Benzene: An Energy-Efficient Distributed Hybrid Cache Architecture for Manycore Systems

NAMHYUNG KIM, JUNWHAN AHN, and KIYOUNG CHOI, Seoul National University
DANIEL SANCHEZ, MIT
DONGHOON YOO and SOOJUNG RYU, Samsung Electronics

10

This article proposes Benzene, an energy-efficient distributed SRAM/STT-RAM hybrid cache for manycore systems running multiple applications. It is based on the observation that a naïve application of hybrid cache techniques to distributed caches in a manycore architecture suffers from limited energy reduction due to uneven utilization of scarce SRAM. We propose two-level optimization techniques: intra-bank and inter-bank. Intra-bank optimization leverages highly associative cache design, achieving more uniform distribution of writes within a bank. Inter-bank optimization evenly balances the amount of write-intensive data across the banks. Our evaluation results show that Benzene significantly reduces energy consumption of distributed hybrid caches.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Hardware** → **On-chip resource management**; **Chip-level power issues**; **Memory and dense storage**; **Spintronics and magnetic technologies**;

Additional Key Words and Phrases: Manycore systems, energy-efficient, distributed, hybrid cache, STT-RAM

ACM Reference format:

Namhyung Kim, Junwhan Ahn, Kiyoungh Choi, Daniel Sanchez, Donghoon Yoo, and Soojung Ryu. 2018. Benzene: An Energy-Efficient Distributed Hybrid Cache Architecture for Manycore Systems. *ACM Trans. Archit. Code Optim.* 15, 1, Article 10 (March 2018), 23 pages.
<https://doi.org/10.1145/3177963>

1 INTRODUCTION

Over the last decade, processor speed improvement has been achieved by increasing the number of cores rather than frequency scaling due to the limited amount of instruction-level parallelism and power inefficiency in improving performance with scaling the clock speed. Accordingly, processor architectures with tens of or more cores, called manycore architectures, are proposed and have become popular in both academia and industry (e.g., Tiler TILE64 [6] and Intel Xeon Phi [13]). As these architectures are designed to run large numbers of applications simultaneously to fully utilize all cores in the system, they often incorporate tens of megabytes of last-level caches (LLCs) to efficiently handle frequent memory accesses from the applications.

This work is supported by Samsung Electronics Co., Ltd.

Authors' addresses: N. Kim, J. Ahn, and K. Choi, Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, Korea; emails: nhkim@dal.snu.ac.kr, {junwhan, kchoi}@snu.ac.kr; D. Sanchez, MIT CSAIL, 32 Vassar St, 32-G838, Cambridge, MA 02139; email: sanchez@csail.mit.edu; D. Yoo and S. Ryu, Samsung Electronics, Hwaseong 18448, Korea; emails: {say.yoo, soojung.ryu}@samsung.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 ACM 1544-3566/2018/03-ART10 \$15.00

<https://doi.org/10.1145/3177963>

However, there are two critical issues in using large shared on-chip LLC with manycore systems—energy efficiency and quality of service (QoS) guarantee. First, SRAM, which is the most popular memory technology for constructing on-chip caches, is not suitable for large LLC due to its high static energy and low density. This can be alleviated by employing a new memory technology, such as STT-RAM, that provides much lower static energy and higher density, thereby being much more efficient in constructing large on-chip caches. Despite these advantages over SRAM, STT-RAM suffers from poor write characteristics in terms of energy and latency. To mitigate the problem, recent work proposed hybrid caches [2, 41], which combines small SRAM with large STT-RAM to service write-intensive data with the SRAM. Second, as LLC in manycore systems is shared across more and more applications running at the same time, interference among applications (e.g., thrashing) significantly degrades cache efficiency, resulting in failure to guarantee QoS. To mitigate such an interference problem in manycore systems, distributed cache partitioning [4, 20] was proposed to isolate the impact of each application on the LLC.

While hybrid cache architectures and cache partitioning techniques for manycore systems are widely explored, none of the prior work considers distributed hybrid caches with manycore cache partitioning. More importantly, we observe that simply combining existing techniques together is not efficient enough, because the two problems interact with each other. For example, the block allocation policy in hybrid caches determines cache blocks to be allocated to SRAM instead of STT-RAM, but its effectiveness highly depends also on how applications are mapped to each cache bank (determined by distributed cache partitioning).

In this work, we propose *Benzene*, a distributed hybrid cache architecture for manycore systems. Each distributed LLC slice is composed of an SRAM/STT-RAM hybrid cache and stores write-intensive data in SRAM to reduce costly STT-RAM writes. Under this organization, however, we observe significant write imbalance within a bank and across the banks. This write imbalance harms SRAM utilization, which is the key factor of energy efficiency in hybrid caches. To alleviate the problem, *Benzene* performs two key optimizations: (1) intra-bank optimization, which evenly distributes writes across different sets within each bank, and (2) inter-bank optimization, which balances the amount of write-intensive data in each bank by leveraging cache block placement in distributed caches. With these two optimizations, *Benzene* utilizes SRAM more efficiently and reduces cache energy by 47.1% on average with minimal performance overhead.

In summary, this article makes the following contributions:

- For the first time, we explore distributed hybrid cache design for manycore systems with cache partitioning and observe the write imbalance problem in two different levels, which incurs inefficiency in hybrid cache management.
- To address the write imbalance problem, we propose *Benzene*, a distributed hybrid cache architecture for manycore systems. It is composed of two levels of optimizations for distributed hybrid caches that aim at evenly balancing the amount of writes to each bank and each cache set.
- We evaluate *Benzene* based on an architectural simulator and show that it achieves a 47.1% LLC energy reduction compared to the state-of-the-art distributed cache architecture.

2 BACKGROUND AND MOTIVATION

2.1 STT-RAM

Spin-Transfer Torque RAM (STT-RAM) is an emerging memory technology that is considered a promising alternative to SRAM. Contrary to conventional charge-based memory technologies (e.g., SRAM and DRAM) where power supply is necessary to keep the data alive, STT-RAM stores its information in the form of magnetization direction, and thus, the data stored in STT-RAM persist

even without power supply (i.e., non-volatility). Due to this, STT-RAM consumes very low static energy compared to conventional charge-based memory technologies. This makes STT-RAM more suitable for large LLCs commonly employed by manycore systems where a large number of cores demand high memory bandwidth.

The drawback of STT-RAM caches is that they consume much higher write energy with longer write latency than its SRAM counterpart. According to our experiments and other literature, simply replacing SRAM to STT-RAM is not beneficial in terms of energy efficiency, because its high write energy offsets the benefit of its low static energy. To alleviate the impact of write inefficiency of STT-RAM, various device-/circuit-/architecture-level solutions [1, 7–9, 14, 17, 21–23, 25, 35, 37, 38, 41, 42, 44, 45] have been proposed. In this article, we focus on one of the most popular techniques called *hybrid caches*, which will be explained in the next subsection.

2.2 Hybrid Caches

The key idea of hybrid caches [2, 8, 21–23, 25, 37, 41, 42] is to combine small SRAM and large STT-RAM and allocate write-intensive data to the SRAM. It allows us to take advantage of low static energy of STT-RAM and low write energy of SRAM at the same time. In this style of architecture, energy efficiency of caches is determined largely by how efficiently we can utilize the small SRAM for absorbing writes that would otherwise be made to STT-RAM. For this, there have been several data placement policies for hybrid caches to determine which cache blocks to allocate into SRAM. Earlier work [21, 25, 37, 41, 42] allocates cache blocks loaded by a store (or load) miss to the SRAM (or STT-RAM) based on the assumption that blocks loaded by a store miss may receive more write requests than those loaded by a load miss. Since this simple method can result in inefficient block allocation, it usually comes with a migration policy [8, 22, 25, 41], which moves blocks in SRAM to STT-RAM and vice versa based on the number of reads/writes received by each block. In this article, we adopt a more advanced hybrid cache architecture, called *Prediction Hybrid Cache*.

