Energy Efficient Caching for Phase-Change Memory

Neal Barcelo*, Miao Zhou*, Daniel Cole *, Michael Nugent*, and Kirk Pruhs*
*Department of Computer Science
University of Pittsburgh, Pittsburgh, Pennsylvania 15260
Email: ncb30, miaozhou, dcc20, mnugent, kirk@cs.pitt.edu

Abstract

Phase-Change Memory (PCM) has the potential to replace DRAM as the primary memory technology due to its non-volatility, scalability, and high energy efficiency. However, the adoption of PCM will require technological solutions to surmount some deficiencies of PCM, such as writes requiring significantly more energy and time than reads. One way to limit the number of writes is by adopting a last-level cache replacement policy that is aware of the asymmetric nature of PCM read/write costs. We first develop a cache replacement algorithm, Asymmetric Landlord (AL), and show that it is theoretically optimal in that it gives the best possible guarantee on relative error. We also propose an algorithm Variable Aging (VA), which is a variation of AL. We have carried out a simulation analysis comparing the algorithms LRU, N-Chance, AL, and VA. For benchmarks that are a mixture of reads and writes, VA is comparable or better than N-Chance, even for the best choice of N, and uses at least 11% less energy than LRU. For read dominated benchmarks, we find that AL and VA are comparable to LRU, while N-Chance (using the N that was best for benchmarks that were a mixture of reads and writes) uses at least 20% more energy.

Keywords

cache replacement; algorithms; phase change memory; energy; power;

I. Introduction

Dynamic Random Access Memory (DRAM) has been the memory of choice for most computer systems for decades, but it now faces two critical problems. First, DRAM is projected to encounter severe scalability problems in coming years because DRAM relies on charge placement and control logic, which are inherently unscalable [1]. Second, increasingly large DRAM components have become a major consumer of energy in computer systems. Currently main memory consumes up to 40% of the energy in computer systems, comparable to, or slightly higher than, the energy consumption of processors [2].

In reaction to these problems, *Phase-Change Memory* (PCM) has been proposed as a replacement technology for DRAM (or maybe an add-on technology to DRAM). Desirable properties of PCM include: non-volatility, good scalability, and high energy efficiency (due to its low read power and very low idle power). Undesirable properties of PCM include: non-durability to writes (PCM memory can fail after 10^7 or 10^8 writes, which can happen in days or even hours without wear-leveling techniques [3–6]), PCM is slower than DRAM, and with current technology, PCM writes have up to 20 times higher latency, and 10 times higher energy consumption than PCM reads [1,7].

To mitigate these shortcomings, researchers have proposed to organize PCM main memory with a small cache [1,8,9]. The cache could be an on-chip cache or an off-chip DRAM cache. Logically, it is the *last-level cache* (LLC) of the PCM main memory. The LLC improves performance by caching highly accessed data, and further, extends PCM lifetime and reduces energy consumption by filtering a large portion of destructive and energy costly PCM writes.

The LLC replacement policy will be crucial for improving the energy efficiency of PCM as a main memory technology. An ideal replacement strategy must take into account the read/write asymmetry of PCM: PCM reads are relatively fast and energy efficient, while PCM writes are very slow and energy hungry. PCM bits are stored as the physical state of chalcogenide material, and writes are costly because changing this state is a process of

heating followed by controlled cooling. However, since PCM can be written to one bit at a time (unlike DRAM), how much time and energy is needed to write a page is directly related to how many bits of that page have changed. So intuitively a good replacement strategy should be more hesitant (than a replacement strategy for DRAM) to evict dirty cache pages since this involves writing to PCM, and just how hesitant may depend on the number of bits that have changed in that page. The most common DRAM replacement strategy, Least Recently Used (LRU), does not differentiate between evicting clean and dirty pages.

To the best of of our knowledge, there is only one paper in the literature specifically addressing the issue of developing a good cache replacement strategy for PCM. Ferreira et al. [6] proposed a policy N-Chance, where N is a tunable integer parameter, that is a variation of LRU. N-Chance evicts the least recently accessed clean page from cache, unless all of the N least recently accessed pages are dirty, in which case it evicts the least recently accessed page. Note that when $N = 1$, N-Chance is exactly LRU, so there is always a choice of the parameter N where N-Chance is at least as good as LRU. In a simulation study, [6] showed that N-Chance can be significantly better than LRU on common benchmarks, given an appropriate choice for the parameter N. However, [6] also showed that the best choice of the parameter N can vary significantly based on the application, and on the ratio of the write to read cost in PCM.

The goal of the research reported here is to theoretically and experimentally investigate other replacement strategies that account for the read/write cost asymmetry of PCM, and in particular, to develop algorithms that do not require an application specific tunable parameter and that take into account how dirty a page is. We primarily focus on the energy costs of accessing PCM. We adopt the most natural simple model, which assumes that a PCM read costs 1 unit of energy, and a PCM write costs some constant $c > 1$ units of energy. Most naturally c might be the cost to change half of the bits in the page, which would be approximately the number of bits that would have to be written if there was no correlation between a page's original value and the page's new value. Actually our theoretical results generalize to a setting where the write cost is concave in the number of times that page has been written to while being in cache. Discussion of why this might be a reasonable assumption for common applications can be found in [10]. For example, if a random fixed percentage of the bits are changed with each write then the cost to write back to PCM is concave in the number of writes.

One of our research goals was to design provably scalable algorithms. A scalable algorithm guarantees a bounded error relative to the minimum possible achievable cost with slightly less resources, which in our case means a slightly smaller cache. A scalable algorithm guarantees that for every access pattern, if the PCM energy costs were high for the scalable algorithm, then the energy costs would be high for every algorithm equipped with a slightly smaller cache. Thus intuitively scalability means that, for every access pattern, either the cache size was at a phase change point for the access pattern where the costs change dramatically for that cache size, or the high energy costs were unavoidable for that access pattern (and thus not the algorithm's fault).

It is not too difficult to see that even for very simple memory access patterns, N-Chance can have very high PCM energy costs, even if low energy costs are achievable with a smaller cache. Intuitively if N is too large, N-Chance can have an unnecessarily high cost for cycles of read requests to clean pages, because N-Chance can indefinitely freeze up to N dirty pages in cache, which effectively reduces the cache size available for the clean pages. And if N is too small, N-Chance can have unnecessarily high cost for a sequence that alternates between reads to one working set of pages, and writes to another working set of pages, because N-Chance won't keep enough of the dirty pages in cache.

