Data Similarity-Aware Computation Infrastructure for the Cloud

Yu Hua, *Senior Member*, *IEEE*, Xue Liu, *Member*, *IEEE*, and Dan Feng, *Member*, *IEEE*

Abstract—The cloud is emerging for scalable and efficient cloud services. To meet the needs of handling massive data and decreasing data migration, the computation infrastructure requires efficient data placement and proper management for cached data. In this paper, we propose an efficient and cost-effective multilevel caching scheme, called *MERCURY*, as computation infrastructure of the cloud. The idea behind *MERCURY* is to explore and exploit data similarity and support efficient data placement. To accurately and efficiently capture the data similarity, we leverage a low-complexity locality-sensitive hashing (LSH). In our design, in addition to the problem of space inefficiency, we identify that a conventional LSH scheme also suffers from the problem of homogeneous data placement. To address these two problems, we design a novel multicore-enabled locality-sensitive hashing (MC-LSH) that accurately captures the differentiated similarity across data. The similarity-aware *MERCURY*, hence, partitions data into the L1 cache, L2 cache, and main memory based on their distinct localities, which help optimize cache utilization and minimize the pollution in the last-level cache. Besides extensive evaluation through simulations, we also implemented *MERCURY* in a system. Experimental results based on real-world applications and data sets demonstrate the efficiency and efficacy of our proposed schemes.

Index Terms—Cloud computing, multicore processor, cache management, data similarity

1 Introduction

We are entering the era of the cloud that contains massive and heterogeneous data. The data sets have the salient feature of a volume of Petabytes or Exabytes and data streams with a speed of Gigabits per second. These data sets often have to be processed and analyzed in a timely fashion. According to a recent International Data Corporation (IDC) study, the amount of information created and replicated is more than 1.8 Zettabytes (1.8 trillion Gigabytes) in 2011 [1]. From 1700 responses to Science poll [2], about 20 percent respondents often use more than 100 GB data sets, over 63 percent have ever asked colleagues for data sharing. Unfortunately, about half of those polled store the data only in their own labs due to no funding to support archiving. Moreover, from small hand-held devices to huge data centers, we are collecting and analyzing ever-greater amounts of information. Some commercial companies, like Google, Microsoft, Yahoo!, and Facebook, generally handle Terabytes and even Petabytes of data everyday [3], [4], [5],. For the computation infrastructure of the cloud, it is important and challenging to perform efficient processing and analysis upon these data.

The computation infrastructure typically consists of multicore processors. The increasing number of cores on a chip and the different degrees of data similarity exhibited within the workloads present the challenges to the design of cache hierarchies in Chip multi processors (CMPs). These include the organization and the policies associated with the cache hierarchy to meet the needs of system performance improvements and scalability. Cache organization presents multiple levels in the cache hierarchy as well as the size, associativity, latency and bandwidth at each level. Suitable policies help minimize the latency to frequently accessed data [6], [7], [8], [9]. Moreover, CMPs are prevalent these days. Vendors already ship CMPs with four to twelve cores and have the roadmaps to release hundreds of cores to the market. For example, in the commercial markets, there are Tilera TILE64, Ambric Am2045 and Nvidia GeForce GT200. They are widely used in cloud applications. Despite the popularity, it is still a daunting task to accurately and efficiently perform the multicore caching for high performance cloud systems. The focus of this paper is to optimize the data placement of the multilevel cache hierarchy (e.g., L1, L2 caches and main memory) to improve the overall cloud system performance.

Efficient cache hierarchy in the cloud needs to answer the questions, such as *"how to significantly improve the cache utilization and how to efficiently support the data placement?"* These problems are more difficult and challenging to address, especially in the case of large core count. Specifically, we need to address the following challenges.

Challenge 1: *inconsistency gap between CPU and operating system caches*. To bridge the speed gap between CPU and memory, CPU caches (e.g., L1 and L2) and operating system (OS) buffer cache are widely used in a multilevel cache hierarchy. Since the CPU caches are at the hardware level, while the buffer cache is a part of the OS, these two layers are conventionally designed independently without the awareness of each other. This possibly works for small-scale systems. However, with the increments of multicore amounts and increasingly large capacity of main memory, severe performance degradation may occur once the inconsistency gap exists. These two layers hence

• *Y. Hua and D. Feng are with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, P.R. China. E-mail: {csyhua, dfeng}@hust.edu.cn.*
• *X. Liu is with the School of Computer Science, McGill University, Montreal H3A 0E9, Canada. E-mail: xueliu@cs.mcgill.ca.*

Manuscript received 7 Sept. 2012; revised 10 Feb. 2013; accepted 24 Apr. 2013; published online 3 May 2013.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2012-09-0642.

become inefficient to work cooperatively. Moreover, by leveraging a shared cache, a thread, which cooperatively works with multiple corunning threads, can influence each other. This generally leads to severe performance degradation. In the near future, a cache will be shared by many cores, and the gap may degrade the performance even more seriously [10], [11].

Challenge 2: *performance bottleneck shift in high performance cloud systems.* Multicore-based hardware advancements bring new challenges to the design and the implementation of high-performance cloud systems [12]. This is because the performance bottleneck has been shifted from slow I/O access speeds to high memory access latency. The performance bottleneck of accelerating the execution is correlated with the placement problem of cached data. The optimization of cached data placement hence becomes important to improve the overall cloud system performance. Unfortunately, the existing policies in the multicore processors become neither efficient nor scalable to address the data placement problem. To efficiently address this problem, we need to carefully explore and exploit the data similarity that generally hides behind access behaviors. We also need to optimize the capacity utilization of a private cache, while alleviating uncontrolled interference in a shared cache.

