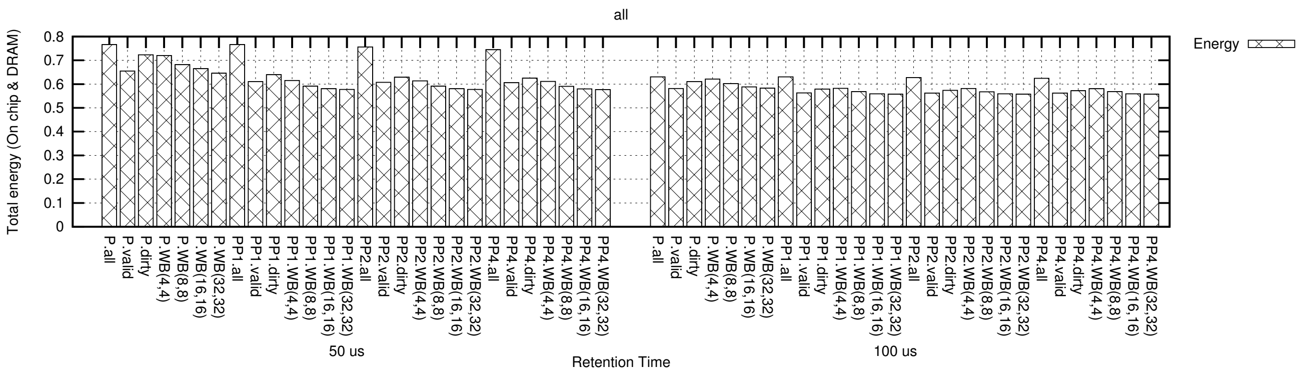
Figure 12: Total energy (normalized to the baseline system energy).

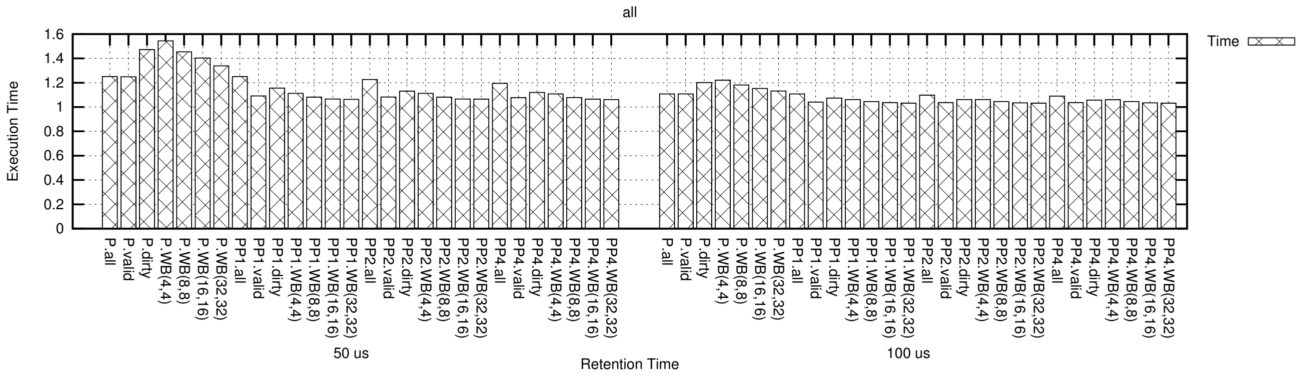
Figure 13: Execution time (normalized to the baseline execution time).

6.2. Total Energy

Using the result from Section 6.1.2 we freeze the L2 refresh policy to periodic-valid. In Fig. 12 we show the normalized total system energy (cores, caches, network and DRAM access energy) averaged over all applications, with a parameter sweep on L3 policies. The X-axis of the plots is the same as in Section 6.1.1. On average, across all applications, Polyphase with 1 phase and WB(32,32) still does the best. At 50 μs, on average, a system with the conventional refresh policy for eDRAM (Periodic All) consumes 77% of the energy of the baseline system. A system with our Polyphase WB(32,32) policy consumes 58% of the energy of the baseline system.

6.3. Execution Time

Using the same refresh policies and parameter sweep as in the above section, in Fig. 13 we show the normalized execution time averaged over all applications. The X-axis of the plot is the same as in Section 6.1.1. The Y-axis represents the total application execution time. All applications have the same trend across the retention times and policies. On average, with increasing retention times, the performance penalty reduces.

Timing Policies: Periodic schemes do worse in comparison to Polyphase schemes. This is because periodic schemes consume a cache cycle per line, while Polyphase schemes only consume it if they issue a refresh request. We also see that, while Polyphase schemes with more phases per retention period could potentially save refreshes over those with fewer phases, our applications do not noticeably improve with more phases per retention period.

Data Policies: The Valid and WB(32,32) data policies are the best with respect to execution time, as they keep all the lines or keep them for a long enough time. The Dirty and other 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 the WB(n,m) policy goes down as (n,m) grow. This is reasonable, as data is kept around for a longer time.

