Rethinking Database Algorithms for Phase Change Memory

Shimin Chen
Intel Labs Pittsburgh
shimin.chen@intel.com

Phillip B. Gibbons
Intel Labs Pittsburgh
phillip.b.gibbons@intel.com

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

Phase change memory (PCM) is an emerging memory technology with many attractive features: it is non-volatile, byte-addressable, 2–4X denser than DRAM, and orders of magnitude better than NAND Flash in read latency, write latency, and write endurance. In the near future, PCM is expected to become a common component of the memory/storage hierarchy for a wide range of computer systems. In this paper, we describe the unique characteristics of PCM, and their potential impact on database system design. In particular, we present analytic metrics for PCM endurance, energy, and latency, and illustrate that current approaches for common database algorithms such as B^+-trees and Hash Joins are suboptimal for PCM. We present improved algorithms that reduce both execution time and energy on PCM while increasing write endurance.

1. INTRODUCTION

Phase change memory (PCM) [3, 10] is an emerging non-volatile memory technology with many attractive features. Compared to NAND Flash, PCM provides orders of magnitude better read latency, write latency and endurance,[1] and consumes significantly less read/write energy and idle power [9, 10]. It is byte-addressable, like DRAM memory, but consumes orders of magnitude less idle power than DRAM. PCM offers a significant density advantage over DRAM, which means more memory capacity for the same chip area and also implies that PCM is likely to be cheaper than DRAM when produced in mass market quantities [22]. While the first wave of PCM products target mobile handsets [24], in the near future PCM is expected to become a common component of the memory/storage hierarchy for laptops, PCs, and servers [9, 15, 22].

An important question, then, is *how should database systems be modified to best take advantage of this emerging trend towards PCM?* While there are several different proposals for how PCM will fit within the memory hierarchy [10] (as SATA/PCIe based data storage or DDR3/QPI based memory), recent computer architecture and systems studies all propose to incorporate PCM as the bulk of the system's main memory [9, 15, 22]. Thus, in the PCM-DB project [19], we are focusing on the use of PCM as the primary main memory for a database system. This paper highlights our initial findings and makes the following three contributions.

First, we describe the unique characteristics of PCM and its proposed use as the primary main memory (Section 2). Several attractive properties of PCM make it a natural candidate to replace or compliment battery-backed reliable memory for general database systems [18], and DRAM in main memory database systems [11]. However, a unique challenge arises in effectively using PCM: Compared to its read operations, PCM writes incur higher energy consumption, higher latency, lower bandwidth, and limited endurance. Therefore, we identify reducing PCM writes as an important design goal of PCM-friendly algorithms. Note that this is different from the goals of flash-friendly algorithms [2, 17], which include reducing the number of *erases* and *random writes* at much *coarser* granularity (e.g., 256KB erase blocks and 4KB flash pages).

Second, we present analytic metrics for PCM endurance, energy, and latency. While we believe that PCM may have a broad impact on database systems in general, this paper focuses on its impact on core database algorithms. In particular, we use these metrics to design PCM-friendly algorithms for two core database techniques, B^+-tree index and hash joins (Section 3). In a nutshell, these algorithms re-organize data structures and trade off an increase in PCM reads for reducing PCM writes, thereby achieving an overall improvement in all three metrics.

Third, we show experimentally, via a cycle-accurate X86-64 simulator enhanced with PCM support, that our new algorithms significantly outperform prior algorithms in terms of time, energy and endurance (Section 4), supporting our analytical results. Moreover, sensitivity analysis shows that the results hold for a wide range of PCM parameters.

The paper concludes by discussing related work (Section 5) and highlighting a few of the many interesting open research questions regarding the impact of PCM main memory on database systems (Section 6).

2. PHASE CHANGE MEMORY

In this section, we discuss PCM technology, its properties relative to other memory technologies, its proposed use as

[1] *(Write) endurance* is the maximum number of writes for each memory cell.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11)
January 9-12, 2011, Asilomar, California, USA.

Table 1: Comparison of memory technologies.

	DRAM	PCM	NAND Flash	HDD
Read energy	0.8 J/GB	1 J/GB	1.5 J/GB [28]	65 J/GB
Write energy	1.2 J/GB	6 J/GB	17.5 J/GB [28]	65 J/GB
Idle power	∼100 mW/GB	∼1 mW/GB	1–10 mW/GB	∼10 W/TB
Endurance	∞	$10^6 - 10^8$	$10^4 - 10^5$	∞
Page size	64B	64B	4KB	512B
Page read latency	20-50ns	$\sim$ 50ns	$\sim 25\ \mu s$	$\sim$ 5 ms
Page write latency	20-50ns	$\sim 1\ \mu s$	$\sim 500\ \mu s$	$\sim$ 5 ms
Write bandwidth	∼GB/s per die	50-100 MB/s per die	5-40 MB/s per die	∼200MB/s per drive
Erase latency	N/A	N/A	$\sim$ 2 ms	N/A
Density	1×	$2-4\times$	4×	N/A

Note: The table contents are based mainly on [10, 15, 22].

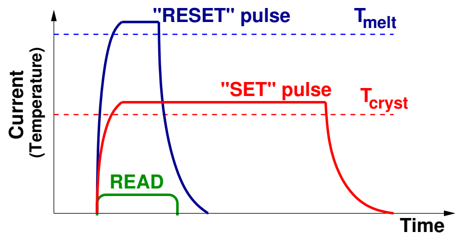


Figure 1: Currents and timings (not to scale) for SET, RESET, and READ operations on a PCM cell. For phase change material $Ge_2Sb_2Te_5$, $T_{melt} \approx 610°C$ and $T_{cryst} \approx 350°C$.

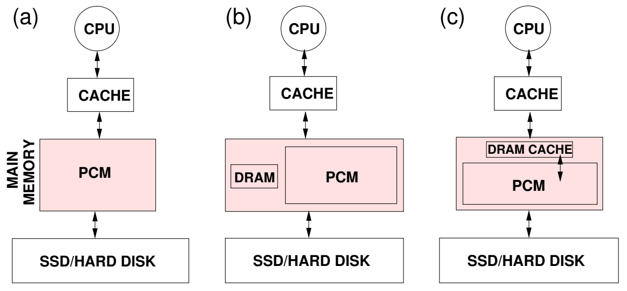


Figure 2: Candidate main memory organizations with PCM.

the primary main memory, and the key challenge of overcoming its write limitations.

2.1 PCM Technology

Phase change memory (PCM) is a byte-addressable non-volatile memory that exploits large resistance contrast between amorphous and crystalline states in so-called phase change materials such as chalcogenide glass. The difference in resistance between the high-resistance amorphous state and the low-resistance crystalline state is typically about five orders of magnitude and can be used to infer logical states of binary data (high represents 0, low represents 1).

Programming a PCM device involves application of electric current, leading to temperature changes that either SET or RESET the cell, as shown schematically in Figure 1. To SET a PCM cell to its low-resistance state, an electrical pulse is applied to heat the cell above the crystalization temperature T_{cryst} (but below the melting temperature T_{melt}) of the phase change material. The pulse is sustained for a sufficiently long period for the cell to transition to the crystalline state. On the other hand, to RESET the cell to its high-resistance amorphous state, a much larger electrical current is applied in order to increase the temperature above T_{melt}. After the cell has melted, the pulse is abruptly cut off, causing the melted material to quench into the amorphous state. To READ the current state of a cell, a small current that does not perturb the cell state is applied to measure the resistance. At normal temperatures ($< 120°C \ll T_{cryst}$), PCM offers many years of data retention.