Challenge 3: exacerbation of the LLC pollution. The last-level cache (LLC) [6] is dynamically shared among the cores while each core has its lowest level of the cache hierarchy. Cache pollution refers to the replacement of a cache element by a less useful one. It occurs when a non-reusable cache line is installed into a cache set. The installed line displaces a reusable cache line. To alleviate the LLC pollution, conventional approaches have the premise that recent ordering serves as the good prediction for subsequent behaviors of cache accesses [10], [13]. In practice, although leveraging the access patterns helps predict future accesses, the caches have to install all cache lines that are accessed. Since performing the identification on the access patterns incurs heavy temporal and spatial overheads, the existing approaches generally demonstrate unsatisfactory performance. Long latency and information staleness further exacerbate the LLC pollution. What we need is a new scheme that simplifies the identification of access locality without the loss of accuracy.

Our proposed *MERCURY* alleviates the limitations in the hardware solutions and the OS-based schemes. The rationale comes from the observation that performing the state maintenance and reference pattern analysis at page granularity generally incurs less overhead than at block [6], [10], [13]. Moreover, learning dynamic reference patterns at page granularity requires less state and storage space compared with the already studied block-grain policies. Our research work hence is related with two areas: system architecture and data-intensive cloud. The two areas are traditionally distinct, but the gap on common system concerns between them has been narrowed recently. The similarity-aware *MERCURY* meets the needs of suitable data placement in the multilevel cache hierarchy. We implement *MERCURY* and manage the similarity at a granularity of pages by leveraging the operating system mechanisms. *MERCURY* is compatible with the existing cloud computing systems and can further improve upon them by providing a scalable and efficient caching scheme. *MERCURY* plays a significant and fruitful role in managing the multilevel cache hierarchy. Specifically, we make the following contributions.

First, (for Challenge 1), to narrow the inconsistency gap and quantify the data correlation, *MERCURY* employs multitype, rather than conventional homogeneous, membership management. Here, the membership refers that an item belongs to a given data set. The data in the similarity-aware multicore caches are judiciously classified into three types, i.e., *Family, Friend*, and *Foreigner*, to respectively represent *frequently accessed and correlated*, *frequently accessed but not correlated*, and *infrequently accessed* memberships. To guarantee the data consistency and integrity, we further quantify these memberships using a new coding technique.

Second, (for Challenges 2 and 3), to address the performance bottleneck and alleviate the LLC pollution, *MERCURY* explores and exploits the access locality by improving a multicore-enabled locality-sensitive hashing (MC-LSH). The MC-LSH uses a self-contained and space-efficient *signature vector*, rather than many hash tables in a standard locality-sensitive hashing (LSH), to accomplish the significant space savings and meanwhile accurately measure the data similarity. Since *MERCURY* minimizes cache conflicts and reduces the amounts of the migrated data, it significantly reduces the low-speed memory accesses. *MERCURY* can accurately identify the data similarity and mitigate the staleness of cached data to meet the needs of high-performance cloud systems.

Third, we have implemented the components and the functionalities of *MERCURY* in a software layer, which is compliant with the existing hardware devices. To further examine and evaluate the efficacy and efficiency of the proposed scheme, we not only examine *MERCURY* in a multicore simulation [7], [14], [15], but also implement it in a production system by patching PostgreSQL [16]. The extensive experiments use real-world traces and data sets, and examine the performance in multiple evaluation metrics.

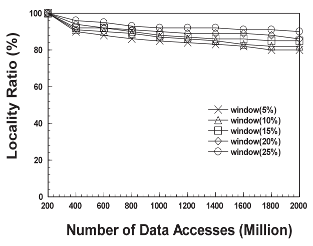
The remainder of this paper is organized as follows: Section 2 presents the data sets analysis and problem statement. Section 3 describes the proposed *MERCURY* architecture and caching schemes. Section 4 shows the cached data management schemes in the multilevel hierarchy. Sections 5 and 6, respectively, demonstrate the performance evaluation results in simulations and implementations. We present the related work in Section 7. Finally, we conclude our paper in Section 8 with summaries of findings.

2 Problem Statement

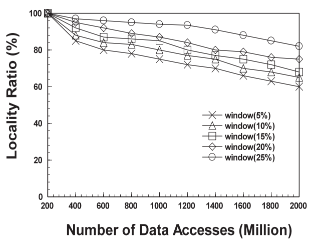
In this section, we first study workload characteristics to show the existence of data similarity and demonstrate its performance impacts on caching schemes. We also present the problem statement and basic ideas of our work.

2.1 Analysis of Real-World Traces

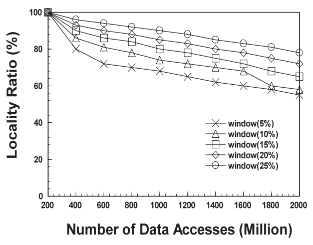
It is well recognized that the property of data similarity is helpful to perform an efficient and scalable caching [7], [11], [17], [18], [19], [20], [21]. Main benefits include throughput improvements and the reduction of the LLC cache miss rates, query latency, and data migration overheads. Hence, the motivation of *MERCURY* design comes from the



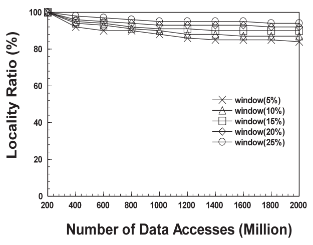
(a) CoverType.