In section III we develop and analyze a scalable cache replacement algorithm Asymmetric Landlord (AL) that guarantees a bounded error. We show that for all access patterns, the PCM energy cost for AL is at most $\frac{1}{\epsilon}$ times the minimum possible cost for that access pattern for a cache of size $(1 - \epsilon)$ times the current cache. Roughly speaking, AL associates a Time-To-Live (TTL) value with each page in cache, and always evicts the least recently used page among the pages with the lowest TTL value (and reduces the TTL of the other pages by the TTL of the evicted page). When clean pages are brought into cache, they are initially given a TTL of 1. When dirty pages are brought into cache they are given a TTL of $c_1 + 1$, where c_1 is the average energy cost to write back a page that has been written to once. When write costs are small, AL is intuitively similar to LRU

(but it is not exactly LRU). Further, AL does not have any application specific tunable parameters.

In an effort to more intuitively describe AL (there are a few more cases than described above), we discovered another intuitive algorithm, that we call Variable Aging (VA). VA is another generalization of LRU. In LRU, each page in cache can be viewed as having an age equal to the number of accesses since it was last accessed (so pages are reborn when they are accessed), where upon eviction, LRU evicts the oldest page in cache. In VA, the rate at which pages age is inversely proportional to the energy cost that the page will incur when evicted. So clean pages age at a rate of 1, and dirty pages age at a rate of $1/c$, where c is the average cost of writing a page to memory. Again VA always evicts the oldest page. So both AL and VA have ages, but in AL the age is a time until death, and in VA the age is a time since rebirth. It is not difficult to see that there are access patterns where VA can have high energy costs, even if low energy costs were achievable with a small cache.

LRU, N-Chance, AL, and VA, all assume that pages must be brought into cache to be accessed. In principle, one might also read-through or write-through cache by accessing the page directly from PCM, avoiding having to move the page into cache. This would be the right choice for example if the cache was all dirty, and there was a single read to a page that was not in cache; a read-through would cost a PCM read, while an eviction would cost a PCM read and a PCM write. In section III we develop a replacement algorithm that incorporates read-throughs and write-throughs (and also the possibility of variable sized pages). We show that for all access patterns, the PCM energy cost for this algorithm is at most $\frac{2}{\epsilon}$ times the minimum possible cost (again assuming the possibilities of read-throughs and write-throughs) for that access pattern for a cache of size $(1-\epsilon)$ times the current cache.

We have carried out a simulation analysis comparing the algorithms that do not use read-through and write-through, namely LRU, N-Chance, AL, and VA. In section IV, we give the experimental set-up. We evaluate the energy efficiency and performance of the proposed policies with SPEC CPU 2006 benchmarks using a trace-driven cycle-accurate simulator. In section V, we report on our experimental results. For benchmarks that are a mixture of reads and writes, VA is comparable or better than N-Chance, even for the best choice of N, and uses at least 11% less energy than LRU. For read dominated benchmarks, we find that AL and VA are comparable to LRU, while N-Chance (using the N that was best for benchmarks that were a mixture of reads and writes) uses at least 20% more energy.

II. Related Work

The design and analysis of AL is heavily indebted to the design and analysis of the Landlord algorithm for browser caching [11]. In browser caching, there is a cost and a size associated with each page. There are two main differences between browser caching, and caching for PCM. The first is that the costs in browser caching can take on any value, not just 1 or c. This would seem to make browser caching harder than PCM caching. The second difference is that the page cost in PCM is time dependent, as it can rise from 1 to c when a previously clean page is written to. This would seem to make PCM caching harder than browser caching. And in fact, while the problem of optimal browser caching with equal sized files can be computed by a min-cost flow computation [12], we do not know how to compute the optimal PCM caching cost with a polynomial time algorithm. The main conceptual difficulty in adapting the Landlord results in [11] to our purposes was that in PCM caching a cost can be incurred without a change in the contents of cache (when a page is dirtied).

One might also wish to limit the number of writes to reduce wear in a memory technology where durability is an issue, such as flash or PCM. [13] develops an algorithm similar to N-Chance for flash memory, with part of the motivation being the asymmetric cost of reads and writes in flash memory.

III. Theoretical Development of the Asymmetric Landlord Algorithm

A. Problem Model

Consider the following model for the problem of *asymmetric weighted page caching* to minimize total energy consumption. There is a fully associative cache of size k, and we are given a sequence of requests $I = r_1, r_2, \ldots r_m$ where $r_i = (f_i, s)$ denotes a request for page f_i with $s \in \{r, w\}$ denoting a read or write. There is a read cost,

normalized to 1, representing the cost in terms of energy of reading a page from slow memory (PCM). There is a concave function $\mathcal{C}$ mapping the number of writes to a page to the cost in terms of energy of writing it to slow memory. An algorithm must satisfy each request by bringing the requested page into cache and when necessary evicting other pages from cache to maintain that the number of pages in cache is no more than k. All requests are received on-line, meaning an algorithm must make a decision for request r_i without knowledge of future requests r_j, where $j > i$. We assume that there is no cost for reading or writing to a page in cache, however, any page that is written to s times while in cache must pay the write cost $\mathcal{C}(s)$ when evicted. For convenience, we define the incremental cost of the i^{th} write to be $c_i = \mathcal{C}(i) - \mathcal{C}(i-1)$. We say a page has "dirtiness i" if it has been written to i times without being evicted. For an on-line algorithm A, and a request sequence I, let $\mathcal{A}_k(I)$ denote the cost of algorithm A with cache size k, where the cost is the total energy consumed by A in satisfying all requests in I. We say that an algorithm A is (k/h)-cache c-competitive if

$$\max_I \frac{\mathcal{A}_k(I)}{\mathcal{O}_h(I)} \leq c$$

where $\mathcal{O}_h(I)$ is the optimal cost using a cache of size $h \leq k$.

The algorithm Asymmetric Landlord (AL) was informally described in the introduction; a formal description may be found in the adjacent figure. The one situation not discussed in the informal description was how to handle the case that a page in cache with dirtiness i is written to. In this case AL increments the age of this page by c_{i+1}. In Theorem 1 we show that Asymmetric Landlord is (k/h)-cache $(k/(k-h+1))$-competitive. Even for symmetric read and write costs, this is the best achievable ratio [14].