2.2 Using PCM in the Memory Hierarchy

To see where PCM may fit in the memory hierarchy, we need to know its properties. Table 1 compares PCM with DRAM (technology for today's main memory), NAND flash (technology for today's solid state drives), and HDD (hard disk drives), showing the following points:

- Compared to DRAM, PCM's read latency is close to that of DRAM, while its write latency is about an order of magnitude slower. PCM offers a density advantage over DRAM. This means more memory capacity for the same chip area, or potentially lower price per capacity. PCM is also more energy-efficient than DRAM in idle mode.

- Compared to NAND Flash, PCM can be programmed in place regardless of the initial cell states (i.e., without Flash's expensive "erase" operation). Therefore, its sequential and random accesses show similar (far superior) performance. Moreover, PCM has orders of magnitude higher write endurance than Flash.

Because of these attractive properties, PCM is being incorporated in mobile handsets [24], and recent computer architecture and systems studies have argued that PCM is a promising candidate to be used in main memory in future mainstream computer systems [9, 15, 22].

Figure 2 shows three alternative proposals in recent studies for using PCM in the main memory system [9, 15, 22]. Proposal (a) replaces DRAM with PCM to achieve larger main memory capacity. Even though PCM is slower than DRAM, clever optimizations have been shown to reduce application execution time on PCM to within a factor of 1.2 of that on DRAM [15]. Both proposals (b) and (c) include a small amount of DRAM in addition to PCM so that frequently accessed data can be kept in the DRAM buffer to improve performance and reduce PCM wear. Their difference is that proposal (b) gives software explicit control of the DRAM buffer [9], while proposal (c) manages the DRAM buffer as another level of transparent hardware cache [22]. It has been shown that a relatively small DRAM buffer (3% size of the PCM storage) can bridge most of the latency gap between DRAM and PCM [22].

2.3 Challenge: Writes to PCM Main Memory

One major challenge in effectively using PCM is overcoming the relative limitations of its write operations. Compared to its read operations, PCM writes incur higher energy consumption, higher latency, lower bandwidth, and limited endurance, as discussed next.

- *High energy consumption:* Compared to reading a PCM cell, a write operation that SETs or RESETs a PCM cell draws higher current, uses higher voltage, and takes longer time (Figure 1). A PCM write often consumes 6–10X more energy than a read [15].
- *High latency and low bandwidth:* In a PCM device, the write latency of a PCM cell is determined by the (longer) SET time, which is about 3X slower than a read operation [15]. Moreover, many PCM prototypes support "iterative writing" of a limited number of bits per iteration in order to limit the instantaneous current level. Several prototypes support $\times 2$, $\times 4$, and $\times 8$ write modes in addition to the fastest $\times 16$ mode [3]. This limitation is likely to hold in the future as well, especially for PCM designed for power-constrained platforms. Because of the limited write bandwidth, writing 64B of data often requires several rounds of writing, leading to the ~ 1 μs write latency in Table 1.
- *Limited endurance:* Existing PCM prototypes have a write endurance ranging from 10^6 to 10^8 writes per cell. With a good wear-leveling algorithm, a PCM main memory can last for several years under realistic workloads [21]. However, because such wear-leveling must be done at the memory controller, the wear-leveling algorithms need to have small memory footprints and be very fast. Therefore, practical algorithms are simple and in many cases, their effectiveness significantly decreases in the presence of extreme hot spots in the memory. For example, even with the wear-leveling algorithms in [21], continuously updating a counter in PCM in a 4GHz machine with 16GB PCM could wear a PCM cell out in about 4 months (without wear-leveling, the cell could wear out in less than a minute).

Recent studies proposed various hardware optimizations to reduce the number of PCM bits written [8, 15, 31, 32]. For example, the PCM controller can perform *data comparison writes*, where a write operation is replaced with a *read-modify-write* operation in order to skip programming unchanged bits [31]. Another proposal is *partial writes* for only dirty words [15]. In both optimizations, when writing a chunk of data in multiple iterations, the set of bits to write in every iteration is often hard-wired for simplicity; if all the hard-wired bits of an iteration are unchanged, the iteration can be skipped [7]. However, these architectural optimizations reduce the volume of writes by only a factor ~ 3. We believe that applications (such as databases) can play an important role in complementing such architectural optimizations by carefully choosing their algorithms and data structures in order to reduce the number of writes, even at the expense of additional reads.

Table 2: General Terms.

Term	Description	Example
E_{rb}	Energy consumed for reading a PCM bit	2 pJ
E_{wb}	Energy consumed for writing a PCM bit	16 pJ
T_l	Latency of accessing a cache line from PCM	230 cycles
T_w	Additional latency of writing a word to PCM	450 cycles
C	size in bytes of the largest CPU cache	8 MB
L	Cache line size in bytes	64B
W	Word size used in PCM writes	8B
N_l	Number of cache line fetches from PCM	-
N_{lw}	Number of cache line write backs to PCM	-
N_w	Number of words written to PCM	-
γ	Avg number of modified bits per modified word	-

3. PCM-FRIENDLY DB ALGORITHMS

In this section, we consider the impact of PCM on database algorithm design. Specifically, reducing PCM writes becomes an important design goal. We discuss general considerations in Section 3.1. Then we re-examine two core database techniques, B$^+$-tree index and hash joins, in Sections 3.2 and 3.3, respectively.

3.1 Algorithm Design Considerations

Section 2 described three candidate organizations of future PCM main memory, as shown in Figure 2. Their main difference is whether or not to include a transparent or software-controlled DRAM cache. For algorithm design purposes, we consider an abstract framework that captures all three candidate organizations. Namely, we focus on a PCM main memory, and view any additional DRAM as just another (transparent or software-controlled) cache in the hierarchy. This enables us to focus on PCM-specific issues.

Because PCM is the primary main memory, we consider algorithm designs in main memory. There are two traditional design goals for main memory algorithms: (i) low computation complexity, and (ii) good CPU cache performance. Power efficiency has recently emerged as a third design goal. Compared to DRAM, one major challenge in designing PCM-friendly algorithms is to cope with the asymmetry between PCM reads and PCM writes: PCM writes consume much higher energy, take much longer time to complete, and wear out PCM cells (recall Section 2). Therefore, one important design goal of PCM-friendly algorithms is to *minimize PCM writes*.

What granularity of writes should we use in algorithm analysis with PCM main memory: *(a) bits, (b) words, or (c) cache lines*? All three granularities are important for computing PCM metrics. Choice (a) impacts PCM endurance. Choices (a) and (c) affect PCM energy consumption. Choice (b) influences PCM write latency. The drawback of choice (a) is that the relevant metric is the number of *modified* bits (recall that unmodified bits are skipped); this is difficult to estimate because it is often affected not only by the structure of an algorithm, but also by the input data. Fortunately, there is often a simple relationship between (a) and (b). Denote γ as the average number of modified bits per modified word. γ can be estimated for a given input. Therefore, we focus on choices (b) and (c).

Let N_l (T_l) be the number (latency, resp.) of cache line fetches (a.k.a. cache misses) from PCM, N_{lw} be the number of cache line write backs to PCM, and N_w (T_w) be the number (latency, resp.) of modified words written. Let E_{rb} (E_{wb}) be the energy for reading (writing, resp.) a PCM bit. Let L be the number of bytes in a cache line. (Table 2 summarizes the notation used in this paper.) We model the key PCM metrics as follows:

- TotalWear: $NumBitsModified = \gamma N_w$
- $Energy = 8L(N_l + N_{lw})E_{rb} + \gamma N_w E_{wb}$
- $TotalPCMAccessLatency = N_l T_l + N_w T_w$

The total wear and energy computations are straightforward. The latency computation requires explanations. The first part ($N_l T_l$) is the total latency of cache line fetches from PCM. The second part ($N_w T_w$) is the estimated impact of cache line write backs to PCM on the total time. In a traditional system, the cache line write backs are performed asynchronously in the background and often completely hidden. Therefore, algorithm analysis typically ignores the write backs. However, we find that because of the asymmetry of writes and reads, PCM write latency can keep PCM busy for a sufficiently long time to stall front-end cache line fetches significantly. A PCM write consists of (i) a read of the cache line from PCM to identify modified words then (ii) writing modified words in possibly multiple rounds. The above computation includes (ii) as $N_w T_w$, while the latency of (i) ($N_{lw} T_l$) is ignored because it is similar to a traditional cache line write back and thus likely to be hidden.