(b) EECS.



(c) HP.



(d) vpr& twolf.

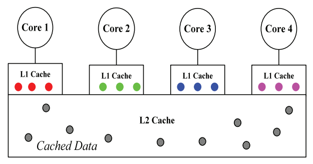
Fig. 1. Average locality ratios from real-world data sets.

observations of data similarity widely existing in the real-world applications. Furthermore, we present the definition of data similarity. For two data with point representations as a and b, we assume that they have d-dimensional attributes that are represented as vectors $\vec{a_d}$ and $\vec{b_d}$. If the geometric distance between vectors $\vec{a_d}$ and $\vec{b_d}$ is smaller than a predefined threshold, they are *similar*. The data similarity often hides behind the locality of access patterns [8].

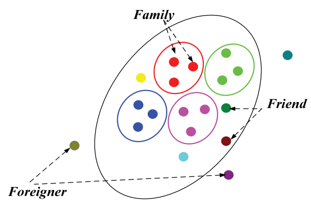
We study typical large-scale applications [22], [23], [24] and the main benchmarks from the SPEC2000 evaluation [25], i.e., *175.vpr* and *300. twolf*. The properties of used traces and data sets are listed:

1. The CoverType data set [22] contains 581,012 data points, each of which has 54D attributes.
2. The EECS NFS server at Harvard [23] collected I/O accesses. This data set contains concurrent requests with a total of 4.4 million operations.
3. The HP file system provides a 10-day 500-GB trace [24] that records the accesses from 236 users.
4. The 175.vpr and 300.twolf benchmarks show the CPU performance in the SPEC2000 evaluation [25]. 175.vpr leverages combinatorial optimization technique to automatically synthesize the mapped circuits. The 300.twolf makes use of TimberWolfSC placement and global routing package.

To obtain explicit demonstration, we intensify the above traces and benchmarks into larger scales by a combination of spatial scale-up and temporal scale-up. Specifically, the scale-up method needs to first decompose a trace into subtraces, where the timing relationships are preserved to faithfully maintain the semantic dependences among trace records. These subtraces are replayed concurrently by setting the same start time. Note that, the combined trace maintains the same histogram of system calls as the original trace, but presents a heavier workload (higher intensity). As a result, data can be both spatially and temporally scaled up by different factors, depending upon the number of subtraces replayed simultaneously. We intensify experimental data to be scaled up to 2,000 million accesses.



(a) Cached data membership.



(b) Differentiated data placement.

Fig. 2. Problem description.

We use *locality ratio* as a measure to represent the locality in the access pattern. Fig. 1 shows the results of locality ratio that is the percentage of the times accessing data within defined time interval to those in the entire data set. The time interval comes from the used traces, in which all accesses are listed in the order of time. For instance, a data set contains a 20-hour trace record and we select a 25 percent interval, i.e., 5-hour access record. For a file, if it has 8 accesses within a randomly selected 25 percent time interval, and the accessed times during the entire running trace is also 8 (i.e., all accesses to this file occur within this 25 percent interval), the locality ratio becomes $8/8 = 100\%$. We observe that there exists a strong data access locality within certain number of instances. The observations also conform to the conclusions in [7]. According to our experimental results and observations, similar data generally demonstrate the locality of access patterns. If they are placed together, we can improve cache utilization and decrease the complexity and execution costs of data access operations.

2.2 Basic Idea

The hardware design of cloud computation infrastructure still works for scenarios they are designed for, but the lack of flexibility can be an unavoidable issue and inherent weakness, particularly for multicore or many-core processors with an increasingly large number of cores. In contrast cache optimization and cache resource management at different levels of software, such as operating systems, compilers, and application programs have shown their effectiveness to address the limitations of hardware solutions. With a software approach, long-term memory access patterns of most cloud applications can be analyzed or predicted, cache management and optimization decisions can be made more effectively. There have been several successful examples on uniprocessors with simple LRU cache replacement policies [18], [26], [27], [28]. However, using a software scheme to manage cache resources in multicore processors is much more challenging than in uniprocessors, since hardware resources are almost not shared and coordinated for multithreads. Researchers have evaluated OS-based cache partitioning methods in multicore processors without any additional hardware support [29]. The OS-based method offers the flexibility in implementing various resource allocation policies. However, since hardware caches are not in the scope of OS management, the OS-based methods can inevitably cause nontrivial software overhead, and are not ready to be used as computation infrastructure of the cloud.

To offer efficient computation infrastructure for the cloud, we investigate the problem of the cached data placement in the multilevel cache hierarchy when executing multiple parallel instances as shown in Fig. 2. We term this problem as "*cache-member*," which determines the data memberships in each cache based on the given constraints. The constraints include migration costs and data access latency.

Fig. 2a shows the cache-member problem. Specifically, we need to first identify and aggregate similar data into the same or adjacent private L1 caches, and then allocate the data accessed by more than one core into a shared L2 cache. We, hence, can manage the cached data in both L1 and L2 caches. Moreover, an ideal multicore architecture is scalable and flexible to allow dynamic and adaptive management on the cached data. The premise is to accurately capture the similar data [7], [17], which unfortunately is nontrivial due to expensive operation costs of comparing arriving data with all the existing cache lines.