Prediction Hybrid Cache. Prediction hybrid cache (PHC) [2] is a state-of-the-art hybrid cache architecture that outperforms conventional migration-based hybrid caches. The key idea of PHC is to *predict* the write intensity of each cache block at the time of block load and use that information to guide block placement. For this purpose, it defines *cost* of a block as the analytic model of write intensity:

$$\begin{aligned} \text{cost} &= N_r \Delta E_r + N_w \Delta E_w \\ &= N_r \times (E_r^{STT} - E_r^S) + N_w \times (E_w^{STT} - E_w^S). \end{aligned} \quad (1)$$

In this equation, N_r and N_w are the number of reads and writes¹ while the block resides in the cache, and E_r^S (or E_r^{STT}) and E_w^S (or E_w^{STT}) are the read and write energy of SRAM (or STT-RAM), respectively. Therefore, the cost of a block implies the amount of extra energy consumption when the block is allocated into STT-RAM instead of SRAM. In other words, blocks with higher cost are more preferred to be allocated to SRAM from the energy efficiency perspective.

Based on this cost model, PHC predicts whether the cost of each block will exceed a write intensity threshold κ or not at the time of block insertion. It uses a PC-directed predictor that correlates a write-intensive block (i.e., $\text{cost} \geq \kappa$) and the instruction that triggers loading of the block. This enables highly accurate prediction of write intensity (e.g., 93% in the original article). In addition, since the amount of write-intensive data varies across different applications, PHC also has a mechanism that adjusts the write intensity threshold κ at runtime. It tries to minimize the

¹We slightly modified this cost model to include writes for block fills (on a cache miss, the loaded data have to be inserted into the cache, which incurs a write operation to the cache) into N_w .

write energy consumption while not increasing the cache miss rate compared to the LRU policy (e.g., if too many blocks are allocated to SRAM, the STT-RAM capacity can be underutilized, which degrades the cache hit rate).

2.3 Distributed Cache Partitioning

In manycore systems, the increased number of cores in a chip allows plenty of applications to run simultaneously, thereby improving the system utilization and throughput. However, this creates a new challenge in that those applications interfere with each other for shared resources (e.g., LLCs) and thus results in the degradation of Quality-of-Service (QoS) and hampers the throughput improvement from manycore systems. For this problem happening at the LLC, cache partitioning [4, 10, 20, 29, 32, 43] is one of the most popular solutions. It restricts the cache size that can be occupied by each application (called *partition*) to isolate the impact on LLC performance of each application within each partition. However, most of them are not scalable and only focus on centralized caches, and distributed cache partitioning for manycore systems [4, 20] is also proposed. In this article, we focus on a state-of-the-art cache partitioning for distributed cache architectures, called *Jigsaw*.

Jigsaw [4] partitions distributed caches by introducing a new concept of virtual caches (called *shares*). Each application has its own share, which is constructed with multiple partitions (each of which is located at a single bank) from different banks and is determined by the placement policy. From the local bank perspective, each bank locally adopts a scalable cache partitioning scheme (e.g., Vantage [32]) to manage multiple partitions in a bank from different shares. Globally, the size and placement of each share are periodically updated in a way to minimize the total number of misses based on the monitored statistics. In this way, Jigsaw can address both scalability and interference issues in distributed caches at the same time.

Jigsaw also has a mechanism to enable efficient lookups in distributed cache architectures. It adds a special structure called *share-bank translation buffer (STB)*, which keeps the information about which cache blocks are stored in which banks. Since STB gives the exact location of the given cache block, it does not have to traverse multiple banks to find out where the cache block is stored, unlike other distributed cache architectures.

2.4 Intra-Bank and Inter-Bank Write Imbalance in Distributed Hybrid Caches

The motivation in this work comes from our observation of write imbalance in distributed hybrid caches. As explained in Section 2.2, the effectiveness of a hybrid cache is determined by how efficiently the small SRAM can be utilized. Thus, to maximize the SRAM utilization, all writes have to be equally distributed across the entire cache, which, however, is not the case in conventional hybrid caches combined with distributed cache architectures.

First, conventional set-associative caches, on which existing hybrid cache architectures are based, experience significant write imbalance across sets (i.e., *intra-bank write imbalance*). This is because set indexes are determined only by the lower-order bits of the address, which incurs skewed access distribution particularly for applications with irregular memory access patterns. Considering that existing hybrid caches are built on set-associative caches with only a few SRAM ways in each set, such write imbalance can significantly degrade the SRAM utilization as some sets need to handle much more writes with the same number of SRAM blocks in a set. Figure 1 is an empirical evidence of such set-level write imbalance and low SRAM utilization. It shows the number of SRAM and STT-RAM writes in a prediction hybrid cache having one SRAM way and seven STT-RAM ways, measured by running mix of 64 applications from SPEC CPU2006 (similarly to the configuration in ZCache [31]). As can be seen in the figure, the number of writes

Fig. 1. The distribution of SRAM/STT-RAM writes across different sets in a set-associative cache.

Fig. 2. The distribution of writes across different banks in a distributed cache architecture (i.e., Jigsaw).

varies significantly across different sets, thereby incurring low SRAM utilization in some sets. For example, some sets (SRAM and STT-RAM together) receive up to 8.66 times as many writes compared to those with fewer writes. This leads to over a 22 $\times$ difference in the number of SRAM writes in those sets (e.g., the set with the lowest SRAM utilization handles only 4.3% of writes in SRAM despite the 1:7 ratio of SRAM and STT-RAM ways). The dual-associative hybrid cache (DAHYC) [21] tries to address this inefficiency by allowing neighbor sets to share their SRAM blocks, but such a restricted style of sharing is expected to have a limited impact on mitigating the write imbalance, since sets that receive more writes are often far apart from those with fewer writes.

Second, distributed caches introduce another level of write imbalance, which we call *inter-bank write imbalance*. As a motivational result, Figure 2 shows the distribution of writes across different banks in Jigsaw (see Section 4 for system configuration), which demonstrates severe write imbalance across different banks (up to 9.5 $\times$ difference). Such inter-bank write imbalance is mainly because manycore systems simultaneously run a large number of applications with different memory access patterns and each cache bank is shared by its own combination of application mixes with varying write intensity across different banks. This causes another dimension of inefficiency in distributed hybrid caches, because SRAM in banks with fewer writes will be underutilized compared to those with more frequent writes.

Motivated by these two observations, our architecture aims at *evenly distributing write requests across different sets and banks in a distributed hybrid cache*. Globally balancing the write distribution facilitates higher SRAM utilization in distributed hybrid caches, thereby improving the energy efficiency. As will be explained in the next section, we address both intra-bank and inter-bank write imbalance problems through new architectural techniques.

Fig. 3. Overall architecture.

3 ARCHITECTURE

We consider the tiled architecture used in Jigsaw [4] as a baseline of Benzene. As shown in Figure 3, each tile has a core, its own private cache (L1) and a slice of the shared LLC. The tiles are connected through an on-chip network.