3.2 B$^+$-Tree Index

As case studies, we consider two core database techniques for memory-resident data, B$^+$-trees (in the current subsection) and hash joins (in the next subsection), where the main memory is PCM instead of DRAM.

B$^+$-trees are preferred index structures for memory-resident data because they optimize for CPU cache performance. Previous studies recommend that B$^+$-tree nodes be one or a few cache lines large and aligned at cache line boundaries [5, 6, 12, 23]. For DRAM-based main memory, the costs of search/insertion/deletion are similar except in those cases where insertions/deletions incur node splits/merges in the tree. In contrast, for PCM-based main memory, even a normal insertion/deletion that modifies a single leaf node can be more costly than a search in terms of total wear, energy, and elapsed time, because of the writes involved.

We would like to preserve the good CPU cache performance of B$^+$-trees while reducing the number of writes. A cache-friendly B$^+$-tree node is typically implemented as shown in Figure 3(a), where all the keys in the node are sorted and packed, and a counter keeps track of the number of valid keys in the array. The sorted key array is maintained upon insertions and deletions. In this way, binary search can be applied to locate a search key. However, on average, half of the array must be moved to make space for insertions and deletions, resulting in a large number of writes. Suppose that there are K keys and K pointers in the node, and every key, every pointer, and the counter have size equal to the word size W used in PCM writes. Then an insertion/deletion in the sorted node incurs $2(K/2) + 1 = K + 1$ word writes on average.

To reduce writes, we propose two simple unsorted node organizations as shown in Figures 3(b) and 3(c):

- **Unsorted**: As shown in Figure 3(b), the key array is still packed but can be out of order. A search has to scan the array sequentially in order to look for a match or the next smaller/bigger key. On the other hand, an insertion can simply append the new entry to the end of the array, then increment the counter. For a deletion, one can overwrite the entry to delete with the last entry in the array, then decrement the counter. Therefore, an insertion/deletion incurs 3 word writes.
- **Unsorted with bitmap**: We further improve the unsorted organization by allowing the key array to have holes. The counter field is replaced with a bitmap recording valid locations. An insertion writes the new entry to an empty location and updates the bitmap, using 3 word writes, while a deletion updates only the bit in the bitmap, using 1 word write. A search incurs the instruction overhead of a more complicated search process. For 8-byte keys and pointers, a 64-bit bitmap can support nodes up to 1024 bytes large, which is more than enough for supporting typical cache-friendly B$^+$-tree nodes.

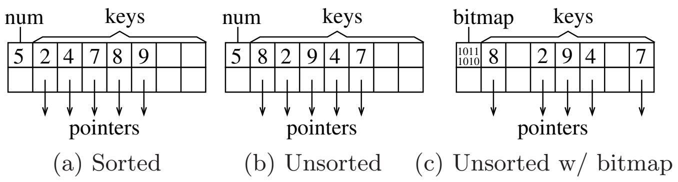


Figure 3: B$^+$-tree node organizations.

Table 3: Terms used in analyzing hash joins.

Term	Description
M_R, M_S	Number of records in relation R and S, respectively
L_R, L_S	Record sizes in relation R and S, respectively
N_{hR}	Number of cache line accesses per hash table visit when building the hash table on R records
N_{hS}	Number of cache line accesses per hash table visit when probing the hash table for S records
$HashTable_{lw}$	Number of line write backs per hash table insertion
$HashTable_w$	Number of words modified per hash table insertion
$MatchPerR$	Number of matches per R record
$MatchPerS$	Number of matches per S record

Given the pros and cons of the three node organizations, we study the following four variants of B$^+$-trees:

- **Sorted:** a normal cache-friendly B$^+$-tree. All the non-leaf and leaf nodes are sorted.
- **Unsorted:** a cache-friendly B$^+$-tree with all the non-leaf and leaf nodes unsorted.
- **Unsorted leaf:** a cache-friendly B$^+$-tree with sorted non-leaf nodes but unsorted leaf nodes. Because most insertions/deletions do not modify non-leaf nodes, the unsorted leaf nodes may capture most of the benefits.
- **Unsorted leaf with bitmap:** This variant is the same as unsorted leaf except that leaf nodes are organized as unsorted nodes with bitmaps.

Our experimental results in Section 4 show that the unsorted schemes can significantly improve total wear, energy consumption, and run time for insertions and deletions. Among the three unsorted schemes, unsorted leaf is the best for index insertions and it incurs negligible index search overhead, while unsorted leaf with bitmap achieves the best index deletion performance.

3.3 Hash Joins

One of the most efficient join algorithms, hash joins are widely used in data management systems. Several cache-friendly variants of hash joins are proposed in the literature [1, 4, 27]. Most of these algorithms are based on the

Algorithm 1 Existing algorithm: simple hash join.

```
Build phase:
 1: for ($i = 0$; $i < M_R$; $i$++) do
 2:    $r$= record $i$ in Relation $R$;
 3:    insert $r$ into hash table;
Probe phase:
 1: for ($j = 0$; $j < M_S$; $j$++) do
 2:    $s$= record $j$ in Relation $S$;
 3:    probe $s$ in the hash table;
 4:    if there are match(es) then
 5:       generate join result(s) from the matching records;
 6:       send join result(s) to the upper-level operator;
```

Algorithm 2 Existing algorithm: cache partitioning.[2]

```
Partition phase:
 1: $htsize = M_R*$ hash_table_per_entry_metadata_size;
 2: $P = \lceil (M_R L_R + M_S L_S + htsize)/C \rceil$;
 3: for ($i = 0$; $i < M_R$; $i$++) do {partition $R$}
 4:    $r$= record $i$ in Relation $R$;
 5:    $p$= hash($r$) modulo $P$;
 6:    copy $r$ to partition $R_p$;
 7: for ($j = 0$; $j < M_S$; $j$++) do {partition $S$}
 8:    $s$= record $j$ in Relation $S$;
 9:    $p$= hash($s$) modulo $P$;
10:    copy $s$ to partition $S_p$;
Join phase:
 1: for ($p = 0$; $p < P$; $p$++) do
 2:    join $R_p$ and $S_p$ using simple hash join;
```

following two representative algorithms. (Table 3 defines the terms used in describing and analyzing the algorithms.)

Simple Hash Join. As shown in Algorithm 1, in the build phase, the algorithm scans the smaller build relation R. For every build record, it computes a hash code from the join key, and inserts the record into the hash table. In the probe phase, the algorithm scans the larger probe relation S. For every probe record, it computes the hash code, and probes the hash table. If there are matching build records, the algorithm computes the join results, and sends them to upper level query operators.

The cost of this algorithm can be analyzed as in Table 4 with the terms defined in Table 3. Here, we assume that the hash table does not fit into CPU cache, which is usually the case. We do not include PCM costs for the join results as they are often consumed in the CPU cache by higher-level operators in the query plan tree.