We identify the problem of homogeneous data placement that overlooks the distinct properties and multitype memberships of cached data. To alleviate the homogeneous data management, we leverage a differentiated placement policy, in which the cache memberships are classified into three types as shown in Fig. 2b. We place frequently accessed and correlated data into L1 cache, called "*in-cache Family*," frequently accessed but loosely correlated data into L2 cache, called "*shared-cache Friend*" and infrequently accessed data into main memory, called "*in-memory Foreigner*." In this way, we can differentiate the strength of access locality to facilitate the efficient placement of cached data.

In practice, capturing data similarity is timeconsuming and computation-intensive work due to high dimensions and heterogeneous types. Hence, to accomplish a suitable tradeoff between similarity accuracy and operation complexity, we propose to use a hash-based approach, for example, the LSH [30], due to its locality-aware property and ease of use. The LSH can identify and place similar data together with low complexity. The rationale is that similar data contain strong locality to match access patterns of multiple threads. The LSH-based scheme, thus can improve system performance. Unfortunately, it is well recognized that a standard LSH suffers from heavy space overhead due to the use of too many hash tables [31], [32], [33]. Moreover, data placement policy depends upon both access frequency and correlation, which is currently difficult to be represented *quantitatively* and measured accurately.

The basic idea behind *MERCURY* is to leverage the MC-LSH to identify similar data and carry out differentiated data placement. *MERCURY* represents the strength of data similarity as *Family*, *Friend*, and *Foreigner*, respectively, as shown in Fig. 2b. Specifically, the private L1 caches contain *Family* members, which are tightly correlated and frequently used data to facilitate the fast access and maintain the access locality in each cache. Furthermore, a shared L2 cache contains *Friend* members, which in fact consist of two parts. One is the data frequently accessed by multiple cores and the other is the data evicted from correlated L1 caches due to space limitation or staleness. Finally, the main memory contains *Foreigner* members that are not included in the L1 or L2 caches. Differentiated data placement comprehensively considers both the strength of data similarity and access frequency, while allowing the flexible adjustments to support dynamic operations (e.g., insertion/deletion). The

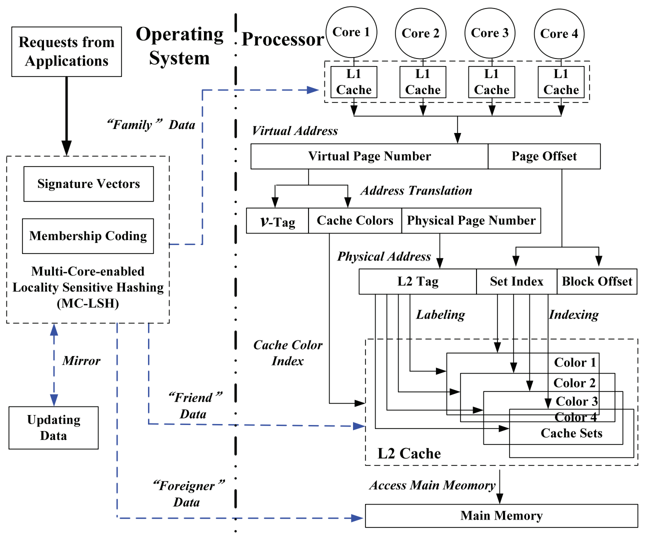


Fig. 3. *MERCURY* multicore caching architecture.

similar data that are placed closely can also significantly reduce the migration costs. *MERCURY*, hence, offers scalable, flexible, and load-balanced caching schemes in a multilevel cache hierarchy.

MERCURY is implemented in a hybrid scheme to address the limitations of both hardware solutions and OS-based methods. Specifically, our multicore shared-cache-management framework consists of two low cost and effective components: a lightweight mechanism for allocating cache resources and providing cache usage information; and OS-based resource allocation policies for dynamic cache allocation. With a simple and low-overhead component, we enable direct OS control over shared caches, and the software system overhead is minimized. With an OS-based management, we are able to design and implement multiple policies to deal with complicated, difficult caching scenarios in multicore systems.

3 *MERCURY* Architecture

MERCURY uses the MC-LSH to identify similar data and leverages an LRU replacement in each cache to update stale data. Fig. 3 shows the *MERCURY* architecture in the multilevel hierarchy. We assume that each core has one private L1 cache and all processor cores share an L2 cache. The *MERCURY* scheme is tightly associated with two parts. One is the processor architecture and the other is the operating system. Furthermore, to explicitly represent the differentiated memberships identified by the MC-LSH, we use different flags to label each cache line and obtain holistic optimization in the multilevel cache hierarchy.

3.1 Caches in a Multicore Processor

The caching schemes in a multicore processor include L1 and L2 cache management, and virtual-physical address translation.

L1 cache management. Each core has one associated cache that contains frequently visited data to increase the access speed and decrease the required bandwidth. We need to update the stale and infrequently accessed data.

L2 cache management. To partition the shared L2 cache, we leverage the well-known page color [34] due to its simplicity and flexibility. Page coloring is an extensively used OS technique for improving cache and memory performance. A physical address contains several common bits between the cache index and the physical page number, which is indicated as a page color. One can divide a physically addressed cache into nonintersecting regions (cache color) by page color, and the pages with the same page color are mapped to the same cache color. A shared cache is divided into N colors, where N comes from the architectural settings. The cache lines are represented by using one of N cache colors. We assign the cache colors of the virtual pages by using the virtual-to-physical page mapping.