On top of the baseline, Benzene applies PHC to each LLC slice and improves the efficiency of distributed hybrid caches at two different levels—intra-bank and inter-bank. The following gives an overview of our schemes. Note that although we assume Jigsaw and PHC as concrete examples of underlying technologies for our architecture, our techniques can be generalized to other distributed cache partitioning and hybrid cache schemes as well.

Intra-Bank Optimization (Section 3.1). To balance the amount of writes to different cache sets, Benzene applies the concept of highly associative caches [30, 31, 34] to hybrid caches and utilizes the idea of an indirection table [11, 12, 30], called *tag-to-data pointer table*, to minimize the energy overhead of using highly associative caches in hybrid caches. With this design, the scarce SRAM resource in each bank can be shared almost uniformly across all sets in a bank, thereby improving the energy efficiency of hybrid caches at the per-bank level.

Inter-Bank Optimization (Section 3.2). To reduce the write imbalance across all cache banks, Benzene tries to allocate data from write-intensive applications and non-write-intensive applications together in the same cache bank so that write-intensive data can be equally distributed to all banks, thereby enabling more efficient use of SRAM in each bank. To identify the write intensity of each application, a write intensity monitor (shown in Figure 3) collects the write statistics of each application and helps data placement. In addition, we devise an adaptive scheme that disables inter-bank optimization for non-write-intensive workloads to minimize its performance overhead.

Other Optimizations (Section 3.3). We introduce two modifications to the original PHC to be more suitable for distributed cache architectures: (1) set sampling that is aware of cache partitioning and (2) an improved threshold adjustment mechanism for escaping local optimum.

3.1 Intra-Bank Optimization

To mitigate the intra-bank write imbalance (explained in Section 2.4), it is important to overcome the limitation of conventional set-associative caches where each cache block can be placed at one of only a few places within a statically determined set. Thus, we propose to employ highly associative caches, such as skewed-associative caches [34], ZCache [31], and V-Way cache [30], into hybrid cache design. These cache designs determine the physical location of cache blocks based on the hashed value of the block address and provide much higher associativity at low cost, both of which

(a) Conventional ZCache

(b) ZCache with the tag-to-data pointer table

Fig. 4. Intra-bank optimization. W_1 to W_3 are write-intensive blocks, while R_1 indicates a read-intensive block.

help to balance the load to each cache set. Therefore, highly associative caches can improve the balance of write distribution across sets compared to conventional set-associative caches.

However, if we consider ZCache, a state-of-the-art highly-associative cache, simply applying it on top of hybrid caches introduces two important issues that can cause high energy overheads. This is because ZCache uses cuckoo hashing [27] to implement high associativity for replacement candidate selection. Similarly to skewed-associative caches, ZCache employs w hash functions to address w physical ways (i.e., w possible physical locations for each cache block); however, to effectively provide associativity higher than w with w physical ways, ZCache performs a *relocation process* during block insertion, where it (1) chooses one of the w locations for the block to be inserted, (2) displaces the victim cache block, (3) reinserts the displaced one into its other $w - 1$ possible locations, and (4) continues this process until we choose the block to be actually evicted from the cache (i.e., cuckoo hashing). This relocation process is particularly harmful to hybrid caches due to the following two reasons:

- **Relocation overhead:** The relocation process greatly increases the number of write operations to the cache. For example, under our ZCache configuration (see Section 4), every relocation incurs up to one extra read and write to the cache. This can be costly for hybrid caches where an STT-RAM write is an energy-consuming operation.
- **Interaction with hybrid caches:** Relocating cache blocks can disturb the data placement in hybrid caches. As we explained before, hybrid caches try to allocate write-intensive blocks into SRAM. However, the relocation process may move blocks in SRAM into STT-RAM and vice versa, which interferes with the data placement determined by the hybrid caches.

Figure 4(a) illustrates the two aforementioned problems. Let us assume that block W_3 is to be inserted at the slot where W_1 is stored and the cache selects W_2 as the victim for this insertion. ZCache performs cache replacement by replacing W_1 with W_3 , R_1 with W_1 , and W_2 with R_1 to insert W_3 into the cache while evicting W_2 (rather than W_1). Not only does this incur a series of reads and writes to the cache for a single cache replacement, but also it makes W_1 (which is a write-intensive block) relocate from SRAM to STT-RAM after the replacement. Both of these can negatively affect the effectiveness of hybrid caches.

To address these problems, we propose to add a table for indirection from tags to data blocks, called *tag-to-data pointer table*. The tag-to-data pointer table contains the same number of entries as that of the tags (or data blocks) and stores one-to-one mapping information from tags to data blocks. With this table, each cache access is done by (1) searching for a matching tag, (2) *looking up the tag-to-data-pointer table to identify the location of the corresponding data block*, and (3) accessing the data block. Note that step 1 and 2 can be done in parallel, and thus, considering that the tag-to-data pointer table is much smaller than the tag array, it does not add any overhead to cache access latency.

Figure 4(b) shows the same example case with our tag-to-data pointer table. Instead of initiating a chain of replacements at the data array, our mechanism relocates only the tag-to-data pointers. Thus, it can directly replace W_2 with W_3 in the data array without additional data movement at the data array. Without the tag-to-data pointer table, this is not possible, because there is only one SRAM block where W_3 can be inserted under the ZCache replacement policy, i.e., the slot where W_1 is located. Note that, from the replacement policy perspective, the resulting state of the ZCache with the tag-to-data pointer table is still exactly the same as the one without the tag-to-data pointer table.

The purpose of the tag-to-data pointer table is to decouple the physical location of data blocks in the data array from the location of tags. With this mechanism, data blocks can be stored anywhere in the data array without restriction, such as memory addresses, set indexing, or even the output of the hash function in ZCache. In other words, the tag-to-data pointer table *virtualizes* the physical location of data blocks in the data array.

Virtualizing the physical location of data blocks solves the two aforementioned inefficiencies. First, it allows the relocation process to be performed simply by relocating the entries in the tag-to-data pointer table without actually moving around the data blocks. This allows us to implement the relocation process without incurring any extra writes to the data array itself, which is beneficial for hybrid caches. Second, it ensures that the block placement done by hybrid caches is maintained even after the relocation process, because it eliminates the need for actually moving the physical location of data blocks during the relocation process.

3.2 Inter-Bank Optimization

To reduce the inter-bank write imbalance (explained in Section 2.4), we propose inter-bank optimization in our architecture, which tries to evenly balance the number of writes per bank to globally improve the SRAM utilization. The high-level overview of our approach is to (1) profile the write intensity of each application by a new hardware module called *write intensity monitor*, (2) derive the write intensity of each bank based on application-level write intensity, and (3) periodically update the data placement by using our *write-intensity-aware data placement policy*, which aims at balancing the bank-level write intensity across all banks.

Write Intensity Monitor. As mentioned previously, we use application-level write intensity as a metric for evenly distributing the writes across different cache banks. This is because cache partitioning for distributed caches controls the placement of data for each application (e.g., share in

Jigsaw) in distributed banks rather than enforcing the placement of each block. Fortunately, profiling write intensity of each application is straightforward, since hybrid caches already track the information about write intensity of each cache block (e.g., cost model of PHC shown in Section 2.2) to guide block placement between SRAM and STT-RAM. Our idea is to utilize such information to estimate the write intensity of each application. For example, we define the write intensity of an application as a sum of positive costs of cache blocks from the application:

$$AI_A = \frac{\sum \text{cost}_{bl}}{\text{size}_A}, \quad \forall bl \in BL_A, \text{ if } \text{cost}_{bl} > 0, \quad (2)$$

where AI_A denotes write intensity of application A , cost_{bl} denotes cost of block bl from application A , BL_A denotes the set of cache blocks loaded by application A , and size_A denotes the number of cache blocks allocated to application A by the cache partitioning scheme. Conceptually speaking, AI_A represents the average per-block cost of application A .