The cache misses of the build phase are caused by reading all the join keys ($\min(\frac{M_R L_R}{L}, M_R)$) and accessing the hash table ($M_R N_{hR}$). When the record size is small, the first term is similar to reading the entire build relation. When the record size is large, it incurs roughly one cache miss per record. Note that because multiple entries may share a single hash bucket, the lines written back can be a subset of the lines accessed for a hash table visit. For the probe phase, the cache misses are caused by scanning the probe relation ($\frac{M_S L_S}{L}$), accessing the hash table ($M_S N_{hS}$), and accessing matching build records in a random fashion. The latter can be computed as shown in Figure 4. The other computations are straightforward.

Algorithm 3 Our proposal: virtual partitioning.[2]

```
Partition phase:
 1: $htsize = M_R*$ hash_table_per_entry_metadata_size;
 2: $P = \lceil ((M_R + M_S)2 + M_R(L_R - 1 + L) + M_S(L_S - 1 + L) + htsize)/C \rceil$;
 3: initiate ID lists $RList[0..P-1]$ and $SList[0..P-1]$;
 4: for ($i = 0$; $i < M_R$; $i$++) do {virtually partition $R$}
 5:    $r$= record $i$ in Relation $R$;
 6:    $p$= hash($r$) modulo $P$;
 7:    append ID $i$ into $RList[p]$;
 8: for ($j = 0$; $j < M_S$; $j$++) do {virtually partition $S$}
 9:    $s$= record $j$ in Relation $S$;
10:    $p$= hash($s$) modulo $P$;
11:    append ID $j$ into $SList[p]$;
Join phase:
 1: for ($p = 0$; $p < P$; $p$++) do {join $R_p$ and $S_p$}
 2:    for each $i$ in $RList[p]$ do
 3:       $r$= record $i$ in Relation $R$;
 4:       insert $r$ into hash table;
 5:    for each $j$ in $SList[p]$ do
 6:       $s$= record $j$ in Relation $S$;
 7:       probe $s$ in the hash table;
 8:       if there are match(es) then
 9:          generate join result(s) from the matching records;
10:          send join result(s) to the upper-level operator;
```

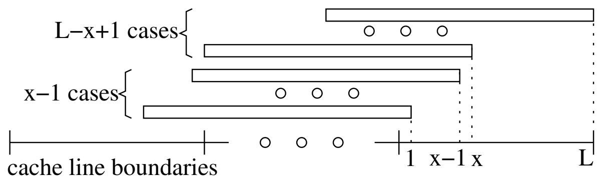


Figure 4: Computing average number of cache misses for unaligned records. A record of $size = yL+x$ bytes, $y \geq 0, L > x \geq 0$, has L possible locations relative to cache line boundaries. Accessing the record incurs on average $\frac{x-1}{L}2 + \frac{L-x+1}{L} + y = \frac{size-1}{L} + 1$ cache misses.

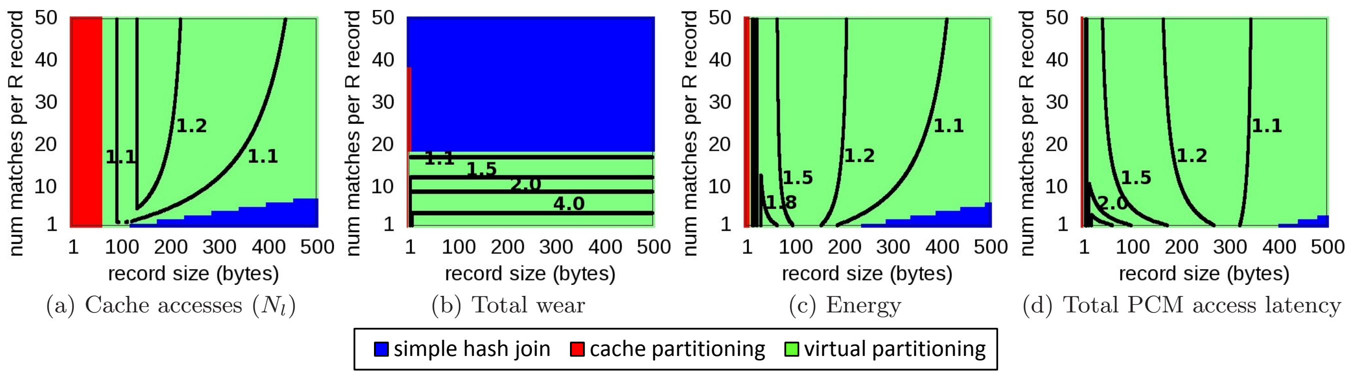
Cache Partitioning. When both input relation sizes are fixed, if we reduce the record sizes (L_R, L_S), then the numbers of records (M_R, M_S) increase. Therefore, simple hash join incurs a large number of cache misses when record sizes are small. The cache partitioning algorithm solves this problem. As shown in Algorithm 2, in the partition phase, the two input relations (R and S) are hash partitioned so that every pair of partitions (R_p and S_p) can fit into the CPU cache. Then in the join phase, every pair of R_p and S_p are joined using the simple hash join algorithm.

The cost analysis of cache partitioning is straightforward as shown in Table 4. Note that we assume that modified cache lines during the partition phase are not prematurely evicted because of cache conflicts. Observe that the number of cache misses using cache partitioning is constant if the relation sizes are fixed. This addresses the above problem of simple hash join.

[2]For simplicity, Algorithm 2 and Algorithm 3 assume perfect partitioning when generating cache-sized partitions. To cope with data skews, one can increase the number of partitions P so that even the largest partition can fit into the CPU cache. Note that using a larger P does not change the algorithm analysis.

Table 4: Cost analysis for three hash join algorithms.

Algorithm		Cache Line Accesses from PCM (N_l)	Cache Line Write Backs (N_{lw})	Words Written (N_w)
Simple Hash	Build	$\min(\frac{M_R L_R}{L}, M_R) + M_R N_{hR}$	$M_R HashTable_{lw}$	$M_R HashTable_w$
	Probe	$\frac{M_S L_S}{L} + M_S N_{hS} + M_S MatchPerS(\frac{L_R-1}{L} + 1)$	0	0
Cache Partition	Partition	$2(\frac{M_R L_R}{L} + \frac{M_S L_S}{L})$	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L}$	$\frac{M_R L_R}{W} + \frac{M_S L_S}{W}$
	Join	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L}$	0	0
Virtual Partition	Partition	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L} + (M_R + M_S)\frac{2}{L}$	$(M_R + M_S)\frac{2}{L}$	$(M_R + M_S)\frac{2}{W}$
	Join	$(M_R + M_S)\frac{2}{L} + M_R(\frac{L_R-1}{L} + 1) + M_S(\frac{L_S-1}{L} + 1)$	0	0



(a) Cache accesses (N_l) (b) Total wear (c) Energy (d) Total PCM access latency

Figure 5: Comparing three hash join algorithms analytically. ($L_S = L_R$, $MatchPerS = 1$, $\gamma = 0.5$; **the hash table in simple hash join does not fit into cache; hash table access parameters are based on experimental measurements:** $N_{hR} \simeq N_{hS} = 1.8$, $HashTable_{lw} = 1.5$, $HashTable_w = 5.0$.) **For configurations where virtual partitioning is the best scheme, contour lines show the relative benefits of virtual partitioning compared to the second best scheme.**

However, cache partitioning introduces a large number of writes compared to simple hash join: it is writing the amount of data equivalent to the size of the entire input relations. As writes are bad for PCM, we would like to design an algorithm that reduces the writes while still achieving similar benefits of cache partitioning. We propose the following variant of cache partitioning.