Address translation. The address translation can translate the virtual address into the physical address by reading page table. The cache color is tightly associated with the number of page colors in the L2 cache. A virtual Tag (v-Tag) helps to identify similar data by using the results from the MC-LSH computation.

3.2 Operating System

The operating system functionalities support the MC-LSH computation and update the locality-aware data.

MC-LSH. A standard LSH helps identify similar data and unfortunately incurs heavy space overhead, i.e., consuming too many hash tables, to identify the locality-aware data. The space inefficiency often results in the overflowing from a limited-size cache. *MERCURY* proposes to use an MC-LSH to offer efficiency and scalability to the multicore caching. Specifically, the MC-LSH uses a space-efficient signature vector to maintain the cached data and utilizes a coding technique to support a differentiated placement policy for the multitype data. We will describe the design details of the MC-LSH in Section 4.

Updating locality-aware data. To execute fast and accurate updates, a key function in *MERCURY* is to identify similar data with low operation complexity. In practice, many high-performance computing applications demonstrate the identical data at the same virtual address, but different physical addresses [7]. All relevant virtual addresses thus need to be mapped to the same cache set. We make use of the MC-LSH to identify similar data and avoid brute-force checking between arriving data and all valid cache lines. The similar data are then placed in the same or close-by caches to facilitate multicore computation and efficiently update data. Since the cached data are locality-aware, *MERCURY*, hence, decreases migration costs and minimizes cache conflicts.

To satisfy query requests and provide flexible use, we design an interface between high-performance applications and operating system as shown in Fig. 3. Its main function is to wrap high-level operation requests to low-level system calls with the aid of the page coloring technique [34]. Page color manages the bits between the cache index and the physical page number in the physical memory address. Specifically, the applications need to specify the required space in their requests. The requests help decide how to partition available cache space among query requests. Query execution processes indicate partitioning results by updating a page color table. The operating system then reads the page color table to know the cache partitions among the query requests.

Although the operating system cannot directly allocate on-chip cache space, it can make use of virtual-physical address mapping to control how to allocate pages in the main memory. The memory pages of the same color can be mapped to the same cache region. To efficiently partition the cache space, we allocate different page colors to memory threads. *MERCURY* can hence leverage the page coloring technique to complete cache partitioning among different processes and support the queries.

4 Cached Data Management in *MERCURY*

To capture the data similarity, we propose an MC-LSH design in *MERCURY*. A space-efficient signature vector and a simple coding technique help maintain and represent the multitype memberships. We finally describe the scheme of updating data in *MERCURY*.

4.1 The MC-LSH Scheme

The MC-LSH is a multicore-enabled scheme that consists of the LSH-based computation, a signature vector structure and the multitype membership coding technique. It offers a deterministic membership for each data item. Compared with the conventional classification schemes for exact results, the MC-LSH provides an approximate and fast scheme to obtain significant time and space-savings. The MC-LSH employs the LSH functions to identify similar data based on the access patterns. To address the problem of space inefficiency (i.e., too many hash tables) in the standard LSH, we employ a signature vector structure. Furthermore, to offer differentiated data placement, we use a multitype membership coding technique.

Limitations of the standard LSH. An LSH [30] captures similar data by allowing them to be placed into the same hash buckets with a high probability.

Definition 1. *Given a distance function $\|*\|$, a data domain S, and some universe U, an LSH function family, i.e., $\mathbb{H} = \{h : S \rightarrow U\}$ is called (R, cR, P_1, P_2)-sensitive, if for $\forall p, q \in S$:*

- *If $\|p, q\| \leq R$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \geq P_1$,*
- *If $\|p, q\| > cR$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \leq P_2$,*

where $c > 1$ and $P_1 > P_2$.

In $\mathbb{H}$, $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{\omega} \rfloor$. a is a d-dimensional random vector with chosen entries following an s-stable distribution and b is a real number chosen uniformly from the range $[0, \omega)$, where ω is a constant. By using the LSH functions, similar data have a higher probability of colliding than the data that are far apart [35].

Although the LSH has been recently used in many applications, it is difficult to be used in the multicore systems due to heavy space overhead and homogeneous data placement. These limitations have severely hampered the use of the multicore benefits for high-performance systems. Unlike the existing work, *MERCURY* enables the LSH to be space-efficient by leveraging signature vectors.

Space-efficient signature vector. The MC-LSH leverages space-efficient signature vectors to store and maintain the locality of access patterns. Specifically, a signature vector is an m-bit array where each bit is initially set to 0. There are totally L LSH functions, $g_i (1 \leq i \leq L)$, to hash a data point

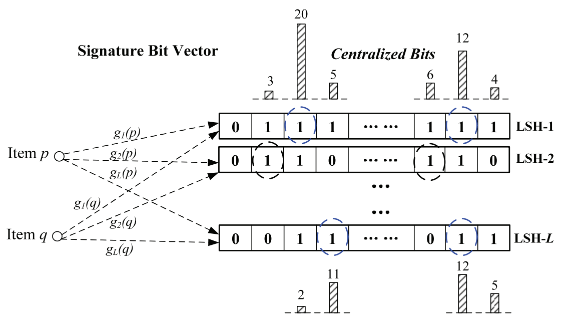


Fig. 4. Signature vector for maintaining page-level data similarity.