Algorithm 1 Asymmetric Landlord

When there is a request (g, s)
if g is not in the cache **then**
    $\Delta = \min_{f \in \text{cache}} \text{TTL}[f]$
    For each page f in cache, decrease TTL$[f]$ by Δ
    Evict the oldest page f such that TTL$[f] = 0$
    Bring g into the cache
    if $s = r$ **then** Set TTL$[g] = 1$
    else Set TTL$[g] = c_1 + 1$
else
    Case 1: g has dirtiness i and $s = w$ **then** Set TTL$[g] = \max(\text{TTL}[g] + c_{i+1}, c_1 + 1)$
    Case 2: g is clean and $s = r$ **then** Set TTL$[g] = 1$
    Case 3: g is dirty and $s = r$ **then** Do nothing

Theorem 1. *Asymmetric Landlord (AL) is (k/h)-cache $\frac{k}{k-h+1}$-competitive for asymmetric weighted page caching.*

Proof: We use a potential function argument very similar to the one found in [11]. That is we find a function Φ that maps the state of AL and the state of the optimal solution to a nonnegative integer, where Φ is zero if the cache is empty in both states. Further we need that for every access:

$$(k-h+1) \cdot \Delta\mathcal{A}_k + \Delta\Phi \leq k \cdot \Delta\mathcal{O}_h \tag{1}$$

where Δ denotes the change due to that request. The claim then follows by summing up these inequalities over all requests, and noting that the various terms telescope.

To define the potential function Φ that we use, let AL be the set of pages in the cache of Asymmetric Landlord, OPT be the set of pages in the Optimal's cache, and OPT$_l \subseteq$ OPT be the set of pages in Optimal's cache that

have dirtiness l or higher. By convention, we assume that for any page f not in the cache $\text{TTL}[f] = 0$. Define the potential function

$$\Phi = (h-1) \cdot \sum_{f \in \text{AL}} \text{TTL}[f] + k \cdot \left(\sum_{f \in \text{OPT}} \max\Big(1 + \sum_{\substack{l: \\ f \in \text{OPT}_l}} c_l - \text{TTL}[f], 0\Big) \right) \tag{2}$$

Clearly Φ is initially 0 when the cache is empty, and $\Phi \geq 0$ since the max term is at least 0. For simplicity, we split up write requests into a read followed by a write (any requested page not in the cache is always read first, so writes are always performed on pages in the cache). When there is a write request to a page having dirtiness l for either algorithm, we assume that the cost of c_{l+1} is paid immediately as opposed to when the page is evicted. In order to prove inequality (1), we show that:

- if Optimal brings a (clean) page into cache, Φ increases by at most k
- if Asymmetric Landlord brings a page into cache, Φ decreases by at least $k - h + 1$ (regardless of whether or not it is a write)
- if there is a write to a page having dirtiness l_1 in Optimal and l_2 in Landlord, Φ increases by at most $kc_{l_1+1} - (k-h+1)c_{l_2+1}$
- at all other times Φ does not increase

The total effect of each request on Φ can be broken down into steps, and we analyze the effect of each step on Φ. Note that each step assumes all previous steps have completed.

- *Optimal evicts a page f*: Since this just removes a term from the sums of (2) and each term is positive, Φ cannot increase.
- *Optimal retrieves a page g*: Optimal pays the read cost 1 (if it is a write request, the write is performed in a future step). Since $\text{TTL}[g] \geq 0$, Φ can increase by at most k.
- *Asymmetric Landlord decreases* $\text{TTL}[f]$ *for all $f \in AL$*: All TTL's are decreased by the same amount, call it Δ. The first term in Φ decreases by $\Delta(h-1)k$. The second term of Φ increases by at most Δk for each page in both OPT and AL. Therefore, the net decrease in Φ is at least Δ times $(h-1)k - k \cdot \text{size}(\text{OPT} \cap \text{AL})$ where $\text{size}(\text{OPT} \cap \text{AL})$ denotes the number of pages that appear in both OPT and AL. We know the requested page g is in OPT but not AL, so $\text{size}(\text{OPT} \cap \text{AL}) \leq h - 1$. Therefore, the decrease in the potential function is at least Δ times $(h-1)k - k(h-1) = 0$ and thus Φ does not increase.
- *Asymmetric Landlord evicts a page f*: Asymmetric Landlord only evicts f when $\text{TTL}[f] = 0$. Thus Φ is unchanged.
- *Asymmetric Landlord retrieves the requested page g and sets* $\text{TTL}[g]$ *to* 1: In this step, Asymmetric Landlord pays the read cost of 1. Since g was not previously in the cache (and so $\text{TTL}[g]$ was zero), and because we know $g \in \text{OPT}$, Φ decreases by $-(h-1) + k = k - h + 1$.
- *Optimal writes to page g with dirtiness l_1 and Asymmetric Landlord writes to page g with dirtiness l_2 and adds c_{l_2+1} to* $\text{TTL}[g]$: Here, Optimal pays the write cost c_{l_1+1} and Asymmetric Landlord pays the write cost c_{l_2+1}. First note that an increase of $\text{TTL}[g]$ by c_{l_2+1} increses the first term of Φ by $(h-1)c_{l_2+1}$. There are two cases: either g is dirtier in Asymmetric Landlord's cache, or it is not.
  - If g is dirtier in Asymmetric Landlord's cache, then $l_2 > l_1$. When g is added to OPT_{l_1+1}, the second term of Φ increases by some amount δ_1, which is at most c_{l_1+1}. Adding c_{l_2+1} to $\text{TTL}[g]$ then decreases the second term of Φ by $\delta_2 = \min(c_{l_2+1}, \delta_1)$. Since $\mathcal{C}$ is concave, $c_{l_2+1} \leq c_{l_1+1}$, so $\delta_1 - \delta_2$ has a maximum value of $c_{l_1+1} - c_{l_2+1}$. Since the total increase to the second term of Φ is $k(\delta_1 - \delta_2) \leq k(c_{l_1+1} - c_{l_2+1})$, the total increase in Φ is at most $k(c_{l_1+1} - c_{l_2+1}) + (h-1)c_{l_2+1} = kc_{l_1+1} - (k-h+1)c_{l_2+1}$.
  - If g is not dirtier in Asymmetric Landlord's cache, then $l_2 \leq l_1$. Since before the increase to $\text{TTL}[g]$ it was the case that $\text{TTL}[g] \leq \sum_{l \leq l_2} c_l \leq \sum_{l \leq l_1} c_l$ the addition of g to OPT_{l_1+1} increases the second term of Φ by c_{l_1+1} and the increase to $\text{TTL}[g]$ decreases the second term of Φ by c_{l_2+1}, so the total increase to the second term of Φ is $k(c_{l_1+1} - c_{l_2+1})$, and thus the total increase of Φ is $kc_{l_1+1} - (k-h+1)c_{l_2+1}$.