New: Virtual Partitioning. Instead of physically copying input records into partitions, we perform the partitioning *virtually*. As shown in Algorithm 3, in the partition phase, for every partition, we compute and remember the record IDs that belong to the partition for both R and S.[3] Then in the join phase, we can use the record ID lists to join the records of a pair of partitions in place, thus avoiding the large number of writes in cache partitioning.

We optimize the record ID list implementation by storing the deltas of two subsequent record IDs to further reduce the writes. As the number of cache partitions is often smaller than a thousand, we find using two-byte integers can encode most deltas. For rare cases with larger deltas, we reserve $0xFFFF$ to indicate that a full record ID is recorded next.

The costs of the virtual partitioning algorithm is analyzed in Table 4. The costs for the partition phase include scanning the two relations as well as generating the record ID lists. The latter writes two bytes per record. In the join phase, the records are accessed in place. They are essentially scattered in the two input relations. Therefore, we use the formula for unaligned records in Figure 4 to compute the number of cache misses for accessing the build and probe records. Note that the computation of the number of partitions P in Algorithm 3 guarantees that the total cache lines accessed per pair of R_p and S_p fit into the CPU cache capacity C.

Comparisons of the Three Algorithms. Figure 5 compares the three algorithms analytically using the formulas in Table 4. We assume R and S have the same record size, and it is a primary-foreign key join (thus $MatchPerS = 1$). From left to right, Figures 5(a) to (d) show the comparison results for four metrics: (a) cache accesses (N_l), (b) total wear, (c) energy, and (d) total PCM access latency. In each figure, we vary the record size from 1 to 500 bytes, and the number of matches per R record ($MatchPerR$) from 1 to 50. Every point represents a configuration for the hash join. The color of a point shows the best scheme for the corresponding configuration: blue for simple hash join, red for cache partitioning, and green for virtual partitioning. For configurations where virtual partitioning is the best scheme, the contour lines show the relative benefits of virtual partitioning compared to the second best scheme.

Figure 5(a) focuses on CPU cache performance, which is the main consideration for previous cache-friendly hash join designs. We see that as expected, simple hash join is the best scheme when record size is very large and cache partitioning is the best scheme when record size is small. Compared to simple hash join, virtual partitioning avoids the many cache misses caused by hash table accesses. Compared to

[3]We assume that there is a simple mapping between a record ID and the record location in memory. For example, if fixed length records are stored consecutively in an array, then the array index can be used as the record ID. If records always start at 8B boundaries, then the record ID of a record can be the record starting address divided by 8.

cache partitioning, virtual partitioning reduces the number of cache misses during the partition phase, while paying extra cache misses for accessing scattered records in the join phase. As a result, virtual partitioning achieves the smallest number of cache misses for a large number of configurations in the middle between the red and blue points.

Figures 5(b) to (d) show the comparison results for the three PCM metrics. First of all, we see that the figures are significantly different from Figure 5(a). This means that introducing PCM main memory can significantly impact the relative benefits of the algorithms. Second, very few configurations benefit from cache partitioning because it incurs a large number of PCM writes in the partition phase, adversely impacting its PCM performance. Third, in Figure 5(b), virtual partitioning achieves the smallest number of writes when $MatchPerR \leq 18$. Virtual partitioning avoids many of the expensive PCM writes in the partition phase of the cache partitioning algorithm. Interestingly, simple hash join achieves the smallest number of writes when $MatchPerR \geq 19$. This is because as $MatchPerR$ increases, the number of S records (M_S) increases proportionally, leading to a larger number of PCM writes for virtual partitioning, while the number of PCM writes in simple hash join is not affected. The cross-over point is 19 here. Finally, virtual partitioning presents a good balance between cache line accesses and PCM writes, and it excels in energy and total PCM access latency in most cases.

4. EXPERIMENTAL EVALUATION

We evaluate our proposed B$^+$-tree and hash join algorithms through cycle-accurate simulations in this section. We start by describing the simulator used in the experiments. Then we present the experimental results for B$^+$-trees and hash joins. Finally, we perform sensitivity analysis for PCM parameters.

4.1 Simulation Platform

We extended a cycle-accurate out-of-order X86-64 simulator, PTLsim [20], with PCM support. PTLsim is used extensively in computer architecture studies and is currently the only publicly available cycle-accurate simulator for out-of-order x86 micro-architectures. The simulator models the details of a superscalar out-of-order processor, including instruction decoding, micro-code, branch prediction, function units, speculation, and a three-level cache hierarchy. PTLsim has multiple use modes; we use PTLsim to simulate single 64-bit user-space applications in our experiments.

We extended PTLsim in the following ways to model PCM. First, we model data comparison writes for PCM writes. When writing a cache line to PCM, we compare the new line with the original line to compute the number of modified bits and the number of modified words. The former is used to compute PCM energy consumption, while the latter impacts PCM write latency. Second, we model four parallel PCM memory ranks. Accesses to different ranks can be carried out in parallel. Third, we model the details of cache line write back operations carefully. Previously, PTLsim assumes that cache line write backs can be hidden completely, and does not model the details of this operation. Because PCM write latency is significantly longer than its read latency, cache line write backs may actually keep the PCM busy for a sufficiently long time to stall front-end cache line fetches. Therefore, we implemented a 32-entry FIFO write

Table 5: Simulation Setup.

Simulator	PTLsim enhanced with PCM support
Processor	Out-of-order X86-64 core, 3GHz
CPU cache	Private L1D (32KB, 8-way, 4-cycle latency), private L2 (256KB, 8-way, 11-cycle latency), shared L3 (8MB, 16-way, 39-cycle latency), all caches with 64B lines, 64-entry DTLB, 32-entry write back queue
PCM	4 ranks, read latency for a cache line: 230 cycles, write latency per 8B modified word: 450 cycles, $E_{rb} = 2$ pJ, $E_{wb} = 16$ pJ

queue in the on-chip memory controller, which keeps track of dirty cache line evictions and performs the PCM writes asynchronously in the background.