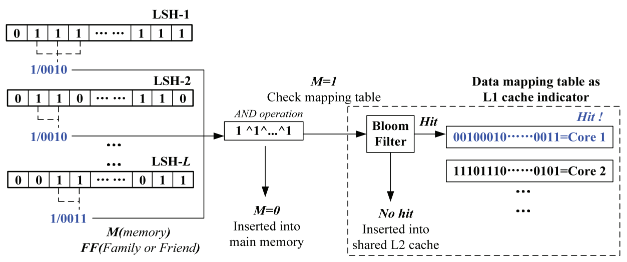


Fig. 5. Differentiated membership coding technique.

into bits, rather than its original buckets in hash tables, to significantly decrease space overhead. A data point as an input of each hash function g_i is mapped into a bit that is thus set to 1 possibly more than once and only the first setting takes effect.

A signature vector is able to maintain the data similarity as shown in Fig. 4. A *centralized bit* is the bit that receives more hits than its left and right neighbors. The hit numbers as shown in this figure are also much larger than a predefined threshold value. The centralized bits become the centers of correlated data and are further selected to be mapped and stored in the L1 caches. When hashing data into the signature vector, we count the hit numbers of bits and carefully select the centralized bits. Moreover, the threshold demonstrates the clustering degree of data distribution, thus depending upon the access patterns of the real-world applications. After selecting the centralized bits, we can construct a mapping between the centralized bits and L1 caches to facilitate the data placement. It is worth noting that the number of centralized bits is unnecessarily equal to that of the L1 caches. If the number of centralized bits is larger than that of L1 caches, an L1 cache may contain the data from more than one adjacent centralized bits.

The MC-LSH computation can guarantee similar data to be hashed into one bit with very high probability that however is not 100 percent, meaning that similar data are still possible to be placed into adjacent bits. False negative, hence, occurs when the hit bit is 0 and one of its neighbors is 1. To avoid potential false negatives, a simple solution is to check extra neighboring bits besides the hit one. Although extra checking on neighboring bits possibly incurs false positives, in practice, a miss from the false negative generally incurs the larger penalty than the false positive.

A reasonable size of checking extra bits is acceptable to obtain a suitable tradeoff between false negatives and false positives. *MERCURY* probes more than one hit bit, i.e., checking left and right neighbors, besides the hashed bit. Note that, the extra checking occurs only when the hit bit is "0". Our result conforms to the conclusion of sampling data in the multiprobe LSH [31].

To efficiently update the signature vectors, *MERCURY* offers scalable and flexible schemes based on the characteristics of the real-world workloads. Specifically, if the workloads exhibit an operation-intensive (e.g., write-intensive) characteristic, we can carry out the operations on the signature vectors and allow the (re)-initialization in the idle time. Moreover, if the workloads become uniform, *MERCURY* makes use of 4-bit counters, rather than bits, as the summary of Bloom filters [36]. Each indexed counter increases when adding an item and decreases when removing an item. In practice, a 4-bit counter can satisfy the requirements for most applications.

Multitype membership coding. The memberships in the MC-LSH include *Family*, *Friend*, and *Foreigner*, which respectively represent different similarities among cached data. The MC-LSH identifies data memberships and places data into L1 cache, L2 cache, or main memory, respectively. One key issue in the data placement is how to determine whether the hits in multiple LSH vectors indicate a single cache. To address this problem, we use a coding technique to guarantee membership consistency and integrity.

We use an example to illustrate the differentiated membership coding as shown in Fig. 5. Given an item, we first compute its hashed values by using hash functions in the signature vector to determine whether it is correlated to one of the existing L1 caches. Based on the conclusion in [31], if the hit bit is any of the centralized bit, its left and right neighbors, the item is considered to be correlated with the corresponding cache and further obtains an $M = 1$ indicator (i.e., in the memory), together with FF (*Family/Friend*) code (e.g., the location) of that centralized bit in an LSH array.

We construct a mapping table between arriving data and multicore caches to facilitate differentiated data placement. If all M indicators from L LSH arrays show 1 for an item by using a bit-based *AND* operation, we determine that this item is correlated with multicore caches to execute further checking on the data mapping table. Otherwise, the item is not considered to be correlated and directly inserted into the main memory. The checking on the table allows to determine whether the item is a *Family* or *Friend*. Since performing direct searching on the entire table consumes too much time, we first hash the concatenated code of that item into a standard Bloom filter [37] that has already stored the code indicators. If a hit occurs, we continue to perform the checking on the mapping table. Otherwise, the item is considered as a *Friend*, and then inserted into the shared L2 cache. Furthermore, since the table contains too many code indicators, the linearly brute-force searching will lead to unacceptable costs, possibly becoming the performance bottleneck. To address this issue, we make use of a hash table to maintain these code indicators and decrease the searching latency. When a hit occurs in the mapping hash table, we insert this item into the corresponding L1 cache.

4.2 Updating Data

In the multilevel hierarchy of *MERCURY*, we need to update cached data and their memberships in the signature vector.

For updating cached data. To update actual data, we make use of a label-based technique to update stale data in multilevel caches. The reason comes from the fact that similar data are potentially reused by corresponding caches in the near future. To decrease recaching costs, we temporarily label stale data for certain time. When the time expires, we update the caches and replace these labeled stale data. Moreover, the L1 caches belonging to multiple cores possibly contain different amounts of similar data. Performing the load balance within multiple L1 caches is hence important to obtain performance improvements. Due to the limited-size capacity in each L1 cache, *MERCURY* temporarily places excess, but correlated data into the shared L2 cache. These correlated data have been inserted into corresponding counting Bloom filters [37]. In the shared L2 cache, we label the data by using page colors of the correlated cores to update caches. Once free space is available in an L1 cache, *MERCURY* reloads these labeled data into the corresponding L1 cache.