At 50 μs, on average, the execution time in a system with the base refresh policy for eDRAM (Periodic All) is about 25% longer than in the baseline system. The execution time in a system with our Polyphase WB(32,32) policy is only 6% longer than in the baseline system.

6.4. Effectiveness of Polyphase

As pointed out in Section 3.3.1 and observed above, applications from the SPLASH-2 and PARSEC suites do not benefit noticeably from time-based policies with phases. This is because such codes do not exhibit substantial amounts of fine-grained sharing, upper-level cache conflicts, or accesses to data uncacheable in upper-level caches. To show the effectiveness of Polyphase in one of these cases, we evaluate *fine share*, a microbenchmark with fine-grained sharing.

Fig. 14 shows the refresh energy in the shared L3 cache

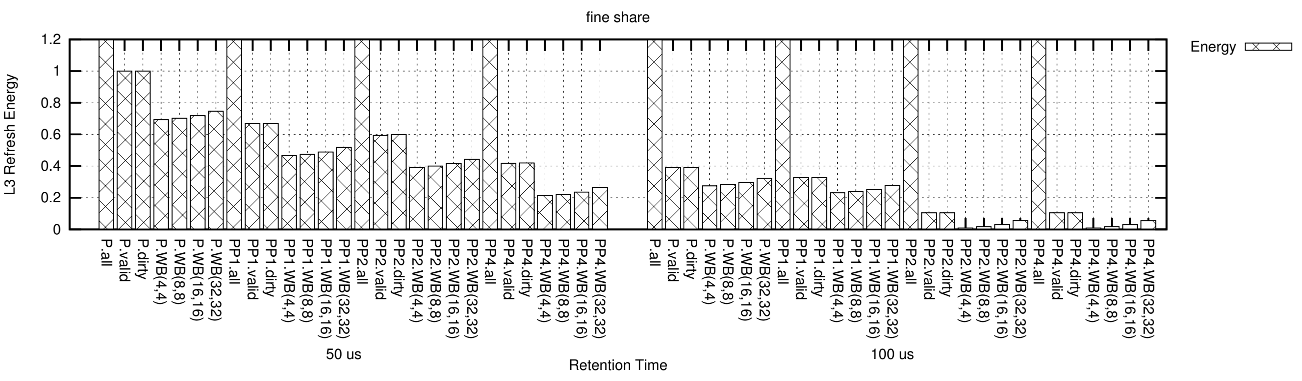

Figure 14: Effectiveness of Polyphase on a microbenchmark with fine-grained sharing.

while running *fine share*. The X-axis is the same as in Section 6.1.1. The Y-axis is normalized to the L3 refresh energy for the Periodic Valid policy at 50 μs. The bars for the All policies extend beyond the range shown and are cut. From the figure, we see that, across the two retention times and all the data policies, there is a significant decrease in L3 refresh energy with an increase in the number of phases. For example, at 50 μs for data policy WB(32,32), the reduction from PP1 to PP4 is 50%.

7. Related Work

To reduce leakage power in conventional SRAM caches, several approaches have been proposed. One of them is Gated-Vdd [16] and Cache Decay [10, 25], which turn off cache lines that are not likely to be accessed in the near future, thereby saving leakage power. Cache Decay uses counters, which may have a significant cost for large lower-level caches. With this appoach, the state of a cache line is lost when the line gets turned off.

A second approach is Drowsy Caches [6, 15]. Inactive lines are periodically moved to a low-leakage mode, where they cannot be read or written. This approach has the advantage that the data state is preserved, using two different supply voltage modes. This scheme will be less applicable in the future, where the difference between V_{dd} and V_{th} will be smaller.

Both of the previous approaches require design changes, power gating/voltage biasing circuitry, and have a non-negligible hardware overhead. On the other hand, eDRAM, the focus of our work, offers intrinsic leakage reduction and area benefits.

eDRAM cells can be logic-based or DRAM based. Logic-based eDRAM operates faster but 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 [14]. To make eDRAM delay characteristics closer to SRAM, and to simplify the process technology, Liang et al. [13] 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. Hu et al. [8] 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 the 6T SRAM cell, 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. [19] introduced a macro-cell that combines SRAM and eDRAM at cell level. They implement an 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, the power consumption and the performance overhead of refreshing the eDRAM cells become the main bottlenecks. An interesting approach is introduced by Venkatesan et al. [20]. It 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 of the cache used instead of the entire cache.

Ghosh et al. [7] proposed Smart Refresh, which is a technique to avoid unnecessary refreshes of lines in the main memory DRAM that have recently been read or written. To accomplish this, it uses timeout counters per line. Smart Refresh differs from Refrint in several ways. First, Refrint uses two approaches to eliminate refreshes: not refresh the lines that are not being used, and not refresh the lines that have recently been accessed. Smart Refresh is only relevant to the second approach. In addition, Smart Refresh needs to change DRAM arrays, which is harder to do, while Refrint, being on chip, can augment memory structures more easily. Finally, Smart Refresh has a counter per line, while Refrint just keeps count bits (no logic) per line. Of course, Smart Refresh lines are longer.