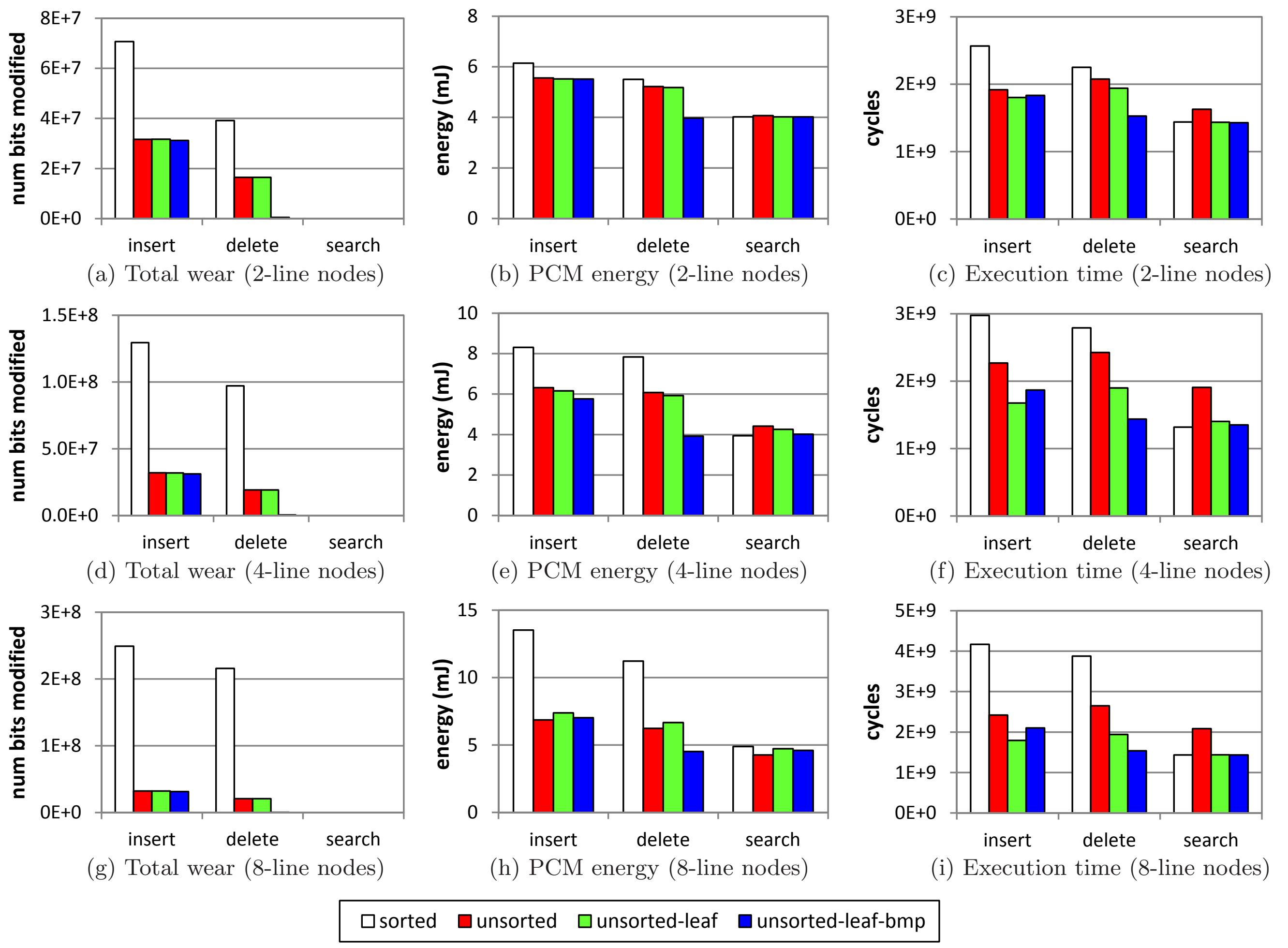
Table 5 describes the simulation parameters. The cache hierarchy is modeled after the recent Intel Nehalem processors [14]. The PCM latency and energy parameters are based on a previous computer architecture study [15]. We adjusted the latency in cycles according to the 3GHz processor frequency and the DDR3 bus latency. The word size of 8 bytes per iteration of write operations is based on [8].

4.2 B$^+$-Tree Index

We implemented four variants of B$^+$-trees as described in Section 3.2: sorted, unsorted, unsorted leaf, and unsorted leaf with bitmap. Figure 6 compares the four schemes for common index operations. In every experiment, we populate the trees with 50 million entries. An entry consists of an 8-byte integer key and an 8-byte pointer. We populate the nodes 75% full initially. We randomly shuffle the entries in all unsorted nodes so that the nodes represent the stable situations after updates. Note that the total tree size is over 1GB, much larger than the largest CPU cache (8MB). For the insertion experiments, we insert 500 thousand random new entries into the trees back to back, and report total wear in number of PCM bits modified, PCM energy consumption in millijoules, and execution time in cycles for the entire operation. Similarly, we measure the performance of 500 thousand back-to-back deletions for the deletion experiments, and 500 thousand back-to-back searches for the search experiments. We vary the node size of the trees. As suggested by previous studies, the best tree node sizes are a few cache lines large [5, 12]. Since a one-line (64B) node can contain only 3 entries, which makes the tree very deep, we show results for node sizes of 2, 4, and 8 cache lines.

The sub-figures in Figure 6 are arranged as a 3x3 matrix. Every row corresponds to a node size. Every column corresponds to a performance metric. In every sub-figure, there are three groups of bars, corresponding to the insertion, deletion, and search experiments. The bars in each group show the performance of the four schemes. (Note that search does not incur any wear.) We observe the following points in Figure 6.

First, compared to the conventional sorted trees, all the three unsorted schemes achieve better total wear, energy consumption, and execution time for insertions and deletions, the two index operations that incur PCM writes. The sorted trees pay the cost of moving the sorted array of entries in a node to accommodate insertions and deletions. In contrast, the unsorted schemes all save PCM writes by allowing entries to be unsorted upon insertions and deletions. This saving increases as the node size increases. Therefore,



(a) Total wear (2-line nodes) (b) PCM energy (2-line nodes) (c) Execution time (2-line nodes)

(d) Total wear (4-line nodes) (e) PCM energy (4-line nodes) (f) Execution time (4-line nodes)

(g) Total wear (8-line nodes) (h) PCM energy (8-line nodes) (i) Execution time (8-line nodes)

Figure 6: B$^+$-tree performance. (50 million entries in trees; 75% full; "insert": inserting 500 thousand random keys; "delete": randomly deleting 500 thousand existing keys; "search": searching for 500 thousand random keys)

the performance gaps widen as the node size grows from 2 cache lines to 8 cache lines.

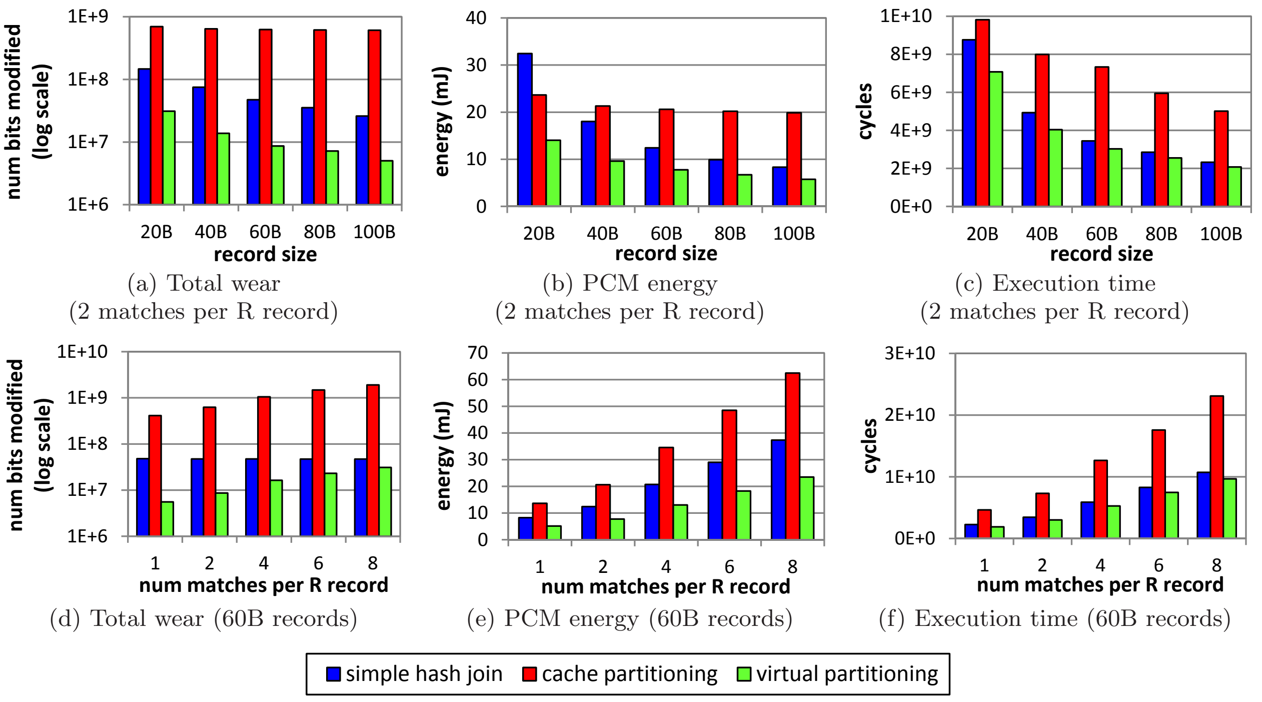
Second, compared to the conventional sorted trees, the scheme with all nodes unsorted suffers from slower search time by a factor of 1.13–1.46X because the hot, top tree nodes stay in CPU cache, and a search incurs a lot of instruction overhead in the unsorted non-leaf nodes. In contrast, the two schemes with only unsorted leaf nodes achieve similar search time as the sorted scheme.