The operations of updating data are actually a multilevel migration process from the L1 cache, then the L2 cache, finally to the main memory. The workflow steps are described below.

1. Updating cache in *MERCURY* needs to replace stale data in both L1 and L2 caches, while guaranteeing high hit rates and low maintenance costs. *MERCURY* makes use of the MC-LSH to identify similar data that are then placed into the L1 caches.
2. The L1 caches employ the simple LRU replacement to update stale data.
3. When the data in the L1 caches become stale, they are transferred into the shared L2 cache among multiple cores.
4. When the data in the L2 cache become stale, they move to the main memory.

For updating memberships. To update the data membership in the signature vectors, we leverage counting Bloom filters to facilitate the data deletion and maintain the membership of the data that have been identified to be correlated and placed into the corresponding L1 caches. The counting Bloom filters help maintain the membership of cached data in a space-efficient way, carry out the initialization of the L1 caches and keep the load balance among multiple L1 caches. Each counting Bloom filter is associated with one L1 cache.

When an item is inserted into the L1 cache, it is meanwhile inserted into the counting Bloom filter, in which the hit counters are increased by 1. Since each counting Bloom filter only needs to maintain the items existing in the corresponding L1 cache and the number of stored data is relatively small, thus not requiring too much storage space. Moreover, when deleting an item, the hit counters are decreased by 1. If all counters become 0, meaning that there are no cached data, we initialize the associated caches by sampling data to determine the locality-aware representation in the signature vector. Note that, the size of a signature vector depends on not only the amounts of data to be inserted, but also their distribution. We, hence, leverage well-recognized sampling methods [31], [35], [38], [39] to obtain the suitable size.

TABLE 1
Simulation Parameters

Parameters	Values
Processor	4/8/16 cores
Issue/Commit	8/8
ALU/FPU/Mult/Div	4/4/1/1
I-Fetch Q/LSQ/RUU	16/64/128
Branch predictor	2-level 1024 entry, history length 10
BTB size	4K-entry and 4-way
RAS entries	16
Branch penalty	3 cycles
DRAM latency	200 cycles
L2 Cache (shared)	4MB, 8-way, 64B lines
L2 Latency	6 cycles
L1 Cache (per core)	64KB Inst/64KB Data, 64B lines, 2-way
L1 Latency	1 cycle
Memory regions	32-memory region/process
Cache color	8/core

5 Performance Evaluation

This section presents the performance evaluation of our proposed scheme by describing simulation framework and examining the scalability of *MERCURY* compared with the state-of-the-art work.

5.1 Experiment Configuration

We use simulation study primarily for the evaluation of *MERCURY*'s scalability. Our simulation is based on PolyScalar that is widely used in the multicore simulation [7], [14], [15]. We add page tables into PolyScalar for each process to enhance its virtual-to-physical address translation functionality. We further improve PolyScalar by adding the similarity-aware functionalities that are described in Sections 3 and 4. The size of each OS page is 8 KB. Since our study focuses on the last-level cache (L2 cache) that has strong interaction with the main memory, we extend PolyScalar to simulate DDR2 DRAM systems. The simulated memory transactions are pipelined.

MERCURY leverages the MC-LSH to identify similar data that are placed into L1 and L2 caches, respectively, with an LRU replacement policy. Specifically, each processor has its own private L1 cache. An L2 cache is shared by multiple cores. We evaluate the scalability of *MERCURY* by increasing the number of cores. In the page color policy of the L2 cache, each core has eight colors and each color has 128 cache sets. We, hence, allocate 1-MB cache for 4-core system, 2-MB cache for 8-core system, and 4-MB cache for 16-core system. Table 1 shows the parameter settings in the simulations.

The used traces and data sets include Forest CoverType data set [22], EECS NFS server at Harvard [23], HP file system trace [24], and 175.vpr and 300.twolf in SPEC2000 [25]. Moreover, by using the proposed sampling approach [31], [35], [38], [39] described in Section 4.2, the suitable sizes of signature vectors are 7.6 KB in 175.vpr, 7.9 KB in 300.twolf, 8.3 KB in CoverType, 8.7 KB in EECS, and 9.2 KB in HP.

TABLE 2
Performance Evaluation Metrics

Metric	Description
Throughput	$\sum_{i=1}^{n}(IPC_{scheme}[i])$
Weighted Speedup	$\sum_{i=1}^{n}(IPC_{scheme}[i]/IPC_{base}[i])$
Fair Speedup	$n/\sum_{i=1}^{n}(IPC_{base}[i]/IPC_{scheme}[i])$

We use the multiple metrics to evaluate the performance, including *Throughput*, *Weighted speedup*, and *Fair speedup* as shown in Table 2. Specifically, the *Throughput* refers to the absolute IPC numbers to evaluate the system utilization. The *Weighted speedup* is the sum of speedups of all programs over a baseline scheme to indicate the decrease of execution time. The *Fair speedup* is the harmonic mean of the speedups over a baseline scheme to obtain the balance between fairness and performance. We also examine the performance in terms of cache update latency, migration cost, hit rate and time, and space overheads.

5.2 Results

We compare *MERCURY* with baseline approaches, i.e., *private* and *shared* caches, and the state-of-the-work, PCM [12] and Mergeable [7] schemes, which we reimplemented for the experiments.