- *Asymmetric Landlord increases TTL$[g]$ to a maximum of* 1 *if the request is a read or* $c_1 + 1$ *if the request is a write*: Again, we know $g \in$ OPT, and if the request is a write we further know $g \in \text{OPT}_1$. If TTL$[g]$ changes, it can only increase. In this case, since $h - 1 < k$, Φ decreases (though the amount of decrease depends on whether or not the request is a write).

■

B. Asymmetric Landlord with Read-throughs and Write-throughs

In this section, we consider the situation where a page need not be brought into cache to be accessed. We also assume that pages may be of different sizes, and that there is only one level of dirtiness (i.e., $c_i = 0$ for $i > 1$ and $c_1 = c$). For a page f, we denote its size by $size(f)$. The algorithm AL2 that we give is similar to Asymmetric Landlord. Intuitively, the difference is when there is a cache miss, AL2 gives the accessed page the TTL that it would have gotten had it been moved into cache, and then performs the steps of AL; If AL would then have evicted this new page, then AL2 will access the page directly from PCM memory. To accomplish this, AL2 keeps both TTL_r and a TTL_w credits for each page which are affected differently by reads and writes. The time-to-live of a page can be thought of as the sum of these two values. The pseudo-code for AL2 can be found in the adjacent figure. Note that in line 10 when decreasing the sum we must make two assignment statements to ensure neither TTL goes negative. We then claim that AL2 is k/h-cache $2(k+1)/(k-h+1)$-competitive.

Algorithm 2 Asymmetric Landlord 2

```
1:  When there is a request (g, s)
2:  if g is not in the cache then
3:      TTL_r[g] = 1
4:      if s = w then TTL_w[g] = c
5:      until |C| + size(g) ≤ k or TTL_r[g]+ TTL_w[g] = 0
6:          Λ = min_{f∈C∪g}(TTL_r[f] + TTL_w[f])/size[f]
7:          for each f ∈ C ∪ g: Decease TTL_r[f] + TTL_w[f] by Δ·size[f]
8:          Evict all f such that TTL_r[f]+ TTL_w[f] = 0
9:      if TTL_r[g]+ TTL_w[g] > 0 then Bring g into the cache
10:     else Perform requested operation directly in memory
11: else
12:     Case 1: g is clean and s = w then Set TTL_w[g] = c
13:     Case 2: g is dirty and s = w then Set TTL_w[g] = c
14:     Case 3: g is clean and s = r then Set TTL_r[g] = 1
15:     Case 4: g is dirty and s = r then Do nothing
```

Theorem 2. *Asymmetric Landlord 2 is* $2(k+1)/(k-h+1)$ *competitive for asymmetric weighted page caching with read-throughs and write-throughs.*

Proof: To prove theorem 2, we start with a definition of fake caches that we will need to fully define the potential function.

Fake Caches: The concept of a fake cache is used to maintain a consistent relationship between OPT's cache and AL2's cache even if one or both of the algorithms satisfy a request for file f with a read-through or write-through. Both OPT and AL2 have a fake cache that is empty between requests, and during a request (sometimes) holds the currently requested file. Whether a fake cache holds the currently requested file simply depends on the

actions of AL2 and OPT. Fake caches are not part of either algorithm, rather, they can be thought of as part of the potential function in that they are used for the accounting of costs only. We now describe each fake cache.

- OPT's Fake Cache: If OPT fulfils a request for file f with a read-through or write-through, then we say that OPT pays the read or write cost for f and brings f into a "fake" cache. At some point before the next request, we say that OPT then evicts f from the fake cache. When, exactly, the eviction occurs is dependent on AL2's actions for the current request, but it is always prior to the next request.
- AL2's Fake Cache: On an AL2 cache miss, we say that AL2 sets $\text{TTL}_r[f]$ to 1, puts f in AL2's "fake" cache and pays cost 1. If the request is a write then when AL2 sets $\text{TTL}_w[f]$ to c, AL2 writes to clean file f in the fake cache and pays c. The decrease step will then determine if f is brought into the actual cache or evicted from the fake cache (i.e. a read or write through is performed). In either case, the fake cache will be empty prior to the next request.

We can now define the potential function to prove our result.

$$\Phi = \frac{h}{k-h+1}\sum_{f\in AL2}(TTL_r[f]+TTL_w[f]) + \frac{k+1}{k-h+1}\Big(\sum_{f\in OPT}(1-TTL_r[f]) + \sum_{f\in OPT_d}(c-TTL_w[f])\Big)$$

where OPT and OPT_d are the set of all pages and dirty pages respectively in OPT's cache and AL2 is the set of pages in AL2's cache. OPT and OPT_d contain the file in OPT's fake cache, and likewise AL2 contain the file in AL2's fake cache.

To prove theorem 2, we will show that for every request,

$$\Delta\mathcal{A}_k + \Delta\Phi \leq 2\frac{k+1}{k-h+1}\Delta\mathcal{O}_h \tag{3}$$

To do this we start with what we call Actions, which are basic operations that either AL2 or OPT perform to fulfill a request. We will first show that equation 3 holds independent of when the Action happens or as long as some precondition is met. Then we consider what OPT and AL2 do when requests arrive. We show that when a request arrives, AL2 and OPT perform a sequence of Actions plus some additional steps, and that equation 3 holds at every point in this sequence.

Actions:

1) *OPT evicts from real or fake cache*: As $1 \geq \text{TTL}_r[f]$ and $c \geq \text{TTL}_w[f]$, Φ cannot increase.
2) *Landlord evicts*: Landlord only evicts when $\text{TTL}_r[f] + \text{TTL}_w[f] = 0$, thus Φ cannot increase.
3) *OPT reads from memory, writes to memory, or writes to a clean page in real or fake cache*:
   a) OPT reads: OPT pays 1, and Φ can increase by at most $\frac{k+1}{k-h+1}$, giving, $\Delta\Phi \leq \frac{k+1}{k-h+1} \leq 2\frac{k+1}{k-h+1}\Delta\text{OPT}_h$
   b) OPT writes: Follow the read case with cost of c instead of cost of 1.
4) *AL2 reads g from memory, writes g to memory, or writes to a clean page g in real or fake cache* (precondition: $g \in OPT$ if reading, $g \in OPT_d$ if writing):
   a) AL2 reads g and sets $\text{TTL}_r[g]$ to 1: AL2 pays a cost of 1 while Φ increases by $\frac{h}{k-h+1}$ and decreases by $\frac{k+1}{k-h+1}$ as we know $g \in OPT$. This gives,$1 + \frac{h}{k-h+1} - \frac{k+1}{k-h+1} = (1 - \frac{k-h+1}{k-h+1}) = 0$
   b) AL2 writes to clean page g and sets $\text{TTL}_w[g]$ to c: Follow the read case but with cost of c instead of cost of 1 and OPT_d instead of OPT.
5) *Landlord performs a decrease* (precondition: OPT's fake cache is empty):
   a) If $\text{size}(X)$ denotes the sum of the sizes of all files in X, then because the decrease of $\text{TTL}_r[f] + \text{TTL}_w[f]$ is Λ $\text{size}(f)$, Φ increases by $\Lambda\frac{k+1}{k-h+1}\text{size}((OPT\cup OPT_d)\cap AL2)$ and decreases by $\Lambda\frac{h}{k-h+1}\text{size}(AL2)$. Thus we get, $\Delta\Phi = \Lambda(\frac{k+1}{k-h+1}\text{size}((OPT \cup OPT_d) \cap AL2) - \frac{h}{k-h+1}\text{size}(AL2))$ However, because OPT's fake cache is empty, any file in OPT_d is also in OPT, thus $\text{size}((OPT \cup OPT_d) \cap AL2) = \text{size}(OPT \cap AL2) \leq \text{size}(OPT) \leq h$. It must be that $\text{size}(AL2) \geq k+1$, otherwise everything could

fit in AL2's cache and AL2 wouldn't perform a decrease. Thus we have that $\Delta\Phi \leq \Lambda(h\frac{k+1}{k-h+1} - (k+1)\frac{h}{k-h+1}) = 0$

We break Requests into two main cases, when AL2 has a cache hit and when AL2 has a cache miss. For the cache miss case, we have three sub-cases, if OPT does a read-through, if OPT does a write-through, or if OPT does neither (OPT will never do both for a single request). We break each case into a number of steps where each step is either an Action or a more complex step which we explain in detail. If a step is an Action, then the number at the end of a step represents the Action of the current step, thus implying 3 holds at that step. For steps consisting of an Action with a precondition, we specify why the precondition is met at that step. For the rest of the steps we describe explicitly why equation 3 holds. Lastly, note that the case of a write request to clean file in AL2's cache will be accounted for as an AL2 cache miss. This is mainly because AL2 pays a cost in this case, while in the normal AL2 cache hit case, AL2 pays no cost.

Requests:

1) *AL2 Cache Hit*:
   a) (if necessary) OPT evicts file $h \neq g$ from cache. (1)
   b) (if $g \notin OPT$) OPT reads from memory, writes to memory, or writes to a clean page in real or fake cache. (3)
   c) Landlord resets either $\text{TTL}_r[g]$ or $\text{TTL}_w[g]$: Call the amount increased in either case $\lambda \geq 0$. In both cases Φ increases by $h/(k-h+1) \cdot \lambda$. In the first case, it must be that $g \in OPT$, and in the second $g \in OPT_d$, thus in both cases, Φ decreases by $(k+1)/(k-h+1) \cdot \lambda$. Because $k+1 > h$, Φ cannot increase.
   d) (if necessary) OPT evicts from fake cache. (1)
2) *AL2 Cache Miss*:
   a) OPT doesn't perform a read-through or write-through:
      i) (if necessary) OPT evicts file $h \neq g$ from cache. (1)
      ii) (if necessary) OPT reads from memory and/or writes to a clean file in cache. (3)
      iii) AL2 reads/writes into fake or real cache: By the definition of the case, it must be that $g \in OPT$ and if the request is a write, $g \in OPT_d$ also, thus we can apply 4.
      iv) (if necessary) AL2 performs a decrease step: By the definition of the case, OPT's fake cache is empty, thus we can apply 5.
      v) (if necessary) AL2 evicts. (2)
   b) OPT does a read-through:
      i) OPT reads into fake cache. (3)
      ii) AL2 reads into fake cache: By the definition of the case and the previous step, it must be that $g \in OPT$, thus we can apply 4.
      iii) OPT evicts from fake cache. (1)
      iv) (if necessary) AL2 decreases: By the previous step, OPT's fake cache is empty, thus we can apply 5.
      v) (if necessary) AL2 evicts. (2)
   c) OPT does a write-through:
      i) We combine the case when OPT writes into fake cache and when AL2 reads/writes into fake cache. Here $\Delta\text{OPT}_h = c$ and $\Delta\text{AL2}_k = c+1$. When OPT writes, Φ increases by $\frac{k+1}{k-h+1}c$, when $AL2$ does the read, Φ increases by $\frac{h}{k-h+1}$. When $AL2$ does the write, Φ decreases by $\frac{k+1}{k-h+1}c$ and increases by $\frac{h}{k-h+1}c$. After cancelling we have that $\Delta\text{AL2}_k + \Delta\Phi$ is bounded by $c + 1 + \frac{h}{k-h+1} + \frac{h}{k-h+1}c \leq 2\frac{k+1}{k-h+1}\Delta\text{OPT}_h$.
      ii) OPT evicts from fake cache. (1)

iii) (if necessary) LL decreases: By the previous step, OPT's fake cache is empty, thus we can apply 5.
iv) (if necessary) LL evicts. (2)

We have shown that for all requests, equation 3 holds. The desired result then follows by summing over all requests.

■

IV. Experimental methodology

A. PCM Main Memory Architecture

Figure 1(a) shows the main memory architecture that we consider throughout the paper. Memory operations issued by the CPU are serviced by L1 and L2 caches. Misses to the L2 cache, as well as writebacks from the L2 cache, are sent to the L3 cache, which is the last-level cache. The LLC works as a traditional write-allocate cache with a write-back policy (to PCM). That is, when a modified cache line is evicted from the LLC, it must be written to the PCM. A primary benefit the LLC provides to PCM main memory is that by coalescing a sequence of writes to the same line in the cache, the LLC partially mitigates the negative impacts of PCM writes.

Figure 1(b) presents the organization of PCM main memory that we consider, with 2 channels of 8 banks each. Each bank has an 8-entry read queue (RDQ) and a 16-entry write queue (WRQ) for pending requests. When a writeback of a cache page from the LLC occurs, a write request is sent to the PCM, which is queued into a write queue. The application progresses without delay if the write queue has available entries since the writebacks are not on the critical path. Otherwise, the application stalls.