Based on the definition of AI , we now derive the write intensity of a bank. The key insight behind this is that, for application A that stores part of its data to bank B , the amount of contribution of A to the write intensity of B is proportional to the proportion of A 's data in B . Thus, we define the write intensity of bank B , or BI_B , as follows:

$$BI_B = \sum_A \left(AI_A \times \frac{\text{size}_B^A}{\text{bank\_size}} \right), \quad (3)$$

where size_B^A is the size of the A 's partition in bank B and "bank_size" is the capacity of a bank. Our data placement policy tries to balance this per-bank write intensity metric across all banks in the system.

Write-Intensity-Aware Data Placement. Our data placement scheme is based on existing schemes for data placement in distributed caches with cache partitioning (e.g., Jigsaw). First, our approach obtains the initial placement result from the existing scheme that does not consider write imbalance. Then, we refine this initial placement by iteratively swapping partitions from different banks in a way to improve the balance of per-bank write intensity. Since the write intensity of all banks should have the same value under perfect balance, the target per-bank write intensity is equal to the average write intensity of all banks:

$$\overline{BI} = \frac{\sum_B BI_B}{N_B}, \quad (4)$$

where N_B is the number of banks in the system.

To adjust the data placement so that all banks have write intensity close to $\overline{BI}$, we swap a portion of partitions from the bank that has too-high write intensity to the one that has too-low write intensity. First, we pick two (bank, application) pairs, (B_{high}, A_{high}) and (B_{low}, A_{low}) , such that (1) the application in each pair stores part of its data in the associated bank, (2) the write intensity of B_{high} (or B_{low}) is higher (or lower) than $\overline{BI}$, and (3) the write intensity of A_{high} is higher than that of A_{low} . Then swapping some amount of A_{high} 's data in B_{high} with A_{low} 's data in B_{low} improves the balance of per-bank write intensity distribution, because it exchanges write-intensive data in B_{high} (where write intensity is too high) with non-write-intensive data in B_{low} (where write intensity is too low).

The remaining question is on determining S , the amount of data to be swapped between B_{high} and B_{low} . If S is too small, then it will have a minimal impact on the per-bank write intensity distribution. On the other hand, if S is too large, then the write intensity of B_{high} (or B_{low}) may decrease (or increase) too much beyond $\overline{BI}$. To avoid such situations, we determine S based on

the definition of per-bank write intensity. By definition, the swap operation changes the write intensity of B_{high} as follows:

$$\begin{aligned}\Delta BI_{B_{high}} &= -AI_{A_{high}} \times \frac{S}{\text{bank\_size}} \\ &\quad + AI_{A_{low}} \times \frac{S}{\text{bank\_size}}.\end{aligned}\quad (5)$$

The first term on the right-hand side reflects that S amount of A_{high} 's data is moved out, while the second term adds the contribution of S amount of A_{low} 's data from B_{low} . Similarly, we can calculate the difference in the write intensity of B_{low} as follows:

$$\begin{aligned}\Delta BI_{B_{low}} &= -AI_{A_{low}} \times \frac{S}{\text{bank\_size}} \\ &\quad + AI_{A_{high}} \times \frac{S}{\text{bank\_size}}.\end{aligned}\quad (6)$$

Since our goal is to adjust the write intensity of both banks to be closer to $\overline{BI}$, we need to satisfy one of the following two equations:

$$BI_{B_{high}} + \Delta BI_{B_{high}} = \overline{BI}, \quad (7)$$

$$BI_{B_{low}} + \Delta BI_{B_{low}} = \overline{BI}. \quad (8)$$

Solving each equation for S results in the following two possible values for S :

$$S_{high} = \frac{\overline{BI} - BI_{B_{high}}}{-AI_{A_{high}} + AI_{A_{low}}} \times \text{bank\_size}, \quad (9)$$

$$S_{low} = \frac{\overline{BI} - BI_{B_{low}}}{-AI_{A_{low}} + AI_{A_{high}}} \times \text{bank\_size}. \quad (10)$$

Between the two, we choose the smaller one as S (i.e., $S = \min(S_{high}, S_{low})$) for more gradual refinement.² Conceptually speaking, this algorithm chooses the amount of exchange to be *proportional to the difference between the write intensity of B_{high} or B_{low} and $\overline{BI}$* .

Minimizing the Performance Overhead of Data Placement Adjustment. There are two sources of performance overhead in our data placement adjustment. First, moving cache blocks from one bank to another may cause a significant overhead even under a long reconfiguration period. Thus we do not perform proactive movement but adopt an on-demand migration technique in CDCS [5]. It adds an additional structure called shadow descriptor, which stores placement information from the previous reconfiguration period. The mechanism directs cache accesses to the old bank if the target cache block is not yet moved to the new bank and triggers migration of the block *after the access*.

Second, if our placement algorithm moves cache blocks from a particular application farther from the core that runs the application, then it may increase the on-chip network latency in accessing the cache blocks. Our approach mitigates such an overhead by rejecting swap operations that increase the hop-count between the core that run the application and cache blocks from that

²If S is larger than the size of the partition allocated to A_{high} or A_{low} (i.e., $\text{size}_{B_{high}}^{A_{high}}$ and $\text{size}_{B_{low}}^{A_{low}}$), then we use the smaller one between the two as the final S .

application by more than two hops (which we call *hop count constraint* λ). This prevents excessive increase in on-chip network latency.

Even with these two mechanisms, there are cases where the increase in on-chip network latency and traffic may offset the benefit of inter-bank optimization. This happens if workloads have low write intensity, because the small amount of cache writes limits the energy reduction that can be achieved by distributing writes across different banks. Thus, Benzene monitors the sum of write intensity of all workloads in the system in every data placement period and *adaptively* disables inter-bank optimization if the write intensity is lower than a threshold ($\sum_A AI_A < \rho$).³ In Section 5.5, we provide a sensitivity analysis of the hop count constraint and impact of the adaptive control of inter-bank optimization.

3.3 Other Optimizations

In our architecture, each bank is locally organized as a PHC, which was originally designed for centralized hybrid caches without cache partitioning. Thus, we propose two modifications to PHC to make it aware of cache partitioning.

Partitioning-Aware Set Sampling. To identify write-intensive data and update the PC-directed predictor, PHC has a hardware structure called *sampler*, which samples a number of sets (i.e., set sampling [18, 19, 28]) and simulates cache replacement behavior of those sets to track the cost of sampled cache blocks. However, the sampler in PHC does not follow the cache partitioning decision that is enforced to the LLC in Benzene. This may cause mismatch between the cache replacement behavior emulated by the sampler and the actual cache replacement, which can degrade the prediction accuracy. Therefore, we organize a *partitioning-aware set sampler* where the sampler follows the same partitioning decision used in the LLC.

Better Threshold Adjustment. PHC periodically adjusts its write intensity threshold κ (see Section 2.2) to adapt to different levels of write intensity at runtime. Since the write intensity of most workloads slowly changes over time, PHC incrementally adjusts the threshold by incrementing/decrementing the threshold by a certain amount at the end of each period. For this purpose, PHC emulates the cache replacement behavior (particularly the number of misses) of the following four settings: the current threshold κ_0 , its neighbor values $\kappa_- (< \kappa)$ and $\kappa_+ (> \kappa)$, and the plain LRU replacement without hybrid caches. Then it tries to use the lowest threshold that has fewer misses than the LRU replacement. If all of them have more misses than the LRU replacement, then it falls back to the threshold with the fewest misses. This is illustrated in the dotted box of Figure 5 (please refer to the original article [2] for more detailed information).