Using error-correction codes (ECC) is another technique to reduce the refresh power [5, 22]. ECC can tolerate some failures and hence, allows an increase in the refresh time. By employing a stronger ECC, we can increase the refresh period and reduce the refresh power. Nonetheless, strong codes come with overheads in terms of storage, encoding/decoding power, area, and complexity.

8. Conclusions

To reduce the power consumed in the on-chip memory hierarchy of large manycores, this paper considered a low-leakage technology (eDRAM) and proposed to refresh it intelligently for power savings. Our goal is to refresh only the data that will be used in the near future, and only if the data has not been recently accessed (and automatically refreshed). Our technique is called *Refrint*. We introduced the Refrint algorithms and microarchitecture.

We evaluated 16-threaded parallel applications running on a chip multiprocessor with a three-level on-chip cache hierarchy. Our results showed that Refrint is very effective. On average, an eDRAM-based memory hierarchy without Refrint consumed 56% of the energy of a conventional SRAM-based memory hierarchy. However, it increased the execution time of the applications by 25%. On the other hand, an eDRAM-based memory hierarchy with Refrint only consumed 30% of the energy of the conventional SRAM-based memory hierarchy. In addition, it only increased the execution time of the applications by 6%. In this environment, the contribution of refreshes in the energy remaining was negligible.

References

[1] J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies," in *International Symposium on Computer Architecture*, Jun. 2008.

[2] J. Barth, W. Reohr, P. Parries, G. Fredeman, J. Golz, S. Schuster, R. Matick, H. Hunter, C. Tanner, J. Harig, K. Hoki, B. Khan, J. Griesemer, R. Havreluk, K. Yanagisawa, T. Kirihata, and S. Iyer, "A 500 MHz Random Cycle, 1.5 ns Latency, SOI Embedded DRAM Macro Featuring a Three-Transistor Micro Sense Amplifier," *IEEE Journal of Solid-State Circuits*, Jan. 2008.

[3] 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 *IEEE International Solid-State Circuits Conference*, Feb. 2011.

[4] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proc. of the IEEE*, Feb. 2010.

[5] P. Emma, W. Reohr, and M. Meterelliyoz, "Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications," *IEEE Micro*, Nov.-Dec. 2008.

[6] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," in *International Symposium on Computer Architecture*, 2002.

[7] M. Ghosh and H.-H. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in *International Symposium on Microarchitecture*, Dec. 2007.

[8] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. Clark, "Managing Leakage for Transient Data: Decay and Quasi-Static 4T Memory Cells," in *International Symposium on Low Power Electronics and Design*, Aug. 2002.

[9] S. S. Iyer, J. E. B. Jr., P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak, "Embedded DRAM: Technology Platform for the Blue Gene/L chip," *IBM Journal of Research and Development*, Mar. 2005.

[10] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," in *International Symposium on Computer Architecture*, Jun. 2001.

[11] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor," *IEEE Micro*, 2005.

[12] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *International Symposium on Microarchitecture*, Dec. 2009.

[13] X. Liang, R. Canal, G.-Y. Wei, and D. Brooks, "Process Variation Tolerant 3T1D-Based Cache Architectures," in *International Symposium on Microarchitecture*, Dec. 2007.

[14] R. E. Matick and S. Schuster, "Logic-based eDRAM: Origins and Rationale for Use," *IBM Journal of Research and Development*, 2005.

[15] S. Petit, J. Sahuquillo, J. M. Such, and D. Kaeli, "Exploiting Temporal Locality in Drowsy Cache Policies," in *Computing Frontiers*, May 2005.

[16] 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 *International Symposium on Low Power Electronics and Design*, Jul. 2000.

[17] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," Jan. 2005, http://sesc.sourceforge.net.

[18] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi, "CACTI 5.1. Technical Report," Hewlett Packard Labs, Tech. Rep., Apr. 2008.

[19] 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 *International Symposium on Microarchitecture*, Dec. 2009.

[20] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *International Symposium on High Performance Computer Architecture*, Feb. 2006.

[21] D. F. Wendel, R. N. Kalla, J. D. Warnock, R. Cargnoni, S. G. Chu, J. G. Clabes, D. Dreps, D. Hrusecky, J. Friedrich, M. S. Islam, J. A. Kahle, J. Leenstra, G. Mittal, J. Paredes, J. Pille, P. J. Restle, B. Sinharoy, G. Smith, W. J. Starke, S. Taylor, J. V. Norstrand, S. Weitzel, P. G. Williams, and V. V. Zyuban, "POWER7: A Highly Parallel, Scalable Multi-Core High End Server Processor," *IEEE Journal of Solid-State Circuits*, Jan. 2011.

[22] 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 *International Symposium on Computer Architecture*, Jun. 2010.

[23] 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 *International Symposium on High Performance Computer Architecture*, Jan. 2001.

[24] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar, "Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay," in *International Symposium on High-Performance Computer Architecture*, Feb. 2002.

[25] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, "Adaptive Mode Control: A Static-Power-Efficient Cache Design," in *International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2001.