B. Methodology and Experimental Setup

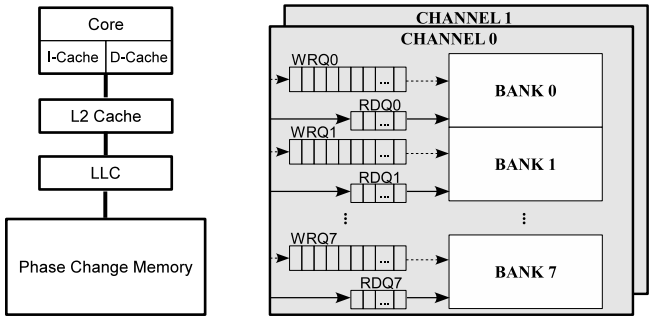
We use Simics [15] to model the processor, L1 and L2 caches, and generate memory traces, which are input to an in-house cycle-accurate simulator that models the LLC and PCM. The memory trace contains, for each memory request by the CPU, the time stamp (assuming zero memory latency, that is, counting only CPU cycles to execute the task and L1/L2 cache latency), the type of request (read vs. write) and the physical address of the memory reference.

Table I shows the simulation parameters. The processor issues two instructions per cycle, and has an 84-entry instruction window. Our baseline system has a three level cache hierarchy. The L1 instruction and data cache is a 4-way 64KB cache and the L2 cache is a unified 2MB 8-way associative cache. We evaluated PCM main memory system with different LLC sizes (4 and 8MB), and the results are similar. For the rest of the paper, we only show results for the PCM main memory system with a 8MB LLC. We assume the PCM write energy cost is 10x of PCM read. In addition, we provide a sensitivity study on the relative energy cost of PCM write. We use the permutation-based page interleaving scheme [16] as the address mapping scheme for phase change main memory. Assuming there are 2^q write queues, the lower order q bits of the LLC tag and the lower order q bits of the LLC set index are used as the input to a q-bit bitwise XOR logic to generate the write queue index.

We use the SPEC CPU2006 [17] benchmarks for evaluation. Notice that we exclude benchmarks with small LLC miss rates from the evaluation (e.g., *400.perlbench*, *416.gamess*, *434.zeusmp*, and *435.gromacs*). Each benchmark was run in Simics for 2 billion instructions.

V. Evaluation Results

To understand the effectiveness of our schemes, we compare Asymmetric Landlord (AL) and Variable Aging (VA) to a previously proposed PCM asymmetry-aware LLC replacement policy, N-Chance [6].



(a) Architecture (b) Organization of PCM

Figure 1: PCM main memory architecture

A. Number of PCM Reads and Writes

Figure 2 shows the number of PCM reads and writes of 4-, 8-, 12-, 16-Chance, AL and VA normalized to LRU. The bar labeled "average" is the average result of all 15 benchmarks. In general, picking a large parameter N for N-Chance reduces the number of PCM writes at the cost of incurring more PCM reads. However, the impact of the parameter N on the number of PCM reads and writes depends on benchmarks.

To better understand this impact, we categorize the benchmarks into three groups. For *gcc* and *mcf*, N-Chance reduces the number of PCM reads as well as writes. This is because the dirty pages of these benchmarks are frequently accessed, and keeping the frequently accessed dirty pages is good for reducing the LLC miss rate and writeback rate. As a result, 16-Chance is always the best choice. On the contrary, for *GemsFDTD*, *lbm*, and *xalancbmk*, N-Chance slightly reduces the number of PCM writes, while increasing the number of PCM reads substantially. Notice that even though 8-Chance eliminates all the PCM writes for *xalancbmk*, the benefit is limited due to the fact that memory accesses of *xalancbmk* are dominated by read references (i.e., $\geq 98\%$). In this case, we should pick $N = 1$. Lastly, for other benchmarks such as *milc*, *cactusADM*, and *leslie3d*, N-Chance reduces the number of PCM writes with comparable increase in the number of PCM reads. In this case, it is difficult to identify the value of N which leads to the best energy efficiency. In summary, there exists no universal value of the parameter N for N-Chance policy that minimizes the energy consumptions for applications. Table II summarizes these three benchmark categories.

On average, 4-, 8-, 12-, and 16-Chance increase the number of PCM reads by 3%, 5%, 21%, and 35%, and reduce the number of PCM writes by 22%, 30%, 33%, and 35% over LRU, respectively. As one might imagine,

Processor	4GHz, Out-of-Order 84-entry instruction window 2 instructions per cycle in each core
L1 I-, D-cache	64KB, 64B line, 4-way, LRU
L2 Cache	2MB, 256B line, 8-way, LRU 15 cycles access latency
LLC	8MB, 256B line, 16-way, writeback policy 50 cycles access latency
Main Memory	32GB PCM, 2 channels of 8-banks each 16 entry write queue per bank 8 entry read queue per bank read priority scheduling via write pausing [18]
PCM	read: 200 cycles (50ns) latency, 1 J/GB energy write: 4000 cycles (1μs) latency, 10 J/GB energy