While this was sufficient in the original PHC, we observe that such gradual adjustment may cause PHC to fall into a local optimum under our architecture. This is because Benzene periodically alters its data placement to mitigate inter-bank write imbalance, which often changes the write intensity of each bank too rapidly to be followed by gradual threshold adjustment. Therefore, to escape from a potential local optimum, we add an extra sampler that constantly tracks the merit of using $\kappa = 0$ and use zero if it yields the lowest miss count among all candidate thresholds (i.e., the first branch of Figure 5).⁴ This allows us to quickly escape from the current local optimum and restart the threshold exploration from the initial point.

³We empirically determined the threshold (ρ) and set its value to 727 in our experiment.

⁴Zero is a good starting point for threshold adjustment, because it is the breakeven point of allocating a block into SRAM instead of STT-RAM. In other words, allocating a block into SRAM reduces the energy consumption if and only if $\kappa > 0$.

Fig. 5. Decision tree for threshold adjustment in Benzene. Decision tree in PHC is shown in the dotted box.

Table 1. Configuration of the Simulated System

Component	Configuration
Core	64 Silvermont-like out-of-order cores, two-issue, 32-entry ROB, 2GHz
L1 cache	Private, separate I/D, 32KB each, 8-way set-associative, 64-B blocks, 3-cycle latency
L2 cache	Shared, 512KB per tile (32MB in total), 64-B blocks, PHC [2] with 1 SRAM way and 3 STT-RAM ways, distributed caches based on Jigsaw [4]
On-chip Network	8×8 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 1-cycle links
Main memory	120-cycle zero-load latency, 4 channels, 12.8GB/s per channel

Table 2. Characteristics of an LLC Tile

	SRAM	STT-RAM	Hybrid
Read Latency (cycles)	5	5	5/5*
Write Latency (cycles)	5	22	5/22*
Read Energy (nJ)	0.07	0.10	0.07/0.10*
Write Energy (nJ)	0.07	0.63	0.07/0.63*
Static Power (mW)	7.31	1.18	2.71

*Values for SRAM/STT-RAM, respectively.

4 EVALUATION METHODOLOGY

We use zsim [33] to model a 64-tile manycore system with a distributed shared LLC (shown in Figure 3) and evaluate our architectural techniques based on it. Table 1 summarizes the detailed configuration of our baseline architecture. We use ZCache with 4 ways and 52 candidates. For distributed cache management, we use Jigsaw, which internally uses Vantage [32] as a partitioning scheme within each bank. We empirically determined the period of data placement in Jigsaw to 50 million cycles (i.e., 25ms) and the period of threshold update in PHC to 5 million cycles (i.e., 2.5ms).

We use CACTI [26] and NVSim [15] to model SRAM and STT-RAM, respectively, under 45nm technology with LOP devices. Table 2 shows the characteristics of an SRAM cache tile, an STT-RAM cache tile, and a hybrid cache tile used in our evaluation. Based on these parameters, we calculate ΔE_w and ΔE_r for PHC (see Section 2.2) to 20 and 1, respectively. Our latency and energy

Table 3. Applications from SPEC CPU2006

Write-Intensive Applications				Non-Write-Intensive Applications			
Application	WBPKI	Application	WBPKI	Application	WBPKI	Application	WBPKI
lbm	43.7	GemsFDTD	10.8	bzip2	3.3	calculix	2.0
soplex	22.0	astar	8.2	hmmer	2.9	tonto	1.9
mcf	14.5	h264ref	6.3	gromacs	2.4	namd	1.7
zeusmp	11.6	milc	6.1	povray	2.2	cactusADM	1.5
leslie3d	10.9			gobmk	2.2		

Table 4. Workload Characteristics

Workload	WBPKI	Workload	WBPKI	Workload	WBPKI	Workload	WBPKI	Workload	WBPKI
M1	18.0	M6	12.4	M11	8.7	M16	5.6	M21	3.2
M2	21.1	M7	11.7	M12	8.2	M17	5.4	M22	3.2
M3	23.6	M8	11.6	M13	8.4	M18	6.6	M23	3.3
M4	19.5	M9	13.8	M14	8.4	M19	5.3	M24	3.2
M5	23.7	M10	12.8	M15	8.5	M20	6.0	M25	3.4

models include the overhead from additional hardware structures for Benzene (e.g., tag-to-data pointer table, extra sampler for $\kappa = 0$, etc.).

Our target is to reduce the energy consumption of hybrid caches for applications with write-intensive data (hybrid caches are already good at optimizing the energy consumption of non-write-intensive applications due to the low static power of STT-RAM). Thus, we choose 18 SPEC CPU2006 [16] applications (see Table 3) that have high L2 cache writebacks per kilo-instruction (WBPKI) and classify them into a write-intensive group (top nine applications with high WBPKI) and a non-write-intensive group (the other nine applications). Then, we synthesize twenty-five 64-application mixes⁵ by varying the ratio of write-intensive applications in a mix from 100% (M1) to 0% (M25). Table 4 summarizes write-intensity of each workload. Each application mix is fast-forwarded for 20 billion instructions and then is simulated for 50 billion instructions in total.⁶

Our evaluation compares the following four configurations in terms of energy and performance. Unless otherwise specified, all results are normalized to the STT-RAM baseline.⁷

- **SRAM Baseline** represents a system with an SRAM-based L2 cache. This is essentially the same as Jigsaw.
- **STT-RAM Baseline** is the same as the SRAM baseline except that the L2 cache is constructed with STT-RAM only.
- **Benzene-Intra** is the proposed architecture with intra-bank optimization only.
- **Benzene** includes both intra-bank and inter-bank optimizations. This is our final solution.

⁵In this article, we evaluate our architecture only with multi-programmed workloads. That said, Benzene can run multi-threaded workloads with no modifications to the partitioning scheme, because the shared data placement used in Benzene is based on Jigsaw, which can handle both multi-programmed and multi-threaded workloads. Further improvements can be made by considering different types of shares (per-thread vs. per-process) in inter-bank optimization when running multi-threaded workloads, which is our future work.

⁶This may cause variation on executed instructions of each application, but the difference is less than 2.6%.

⁷The ideal baseline would be naïve combination of Jigsaw and PHC. However, it cannot be implemented, because Jigsaw requires highly associative caches while PHC is based on set-associative caches. The effectiveness of implementing hybrid caches based on highly associative caches will be shown in Section 5.2.

Fig. 6. LLC energy consumption (above) and performance (below) of our architecture compared to the SRAM and STT-RAM baseline. All results are normalized to the STT-RAM baseline.

Fig. 7. LLC energy breakdown. “Relocation” represents the energy overhead of additional read/write operations caused by the cuckoo hashing mechanism of ZCache.

Benzene-Intra and Benzene includes both of other optimization schemes (e.g., partitioning-aware set sampling and better threshold adjustment).

5 EVALUATION RESULTS

5.1 Energy Consumption and Performance

Energy Consumption. Figure 6 shows the energy consumption of the LLC for four different configurations (normalized to the STT-RAM baseline). In addition, we provide more detailed analysis based on the LLC energy breakdown as shown in Figure 7 (averaged over all application mixes). Energy consumption of the predictor is very small (less than 0.1% in both Benzene-Intra and Benzene) and does not show up in the figure. From these results, we make the following observations.