Third, comparing the two unsorted leaf schemes, we see that unsorted leaf with bitmap achieves better total wear, energy, and time for deletions. This is because unsorted leaf with bitmap often only needs to mark one bit in a leaf bitmap for a deletion (and the total wear is about $5E5$ bits modified), while unsorted leaf has to overwrite the deleted entry with the last entry in a leaf node and update the counter in the node. On the other hand, the unsorted leaf with bitmap suffers from slightly higher insertion time because of the instruction overhead of handling the bitmap and the holes in a leaf node.

Overall, we find that the two unsorted leaf schemes achieve the best performance. Compared to the conventional sorted B$^+$-tree, the unsorted leaf schemes improve total wear by a factor of 7.7–436X, energy consumption by a factor of 1.7–2.5X, and execution time by a factor of 2.0–2.5X for insertions and deletions, while achieving similar search performance. If the index workload consists of mainly insertions and searches (with the tree size growing), we recommend the normal unsorted leaf. If the index workload contains a lot of insertions and a lot of deletions (e.g., the tree size stays roughly the same), we recommend the unsorted leaf scheme with bitmap.

4.3 Hash Joins

We implemented the three hash join algorithms as discussed in Section 3.3: simple hash join, cache partitioning, and virtual partitioning. We model in-memory join operations, where the input relations R and S are in main memory. The algorithms build in-memory hash tables on the R relation. To hash a R record, we compute an integer hash code from its join key field, and modulo this hash code by the size of the hash table to obtain the hash slot. Then we insert (hash code, pointer to the R record) into the hash slot. Conflicts are resolved through chained hashing. To probe an S record, we compute the hash code from its join key field, and use the hash code to look up the hash ta-



(a) Total wear (2 matches per R record)

(b) PCM energy (2 matches per R record)

(c) Execution time (2 matches per R record)

(d) Total wear (60B records)

(e) PCM energy (60B records)

(f) Execution time (60B records)

Figure 7: Hash join performance. (50MB R table joins S table, varying the record size from 20B to 100B and varying the number of matches per R record from 1 to 8.)

ble. When there is an entry with the matching hash code, we check the associated R record to make sure that the join keys actually match. The join results are sent to a high-level operator that consumes the results. In our implementation, the high-level operator simply increments a counter.

Figure 7 compares the three hash join algorithms. The R relation is 50MB large. Both relations have the same record size. We vary the record size from 20B to 100B in Figures 7(a)–(c). We vary the number of matches per R record ($MatchPerR$) from 1 to 8 in Figures 7(d)–(f); in other words, the size of S varies from 50MB to 400MB. We report total wear, energy consumption, and execution times for every set of experiments.

The results in Figure 7 confirm our analytical comparison in Section 3.3. First, cache partitioning performs poorly in almost all cases because it performs a large number of PCM writes in its partition phase. This results in much higher total wear, higher energy consumption, and longer execution time compared to the other two schemes.

Second, compared to simple hash join, when varying record size from 20B to 100B, virtual partitioning improves total wear by a factor of 4.7–5.2X, energy consumption by a factor of 2.3–1.4X, and execution time by a factor of 1.24–1.12X. When varying $MatchPerR$ from 1 to 8, virtual partitioning improves total wear by a factor of 8.6–1.5X, energy consumption by a factor of 1.61–1.59X, and execution time by a factor of 1.19–1.11X.

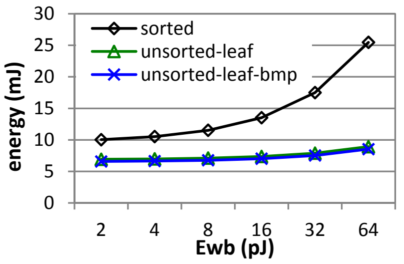
Overall, virtual partitioning achieves the best performance among the three schemes in all the experiments. Compared to cache partitioning, virtual partitioning avoids copying data in the partition phase by remembering record IDs per partition. Compared to simple hash join, virtual partitioning avoids excessive cache misses due to hash table accesses. Therefore, virtual partitioning achieves good behaviors for both PCM writes and cache accesses. Note that the record size and $MatchPerR$ settings in the experiments fall in the region where virtual partitioning wins in Figure 5. Therefore, the experimental results confirm our analytical comparison in Section 3.3.

4.4 PCM Parameter Sensitivity Analysis

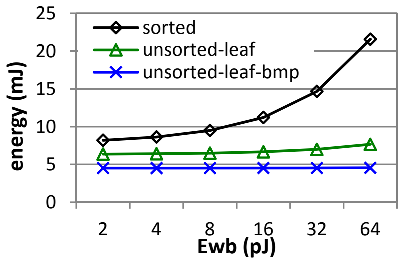
In this section, we vary the energy and latency parameters of PCM in the simulator, and study the impact of the parameter changes on the performance of the B$^+$-tree and hash join algorithms. Note that we still assume data comparison writes for PCM write.

Figure 8 varies the energy consumed by writing a PCM bit (E_{wb}) from 2pJ to 64pJ. The default value of E_{wb} is 16pJ, and 2pJ is the same as the energy consumed by reading a PCM bit. From left to right, Figures 8(a)–(c) show the impact of varying E_{wb} on the energy consumptions of B$^+$-tree insertions, B$^+$-tree deletions, and hash joins. First, we see that as E_{wb} gets smaller, the curves become flat; the energy consumption is more and more dominated by the cache line fetches for reads and for data comparison writes. Second, as E_{wb} gets larger, the curves increase upwards because the larger E_{wb} contributes significantly to the overall energy consumption. Third, changing E_{wb} does not qualitatively change our previous conclusions. For B$^+$-trees, the two unsorted leaf schemes are still better than sorted B$^+$-trees. Among the three hash join algorithms, virtual partitioning is still the best.

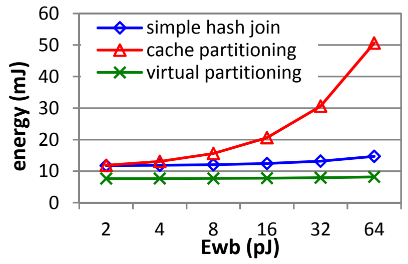
Figure 9 varies the latency of writing a word to PCM (T_w) from 230 cycles to 690 cycles. The default T_w is 450 cycles, and 230 is the same latency as reading a cache line from PCM. From left to right, Figures 8(a)–(c) show the impact of varying T_w on the execution times of B$^+$-tree insertions, B$^+$-tree deletions, and hash joins. We see that as T_w increases, the performance gaps among different schemes become larger. (The performance gap between simple hash



(a) B$^+$-tree insertions

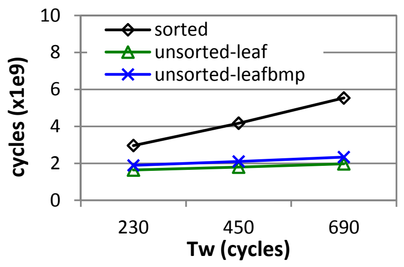


(b) B$^+$-tree deletions

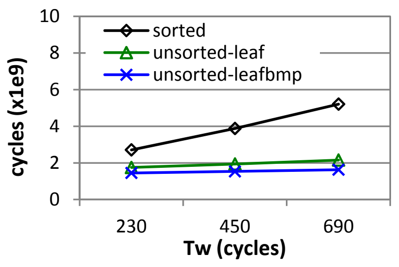


(c) Hash joins (2 matches per R record, 60B)

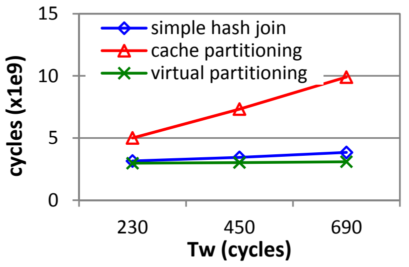
Figure 8: Sensitivity analysis: varying energy consumed for writing a PCM bit (E_{wb}).