Table I: Baseline configuration

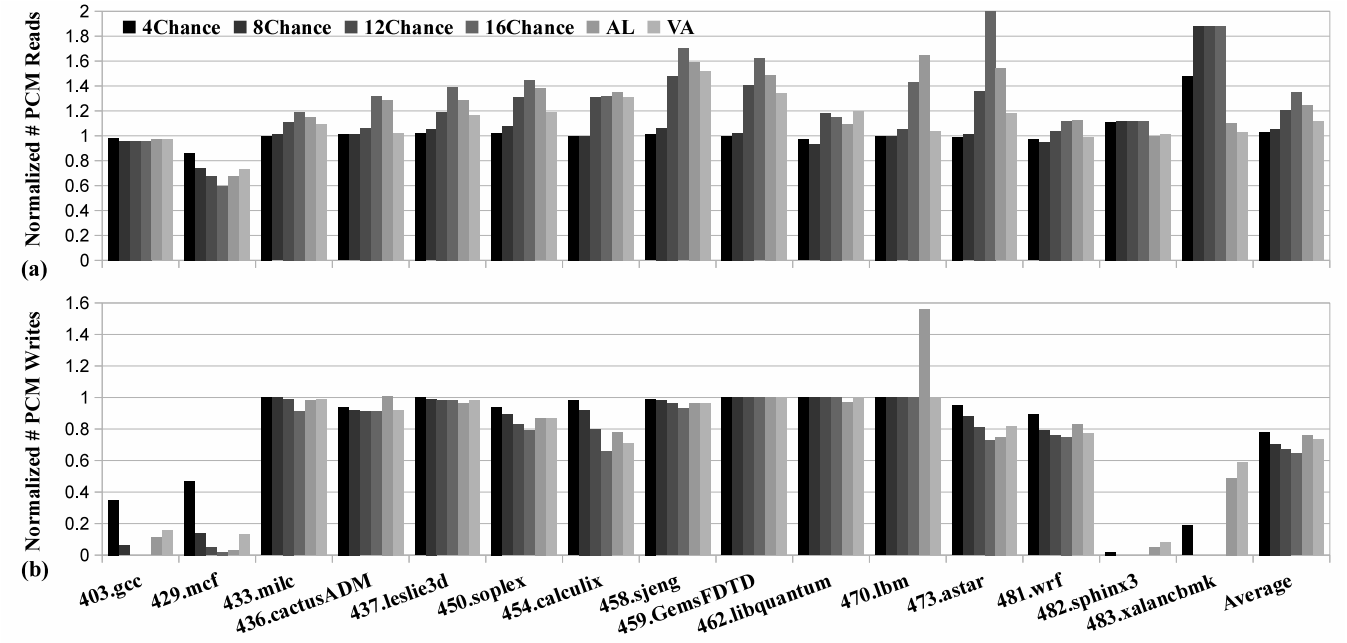


Figure 2: Impact on number of (a) PCM reads, and (b) PCM writes

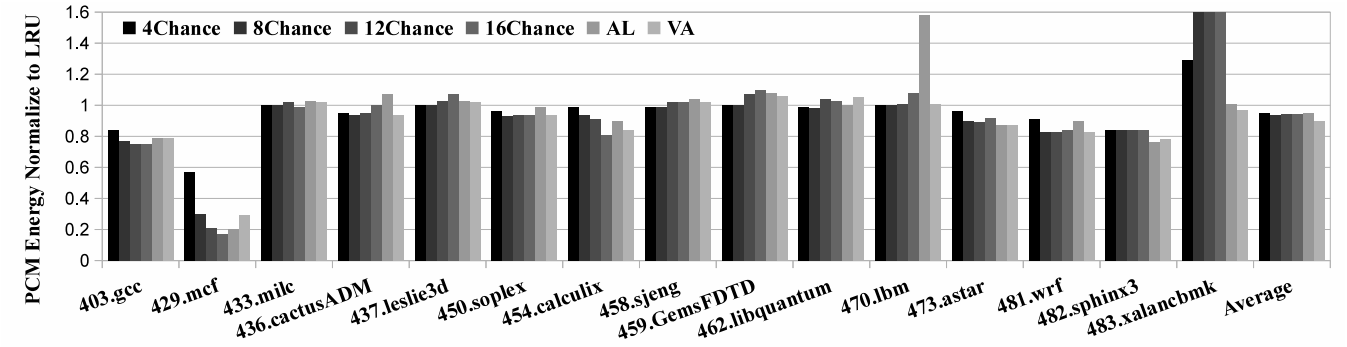


Figure 3: Impact on PCM energy consumption

N-Chance will tend to rigidly reserve part of the LLC space to preferably store dirty pages, effectively reducing the LLC cache size therefore hurting the cache hit rate. For instance, for *xalancbmk*, N-Chance saves limited number of PCM writes (i.e., the total number of dirty pages is limited), but increases the number of PCM reads by up to 88%, compared to LRU. Compared to N-Chance, AL and VA do a better job in balancing the trade-off between PCM write reduction and PCM read increase, and adapting to various applications. Instead of rigidly reserving partial LLC capacity for dirty pages, AL and VA make replacement decisions based on liveliness and age information: AL assigns different TTL values to clean and dirty pages; VA ages clean and dirty pages at different rates. These mechanisms ensure that AL and VA do not suffer from the instances mentioned above. On average, AL and VA only increase the number of PCM reads by 11% and 3% for *xalancbmk* over LRU, respectively.

B. PCM Energy Consumption

Figure 3 shows the PCM energy consumption of 4-, 8-, 12-, 16-Chance, AL and VA normalized to LRU. Overall, N-Chance and our proposed policies save energy. For example, for *mcf*, 16-Chance outperforms LRU

Group	Benchmarks	Impact of N-Chance	Best N
1	*gcc*, *mcf*	Reduces #reads, and #writes in PCM	16
2	*GemsFDTD*, *lbm*, xalancbmk	Increases #reads in PCM	1
3	*milc*, *cactusADM*, *leslie3d*, *soplex*, *calculus*, *sjeng*, *libquantum*, *wrf*, *astar*, *sphinx3*	Reduces #writes, but increases #reads in PCM	Undetermined

Table II: The best choice of parameter N for N-Chance

by 83% due to the 40% reduction in the number of PCM reads and the 98% reduction in the number of PCM writes. AL and VA also save 80% and 91% of the PCM energy consumption for *mcf*. As we pointed out, N-Chance fails to pick a universal parameter that is suitable for all benchmarks. 16-Chance is the best choice for *mcf*, *milc*, and *calculix*; 12-Chance is favorable for *astar*; 8-Chance ensures the best energy efficiency for *cactusADM*, and *libquantum*; while 1-Chance (i.e., LRU) is suitable for *xalancbmk*. The benchmark *xalancbmk* represents a pathological case for N-Chance replacement policy such that any parameter N except 1 will result in large increase in energy consumption. On the contrary, AL and VA can easily adjust to the access patterns of the applications to avoid the pathological cases. For instance, for *xalancbmk*, AL results in the same energy consumption as LRU, and VA even outperforms LRU by 3%.

AL achieves comparable energy savings against the best N-Chance for most of the benchmarks except *cactusADM* and *lbm*. VA delivers comparable energy efficiency to, or even outperforms the best N-Chance. On average, 4-, 8-, 12-, 16-Chance, AS and VA reduce the PCM energy by 5%, 7%, 6%, 6%, 5%, and 10% over LRU, respectively.

Comparisons of the observed throughputs for the various policies can be found in the appendix. Our baseline configuration assumes that a PCM write consumes 10x energy than that of a PCM read [1,19]. As PCM technology advances, the relative energy cost a PCM write to a PCM read will likely fluctuate. In order to demonstrate the versatility of our schemes under different energy costs of PCM write operations, the appendix describes a study of sensitivity to the energy cost of a PCM write.