First, the SRAM baseline consumes 29.0% lower LLC energy with negligible performance differences. Although STT-RAM has lower static energy consumption (24.4% on average), this benefit is offset by higher write energy of STT-RAM. Due to this reason, the energy difference between the SRAM baseline and the STT-RAM baseline is larger in write-intensive workloads (towards M1). This implies the need for architectural techniques to reduce such write overheads especially in write-intensive workloads, which agrees with experimental results in many previous studies on STT-RAM caches [1, 2].

Fig. 8. LLC dynamic energy (left) and miss rate (right) of Benzene-Intra and Benzene (normalized to STT-RAM baseline).

Second, Benzene-Intra reduces the LLC energy consumption by 45.3% compared to the STT-RAM baseline. This energy reduction comes from the following two reasons. First, hybrid caches effectively combine the benefit of lower static energy of STT-RAM and lower write energy of SRAM. Second, our tag-to-data pointer table eliminates the overhead of relocation (32.2% of LLC energy consumption in the STT-RAM baseline). Importantly, the latter enables us to construct highly associative caches with STT-RAM in an energy-efficient manner, thereby realizing more even write distribution across different sets in a bank with very low overhead. We provide more analysis on the effectiveness of our intra-bank optimization in Section 5.2.

Third, our inter-bank optimization contributes an additional LLC energy reduction of 3.2% on average (up to 9.8%) on top of Benzene-Intra. More specifically, it reduces the write energy consumption by 10.4% on average. This is contributed by more even distribution of writes across different banks, which is important, especially considering that only 1/4 of the LLC is constructed with SRAM. We also observe that inter-bank optimization is more effective when the workloads are write intensive (e.g., M1 to M5 reduces LLC energy consumption by 7.5% on average), because balancing writes across different banks is more important if workloads generate a large amount of writes. Section 5.3 analyzes the impact of our inter-bank optimization in detail.

Figure 8 shows the dynamic energy consumption of Benzene-Intra and Benzene, normalized to the STT-RAM baseline. This clearly shows that the reduction in dynamic energy consumption is the key source of LLC energy reduction.

Performance. Figure 6 also compares the system performance of the baseline and our architecture. We use the sum of IPC of all applications (weighted speedup values have a similar trend) as the metric of performance comparison. We observe that our intra-bank optimization has 3.3% performance drop compared to the STT-RAM baseline. This is caused by inaccurate threshold adjustment of PHC for distributed caches (note that PHC is originally designed for centralized caches), which leads to 1.7 percentage points (pp) increase in miss rate on average as shown in Figure 8. Our inter-bank optimization adds a small performance overhead to it (1.2% on average), which comes from the increased network latency caused by our write-intensity-aware data placement. Note that there is a tradeoff relationship between energy reduction and performance overhead and our hop count constraint and adaptive inter-bank optimization allows this performance overhead to be tuned according to system requirement.

In summary, Benzene saves LLC energy by 47.1% with only 4.5% performance overhead.⁸ Although the benefit from inter-bank optimization may be offset by performance drop, inter-bank

⁸According to the modeling result from McPAT [24] for the baseline architecture (Silvermont-like core), LLC consumes 15% of total processor energy and the static energy of the rest of the components (excluding LLC) takes 13% of the total processor energy. If we combine these numbers with the energy reduction (26% over SRAM Baseline) and performance overhead (4.5%) of Benzene, then Benzene is still expected to be beneficial in terms of total energy consumption ($15\% * 0.26 - 13\% * 0.045 = 3.3\%$ reduction in terms of total processor energy consumption).

Fig. 9. Histogram of SRAM write distribution across different sets in a set-associative cache (above) and a highly associative cache with a tag-to-data pointer table (below).

optimization is still useful in terms of energy reduction, especially when the workloads are write intensive.

5.2 Analysis of Intra-Bank Optimization

Figure 9 shows the distribution of writes across 2,048 sets in a set-associative cache (same as the one in Figure 1) and a highly associative cache. The result is obtained from a 64-process workload running on a 64-core system as in ZCache [31]. In this figure, a histogram with more concentrated bars indicates more uniform distribution of writes across different cache sets. As shown in the figure, using a highly associative cache greatly improves the set-level write distribution. Quantitatively speaking, the standard deviation of SRAM writes per set is 2.86 times lower in a highly associative cache than in a set-associative cache for this particular example. In the end, this balanced write distribution across sets improves SRAM utilization, e.g., the ratio of SRAM write increases from 25.7% to 72.8%.

Note that it is the tag-to-data pointer table that enables such optimization *in an energy-efficient manner*. Simply applying a highly associative cache on top of a hybrid cache does not improve the LLC energy efficiency because of the following two reasons. First, without the tag-to-data pointer table, the relocation overhead of ZCache incurs 32.5% of LLC energy as explained in the previous subsection. Second, 20.7% and 37.9% of total relocation operations move write-intensive data from SRAM to STT-RAM and non-write-intensive data from STT-RAM to SRAM, respectively (as exemplified in Figure 4), which should not be moved. This makes hybrid cache management inefficient.

5.3 Analysis of Inter-Bank Optimization

Figure 10 shows the effectiveness of our inter-bank optimization on balancing the distribution of writes across 64 banks in a distributed cache architecture. The figure shows the distribution obtained from application mix M4. Similarly to the previous subsection, a histogram with more concentrated bars is more desirable in terms of write balance.

As shown in the figure, writes in the conventional distributed cache are concentrated to a few banks. In particular, we observe that some banks receive more than 7.6 $\times$ writes than the average writes per bank, which can degrade the effectiveness of hybrid caches. On the contrary, our

Fig. 10. Histogram of write distribution across different banks in a distributed cache before (above) and after (below) applying our inter-bank optimization.

Fig. 11. Normalized energy consumption of LLC and on-chip network in Benzene-Intra (left) and Benzene (right).

inter-bank optimization achieves more even distribution by adjusting the placement of data in a way to balance writes per bank across the distributed cache. As a result, our approach reduces the standard deviation of writes per bank by 46.6% on average compared to the conventional distributed cache.

Better balance in write distribution directly affects SRAM utilization in distributed hybrid caches. We observe that Benzene improves the ratio of writes to SRAM in a distributed hybrid cache by 4.7pp on average (up to 13.1pp) compared to Benzene-Intra. Since both Benzene and Benzene-Intra have almost the same number of writes in total, this indicates that Benzene puts more writes to SRAM and thus has higher SRAM utilization. This directly contributes to 6.5% (up to 17.1%) of the dynamic energy reduction shown in Figure 8.

5.4 Impact of Inter-Bank Optimization on Network Energy

As mentioned in Section 3.2, inter-bank optimization increases data accesses over on-chip network. Figure 11 compares Benzene-Intra and Benzene in terms of the total energy consumption including both LLC and on-chip network (the network energy is modeled by DSENT [36]). We do not compare other configurations because intra-bank optimization does not affect bank placement decision made by Jigsaw (i.e., no difference in network traffic between the baselines and Benzene-Intra). We observe that, even though inter-bank optimization increases the network energy consumption by 9.1% on average, the total energy is still reduced by 2.8% (up to 4.4%). For write-intensive workloads, the benefit from write energy reduction due to inter-bank optimization

Fig. 12. Impact of the hop count constraint on LLC energy (left), network energy (middle), and performance (right). The results are normalized to Benzene-Intra (Intra in the figure).