(a) B$^+$-tree insertions



(b) B$^+$-tree deletions



(c) Hash joins (2 matches per R record, 60B)

Figure 9: Sensitivity analysis: varying latency of writing a word to PCM (T_w).

join and virtual partitioning is 6% when T_w is 230 cycles.) We find that previous conclusions still hold for B$^+$-trees and hash joins.

5. RELATED WORK

PCM Architecture. As discussed in previous sections, several recent studies from the computer architecture community have proposed solutions to make PCM a replacement for or an addition to DRAM main memory. These studies address various issues including improving endurance [15, 21, 22, 32], improving write latency by reducing the number of PCM bits written [8, 15, 31, 32], preventing malicious wear-outs [26], and supporting error corrections [25]. However, these studies focus on hardware design issues that are orthogonal to our focus on designing efficient algorithms for software running on PCM.

PCM-Based File Systems. BPFS [9], a file system designed for byte-addressable persistent memory, exploits both the byte-addressability and non-volatility of PCM. In addition to being significantly faster than disk-based file systems (even when they are run on DRAM), BPFS provides strong safety and consistency guarantees by using a new technique called *short-circuit shadow paging*. Unlike traditional shadow paging file systems, BPFS uses copy-on-write at fine granularity to atomically commit small changes at any level of the file system tree. This avoids updates to the file system triggering a cascade of copy-on-write operations from the modified location up to the root of the file system tree. BPFS is a file system, and hence it does not consider the database algorithms we consider. Moreover, BPFS has been designed for the general class of byte-addressable persistent memory, and it does not consider PCM-specific issues such as read-write asymmetry or limited endurance.

Battery-Backed DRAM. Battery-backed DRAM (BBDRAM) has been studied as a byte-addressable, persistent memory. The Rio file cache [16] uses BBDRAM as the buffer cache, eliminating any need to flush dirty data to disk. The Rio cache has also been integrated into databases as a persistent database buffer cache [18]. The Conquest file system [29] uses BBDRAM to store small files and metadata. eNVy [30] placed flash memory on the memory bus by using a special controller equipped with a BBDRAM buffer to hide the block-addressable nature of flash. WAFL [13] keeps file system changes in a log in BBDRAM and only occasionally flushes them to disk. While BBDRAM may be an alternative to PCM, PCM has two main advantages over BBDRAM. First, BBDRAM is vulnerable to correlated failures; for example, the UPS battery will often fail either before or along with primary power, leaving no time to copy data out of DRAM. Second, PCM is expected to scale much better that DRAM, making it a better long-term option for persistent storage [3]. On the other hand, using PCM requires dealing with expensive writes and limited endurance, a challenge not present with BBDRAM. Therefore, BBDRAM-based algorithms do not require addressing the challenges studied in this paper.

Main Memory Database Systems and Cache-Friendly Algorithms. Main memory database systems [11] maintain necessary data structures in DRAM and hence can exploit DRAM's byte-addressable property. As discussed in Section 3.1, the traditional design goals of main memory algorithms are low computation complexity and good CPU cache performance. Like BBDRAM-based systems, main memory database systems do not need to address PCM-specific challenges. In this paper, we found that for PCM-friendly algorithms, one important design goal is to minimize PCM writes. Compared to previous cache-friendly B$^+$-trees and hash joins, our new algorithms achieve significantly better performance in terms of PCM total wear, energy consumption, and execution time.

6. CONCLUSION

A promising non-volatile memory technology, PCM is expected to play an important role in the memory hierarchy in the near future. This paper focuses on exploiting PCM as main memory for database systems. Based on the unique characteristics of PCM (as opposed to DRAM and NAND flash), we identified the importance of reducing PCM writes for optimizing PCM endurance, energy, and performance. Specifically, we applied this observation to database algorithm design, and proposed new B$^+$-tree and hash join designs that significantly improve the state-of-the-art.

As future work in the PCM-DB project, we are interested in optimizing PCM writes for different aspects of database system designs, including important data structures, query processing algorithms, and transaction logging and recovery. The latter is important for achieving transaction atomicity and durability. BPFS proposed a different solution based on shadow copying and atomic writes [9]. It is interesting to compare this proposal with conventional database transaction logging, given the goal of reducing PCM writes.

Moreover, another interesting aspect to study is the fine-grain non-volatility of PCM. Challenges may arise in hierarchies where DRAM is explicitly controlled by software. Because DRAM contents are lost upon restart, the relationship between DRAM and PCM must be managed carefully; for example, pointers to DRAM objects should not be stored in PCM. On the other hand, the fine-grain non-volatility may enable new features, such as "instant-reboot" that resumes the execution states of long-running queries upon crash recovery so that useful work is not lost.

7. REFERENCES

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[2] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[3] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. Phase change memory technology. *J. Vacuum Science*, 28(2), 2010.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, 2004.

[5] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, 2001.

[6] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B$^+$-trees: Optimizing both cache and disk performance. In *SIGMOD*, 2002.

[7] S. Cho. Personal communication, 2010.

[8] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO*, 2009.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[10] E. Doller. Phase change memory and its impacts on memory hierarchy. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[11] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE TKDE*, 4(6), 1992.

[12] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B$^+$-trees. In *SIGMETRICS*, 2003.

[13] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

[14] Intel Corp. First the tick, now the tock: Intel micro-architecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[16] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. *Operating Systems Review*, 31, 1997.

[17] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *The VLDB Journal*, 19(1), 2010.

[18] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. *The VLDB Journal*, 7(3), 1998.

[19] PCM-DB. http://www.pittsburgh.intel-research.net/projects/hi-spade/pcm-db/.

[20] PTLsim. http://www.ptlsim.org/.

[21] M. K. Qureshi, J. P. 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*, 2009.

[22] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[23] J. Rao and K. A. Ross. Making B$^+$-trees cache conscious in main memory. In *SIGMOD*, 2000.

[24] Samsung. Samsung ships industry's first multi-chip package with a PRAM chip for handsets. http://www.samsung.com/us/business/semiconductor/newsView.do?news_id=1149, April 2010.

[25] S. E. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ECP, not ECC, for hard failures in resistive memories. In *ISCA*, 2010.

[26] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *ISCA*, 2010.

[27] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.

[28] H.-W. Tseng, H.-L. Li, and C.-L. Yang. An energy-efficient virtual memory system with flash memory as the secondary storage. In *Int'l Symp. on Low Power Electronics and Design (ISPLED)*, 2006.

[29] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *USENIX Annual Technical Conference*, 2002.

[30] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS*, 1994.

[31] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *IEEE ISCAS*, 2007.

[32] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.