ACKNOWLEDGMENT

Supported in part by an IBM Faculty Award, and NSF grants CCF-0830558, CNS-1012070, and 1115575. We thank Neal Young and Rami Melhem for invaluable discussions.

REFERENCES

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*, 2009.

[2] D. Li, J. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "An analysis of scientific applications for using non-volatile memory in high performance computing," in *IPDPS '12*, 2012.

[3] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09*, 2009.

[4] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *MICRO 42*, 2009.

[5] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *MICRO 42*, 2009.

[6] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse, "Increasing pcm main memory lifetime," in *DATE '10*, 2010.

[7] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *CIDR '11*, 2011.

[8] A. P. Ferreira, B. Childers, R. Melhem, D. Mosse, and M. Yousif, "Using pcm in next-generation embedded space applications," in *RTAS '10*, 2010.

[9] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09*, 2009.

[10] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger, "Preventing pcm banks from seizing too much power," in *MICRO 44*, 2011.

[11] N. E. Young, "On-line file caching," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '98, 1998, pp. 82–86.

[12] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu, "Caching is hard: even in the fault model," in *Proceedings of the 18th annual European Symposium on Algorithms: Part I*, ser. ESA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 195–206.

[13] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *CASES '06*. New York, NY, USA: ACM, 2006, pp. 234–241.

[14] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, Feb. 1985.

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, February 2002.

[16] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO 33*, 2000, pp. 32–41.

[17] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006.

[18] M. Qureshi, M. Franceschini, and L. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA '10*, 9-14 2010, pp. 1 –11.

[19] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *DATE 2010*, march 2010, pp. 136 –141.

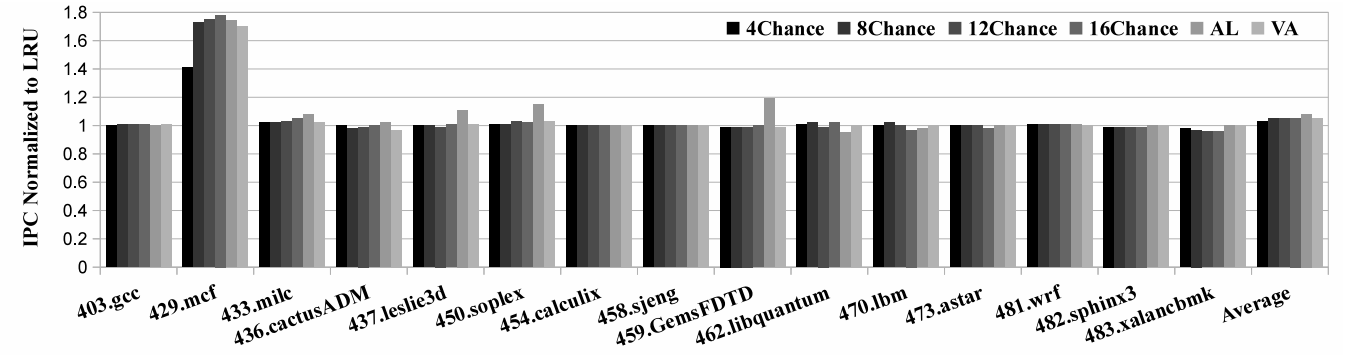


Figure 4: Impact on performance

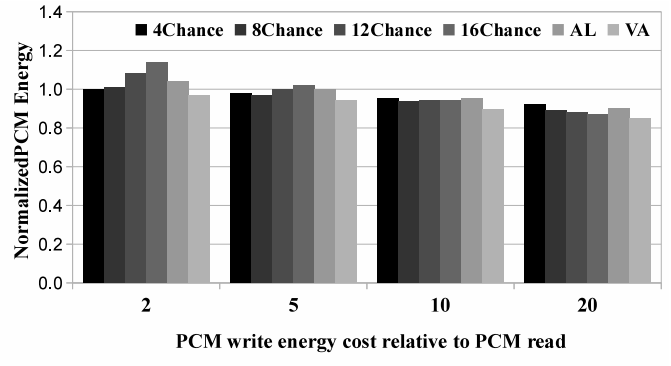


Figure 5: Sensitivity of AL and VA to the relative PCM write energy cost

APPENDIX

A. Performance

Figure 4 shows the throughput for different policies in terms of instruction per cycle (IPC). For most benchmarks, different policies perform similarly in terms of throughput due to two reasons. First, although N-Chance, AL and VA might result in high LLC miss rates delaying the processing of the application, these policies can also reduce the number of PCM writes, speeding up the application by avoiding overwhelmed PCM WRQs. Second, the application throughput might be insensitive to the LLC miss rates in the presence of Out-of-Order processing. For *mcf*, 16-Chance, AL and VA outperform LRU by at least 70% in terms of throughput. On average, AL and VA improve the throughput by 8% and 5% over LRU, respectively.

B. Sensitivity Study

Our baseline configuration assumes that a PCM write consumes 10x energy than that of a PCM read [**?**,19]. As PCM technology advances, the relative energy cost a PCM write to a PCM read will likely fluctuate. In order to demonstrate the versatility of our schemes under different energy costs of PCM write operations, this section describes a study of sensitivity to the energy cost of a PCM write. We vary the PCM write energy cost from 2x to 20x of PCM read energy. Figure 5 shows the normalized PCM energy consumption of 4-, 8-, 12-, 16-Chance, AL and VA as the PCM write energy cost varies. It only shows the average PCM energy among 15 benchmarks. All policies benefit from a higher PCM write energy cost, as the benefit of saving a PCM write becomes larger.

The best choice of parameter N for N-Chance is dependent on the relative PCM write energy cost. A small value of N (i.e., N=1) is preferable when PCM write energy is twice of PCM read energy, a medium value of N (i.e., N=8) performs the best if PCM write energy is 5x or 10x of PCM read energy, while a large value of N

(i.e., N=16) results in the best energy efficiency when a PCM write consumes 20x energy than a PCM read. This is expected as the benefit of reducing PCM writes is proportional to the relative energy cost of a PCM write.

Unlike N-Chance, AL and VA are capable of adaptively adjusting to different PCM write energy costs. AL performs comparable to the N-Chance with the best parameter, while VA outperforms N-Chance with the best parameter constantly. On average, VA reduces PCM energy consumption by 3%, 6%, 10%, and 15% over LRU when PCM write energy cost is 2x, 5x, 10x, and 20x of the read cost, respectively.