Fig. 13. The impact of adaptive inter-bank optimization on LLC energy consumption and performance. Benzene without adaptive inter-bank optimization (middle) and Benzene (right) compared to Benzene-Intra.

is larger than the additional overhead of network energy (note that the increase in network energy is controlled by hop count constraint). For non-write-intensive workloads, Benzene simply disables inter-bank optimization (i.e., adaptive inter-bank optimization), which makes it as efficient as Benzene-Intra. Therefore, we conclude that the inter-bank optimization reduces energy consumption, particularly for write-intensive workloads, even if we take the increased network energy consumption into account.

5.5 Sensitivity Analysis

Hop Count Constraint. Figure 12 shows the LLC/network energy consumption and the performance of Benzene under different hop count constraint λ (see Section 3.2) for our inter-bank optimization. In this figure, we always enable inter-bank optimization (i.e., no adaptive inter-bank optimization) to show the sole impact of hop count constraint. As shown in the figure, the looser the constraint is, the higher the performance drop and network energy consumption are. On the other hand, LLC energy consumption is reduced the most when $\lambda = 2$ because of the following reasons. When $\lambda = 1$, the hop count constraint is too restrictive to evenly balance the write distribution across all banks; when $\lambda \geq 3$, it degrades performance too much and thus increases the static energy consumption, which offsets the benefit of dynamic energy reduction. $\lambda = 2$ balances between the two and achieves maximal energy reduction, which is why we used $\lambda = 2$ throughout the experiments.

Adaptive Inter-Bank Optimization. Figure 13 shows the impact of inter-bank optimization on LLC energy consumption and performance. As shown in the figure, adaptive inter-bank optimization disables inter-bank write distribution for non-write-intensive workloads, where inter-bank optimization is not worthwhile in terms of energy reduction. In other words, adaptive inter-bank

Fig. 14. The impact of partitioning-aware sampling and better threshold adjustment on LLC miss rate (left) and speedup (right). The figure shows comparison among the STT-RAM baseline (**S**), Benzene-Intra without partitioning-aware sampling or threshold adjustment (**B**), Benzene-Intra with partitioning-aware sampling (**P**), Benzene-Intra with threshold adjustment (**T**), and Benzene-Intra with partitioning-aware sampling and threshold adjustment (**P+T**).

optimization allows us to combine the best of Benzene-Intra and Benzene according to the workload characteristics.

Other Optimizations. Figure 14 shows the impact of other optimization techniques, i.e., partitioning-aware sampling (**P** in the figure) and better threshold adjustment (**T** in the figure). When both **P** and **T** are applied (i.e., **P+T**), they improve the LLC miss rate by 0.3pp and the performance by 0.6% compared to the Benzene-Intra without them.

5.6 Implementation Overhead

Benzene introduces very small overhead to existing distributed hybrid cache architectures with distributed cache partitioning. The intra-bank optimization adds a tag-to-data pointer table, which has 8,192 thirteen-bit entries per LLC slice to establish one-to-one mapping from tags to data blocks. The inter-bank optimization mostly utilizes existing hardware (e.g., the write intensity predictor from PHC to estimate per-bank write intensity) and thus has a negligible area overhead. Partitioning-aware sampling adds a partition field to each sampler entry and a vantage controller to each sampler, which introduces 1KB storage overhead per sampler. Better threshold adjustment introduces an additional sampler for $\kappa = 0$, which adds 0.66KB per LLC slice. In total, Benzene introduces 17KB of storage overhead per LLC tile, or 2.1 bits per block.

Benzene runs intra-bank optimization periodically every 50 million cycles (25ms in our implementation). As in Jigsaw, it is performed by software on one of the cores in the system and the extra performance overhead caused by this is only 0.12%.

6 RELATED WORK

Hybrid Caches. Since we already explained the basic concept of hybrid caches in Section 2.2, we describe some of the representative hybrid cache architectures in the literature. The most naïve approaches [21, 25, 37, 41, 42] are to determine the block placement of each cache block based on the type of the instruction that generates the cache miss (e.g., SRAM on a store miss and STT-RAM on a load miss). Other approaches [8, 22, 23] give some hints at compile time and allocate data to SRAM or STT-RAM based on that information, which requires recompilation of applications for each architecture. Since such block placement may be incorrect, many of them combine their approach with migration, where blocks with frequent writes/reads are migrated to SRAM/STT-RAM to amend imperfect initial placement [8, 22, 25, 40, 41]. However, migration is known to incur a significant energy overhead, and that is why an architecture for a better block placement

policy (e.g., PHC) can outperform migration-based approaches. Although this article explains and evaluates Benzene based on a state-of-the-art hybrid cache that does not require migration (i.e., PHC), our inter-bank and intra-bank optimizations are general enough to be compatible with other hybrid cache architectures as well.

Other STT-RAM Cache Architectures. Other than hybrid caches, there have also been several approaches to reducing the write energy consumption of STT-RAM caches. For example, some studies have tried to design volatile STT-RAM cells at the device/circuit level to have lower write energy at the cost of shorter retention time [17, 35, 38]; cache bypassing can also reduce the energy consumption of STT-RAM LLCs [1, 9, 39]. The key difference between such techniques and ours is that their effectiveness is evaluated only on single- or multi-core systems, while ours targets at manycore systems for better scalability. Also, while our approach may not be directly applicable to all of those techniques, some of them can still benefit from the balanced write distribution of Benzene (e.g., hybrid caches composed of volatile/non-volatile STT-RAM cells [38]).

Highly Associative Caches. Since conventional set-associative caches directly use part of memory address as a set index, they suffer from uneven access distribution across the sets in the cache (explained in Section 2.4). In addition, since they select a replacement candidate within a set, blocks in frequently accessed sets are easier to be evicted than those in sets with infrequent accesses.

To mitigate this, highly associative caches are proposed. The skewed-associative cache [34] utilizes hash functions to pick more even distribution of set indexes from memory addresses, which improves the set-level write distribution and the effective associativity of the cache. ZCache [31] is another highly associative cache design that utilizes hash functions combined with cuckoo hashing to increase the number of replacement candidates. Thus, with only a small number of physical ways (e.g., 4 ways), ZCache outperforms set-associative caches with higher associativity (e.g., 32 ways). The V-Way cache [30] increases the number of sets in the tag array while maintaining the same data array size and maintains only the tag array as conventional set-associative caches with indirection from the tag array to the data array (called *forward pointer*). Since it uses more sets in the tag array, it reduces the chance of conflict misses.

Cache Partitioning. Cache partitioning is composed of an allocation policy and a partitioning scheme. The allocation policy decides the size of a partition to be allocated to each application. Based on the partition size information, the partitioning scheme enforces each application to use the allocated portion of the shared cache. Utility-based cache partitioning [29] is a state-of-the-art allocation policy that decides the optimal size of each partition by estimating the number of misses in each application under all possible partition configurations and choosing the one with the fewest cache misses. The most popular partitioning scheme is way partitioning [10], which partitions the cache ways in a set according to the partition decision and enforces each application to use its own ways; however, way partitioning is not scalable for manycore systems because the number of partitions is limited by the number of physical ways.

To overcome the limitations of conventional partitioning in manycore systems, there have been several approaches to *scalable* cache partitioning supporting a large number of partitions (e.g., 64 in our configuration). Promotion/insertion pseudo-partitioning (PIPP) [43] supports fine-grained partitioning by modifying the cache replacement policy; however, it does not strictly guarantee the size and the isolation of each partition. Vantage [32] utilizes highly associative caches [30, 31, 34] to support line-granularity partitioning based on a modified replacement policy. The key idea is to divide the cache into a main region (called *managed region*) and a victim cache (called *unmanaged region*) and control the rate of inserting cache blocks into each partition and demoting them to the unmanaged region to adjust the size of each partition. CloudCache [20] is another

partitioning scheme for manycore systems. Similarly to Jigsaw, it also distributes virtual caches to multiple banks with way partitioning, but it broadcasts local cache misses to reduce the cache latency, which degrades its scalability. Our architecture uses Jigsaw (which internally uses Vantage as a partitioning scheme of each LLC slice) as the baseline, but our ideas can be generalized to other cache partitioning schemes for manycore systems.

Decoupling Tag Array and Data Array. Our intra-bank optimization proposes a tag-to-data pointer table to decouple the tag array and the data array. There have been several approaches that use similar structures to enable a better cache replacement policy with many-to-one mapping from tags to data blocks [3, 30] or sharing of multiple data arrays across different nodes in a NUCA configuration [11, 12]. While our tag-to-data pointer table shares structural similarity with those prior approaches, our work is novel in that it leverages decoupling of the physical location of data blocks from that of tags (realized by the tag-to-data pointer table or similar structures) to *efficiently enable highly associative hybrid caches*, namely eliminating relocation overhead and avoiding unintended cross-region migration (i.e., STT-RAM to SRAM or vice versa).

7 CONCLUSION

In this article, we proposed Benzene, a scalable STT-RAM cache architecture for manycore systems. Our key observation is that (1) distributed cache architectures exhibit significant write imbalance in two different levels (intra-bank and inter-bank) and (2) such write imbalance leads to underutilization of SRAM resources in hybrid caches, thereby leaving room for energy efficiency improvement. Benzene leverages this observation and proposes two architectural optimizations. First, intra-bank optimization reduces write imbalance at the cache set level by exploiting highly associative cache design with a new hardware structure called tag-to-data pointer table. Second, inter-bank optimization achieves better write distribution across all banks in the distributed cache through our write-intensity-aware data placement policy. Our evaluation results show that, with this two optimizations, Benzene reduces the variance in the number of writes within and across the banks by 51.3% and 46.6%, respectively, thereby achieving a 47.1% reduction in LLC energy consumption. We believe that Benzene can facilitate *scalable* application of STT-RAM caches in manycore systems.

REFERENCES

- [1] Junwhan Ahn, Sungjoo Yoo, and Kiyoungh Choi. 2014. DASCA: Dead write prediction assisted STT-RAM cache architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- [2] Junwhan Ahn, Sungjoo Yoo, and Kiyoungh Choi. 2016. Prediction hybrid cache: An energy-efficient STT-RAM cache architecture. *IEEE Trans. Comput.* 65, 3 (2016), 940–951.
- [3] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llaberia. 2013. The reuse cache: Downsizing the shared last-level cache. In *Proceedings of the International Symposium on Microarchitecture*.
- [4] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.
- [5] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Proceedings of International Symposium in High Performance Computer Architecture*.
- [6] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlav Khan, Froilan Montenegro, Jay Stickney, and John Zook. 2008. TILE64-processor: A 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference Digest of Technical Papers*.
- [7] Xiuyuan Bi, Zhenyu Sun, Hai Li, and Wenqing Wu. 2012. Probabilistic design methodology to improve run-time stability and performance of STT-RAM caches. In *Proceedings of the International Conference on Computer-Aided Design*.
- [8] Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman. 2012. Static and dynamic co-optimizations for blocks mapping in hybrid caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

- [9] Hsiang-Yun Cheng, Jishen Zhao, Jack Sampson, Mary Jane Irwin, Aamer Jaleel, Yu Lu, and Yuan Xie. 2016. LAP: Loop-block aware inclusion properties for energy-efficient asymmetric last level caches. In *Proceedings of the International Symposium on Computer Architecture*.
- [10] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. 2000. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference*.
- [11] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2003. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the International Symposium on Microarchitecture*.
- [12] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the International Symposium on Computer Architecture*.
- [13] George Chrysos. 2012. Intel® Xeon Phi coprocessor (codename Knights Corner). In *IEEE Hot Chips Symposium*.
- [14] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai Li, and Yiran Chen. 2008. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Proceedings of the Design Automation Conference*.
- [15] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 31, 7 (2012), 994–1007.
- [16] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Arch. News* 34, 4 (2006), 1–17.
- [17] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proceedings of the Design Automation Conference*.
- [18] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *Proceedings of the International Conference on Computer Design*.
- [19] Samira M. Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of the International Symposium on Microarchitecture*.
- [20] Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. 2011. CloudCache: Expanding and shrinking private caches. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- [21] Jianhua Li, Liang Shi, Chun Jason Xue, Chengmo Yang, and Yinlong Xu. 2011. Exploiting set-level write non-uniformity for energy-efficient NVM-based hybrid cache. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia*.
- [22] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, and Yanxiang He. 2012. MAC: Migration-aware compilation for STT-RAM based hybrid cache in embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*.
- [23] Qingan Li, Mengying Zhao, Chun Jason Xue, and Yanxiang He. 2012. Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache. In *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*.
- [24] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2013. The McPAT framework formulticore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Trans. Arch. Code Optim.* 10, 1 (2013), 5:1–5:29.
- [25] Asit K. Mishra, Xiangyu Dong, Guangyu Sun, Yuan Xie, Vijaykrishnan Narayanan, and Chita R. Das. 2011. Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *Proceedings of International Symposium in Computer Architecture*.
- [26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report HPL-2009-85. HP Laboratories.
- [27] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms*.
- [28] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A case for MLP-aware cache replacement. In *Proceedings of International Symposium in Computer Architecture*.
- [29] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*.
- [30] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. 2005. The V-Way cache: Demand-based associativity via global replacement. In *Proceedings of the International Symposium on Computer Architecture*.
- [31] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the International Symposium on Microarchitecture*.
- [32] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of International Symposium in Computer Architecture*.
- [33] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of International Symposium in Computer Architecture*.

- [34] André Seznec. 1993. A case for two-way skewed-associative caches. In *Proceedings of International Symposium in Computer Architecture*.
- [35] Clinton W. Smullen IV, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- [36] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT-A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the International Symposium on Networks on Chip*.
- [37] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- [38] Zhenyu Sun, Xiuyuan Bi, Hai Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. 2011. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the International Symposium on Microarchitecture*.
- [39] Jue Wang, Xiangyu Dong, and Yuan Xie. 2013. OAP: An obstruction-aware cache management policy for STT-RAM last-level caches. In *Proceedings of the Design, Automation and Test in Europe*.
- [40] Zhe Wang, Daniel A. Jimenez, Cong Xu, Guangyu Sun, and Yuan Xie. 2013. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- [41] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. 2009. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the International Symposium on Computer Architecture*.
- [42] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, and Yuan Xie. 2011. Power and performance of read-write aware hybrid caches with non-volatile memories. In *Proceedings of the Design, Automation and Test in Europe*.
- [43] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture*.
- [44] Tianhao Zheng, Jaeyoung Park, Michael Orshansky, and Mattan Erez. 2013. Variable-energy write STT-RAM architecture with bit-wise write-completion monitoring. In *Proceedings of the International Symposium on Low Power Electronics and Design*.
- [45] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. Energy reduction for STT-RAM using early write termination. In *Proceedings of the International Conference on Computer-Aided Design*.

Received May 2017; revised November 2017; accepted December 2017