5.2.1 Throughput

Fig. 6 shows the throughput results from executing the real-world applications with the increase of multicore number from 4 to 16. The average throughputs on 4-core systems with private cache, shared cache, PCM, Mergeable, and *MERCURY* are 1.352, 1.563, 1.815, 1.925, and 2.162, respectively. For 8-core systems, the average throughputs are 2.481, 2.572, 2.953, 3.104, and 3.305. For 16-core systems, they are 3.281, 3.469, 3.957, 4.152, and 4.452.

We observe that two typical SPEC2000 benchmarks obtain the larger throughputs on average by 15.7 percent increase than other applications. The main reason is that the SPEC2000 benchmarks have better similarity in the access pattern, thus allowing the LSH to accurately and efficiently capture correlated data. In addition, *MERCURY* executes constant-scale hashing computation to quickly and accurately identify correlated data, thus obtaining the larger throughput than the PCM and Mergeable schemes.

5.2.2 Weighted Speedup

We take into account the changes of the relative IPC that is the ratio of absolute IPC to the baseline as the metric of the weighted speedup. The weighted speedups are normalized to those with the private caches as shown in Fig. 7. The shared cache obtains better performance than the private cache due to the ability to adapt to the demands of competing processes. Compared with the private cache, the improvements of the shared cache are 9.87, 17.52, and 23.67 percent, respectively, on 4-core, 8-core, and 16-core systems.

The PCM, Mergeable, and *MERCURY* have much better performance than the shared cache. The average normalized weighted speedups of the PCM scheme are 1.263, 1.376, and 1.482, respectively, on 4-core, 8-core, and 16-core systems. Mergeable obtains 1.372, 1.493, and 1.718 weighted speedups. *MERCURY* obtains 1.527, 1.634, and 1.928 weighted speedups, demonstrating better performance. With the increase of cores, *MERCURY* further exhibits its effectiveness and scalability since it leverages simple hashing to adapt to the workload changes.

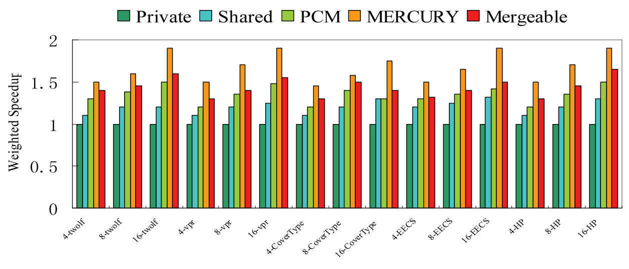


Fig. 7. Normalized weighted speedup.

5.2.3 Fair Speedup

Fair speedup computes the harmonic mean of the normalized IPCs, while taking into account both fairness and performance. Fig. 8 shows the results of comparing *MERCURY* with baseline schemes and PCM in terms of fair speedups. The fair speedups are normalized to those with the private cache.

Compared with the PCM scheme, Mergeable, and *MERCURY* improve the performance on this metric, respectively, by 7.16 and 8.35 percent (4-core), 8.31 and 9.52 percent (8-core), and 8.67 and 9.96 percent (16-core). The main reason is that *MERCURY* leverages the differentiated placement policy that efficiently allocates the data into the correlated caches and improves the utilization of the multicore processor based on the multitype memberships.

5.2.4 Cache Update Latency

Cache management needs to update stale and infrequently accessed data to guarantee high hit rates. We evaluate the update efficiency of private and shared, PCM, Mergeable, and *MERCURY* in terms of operation latency. Fig. 9 shows

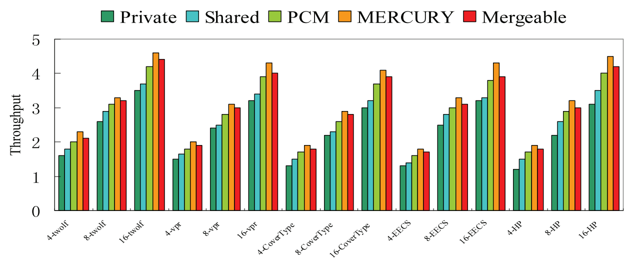


Fig. 6. Throughput (sum of IPCs).

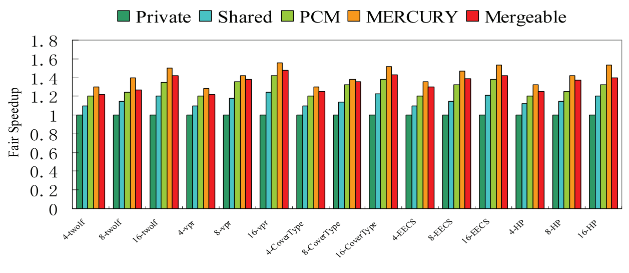


Fig. 8. Normalized fair speedup.

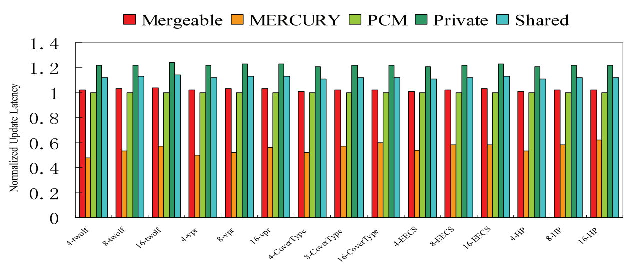


Fig. 9. Normalized update latency.

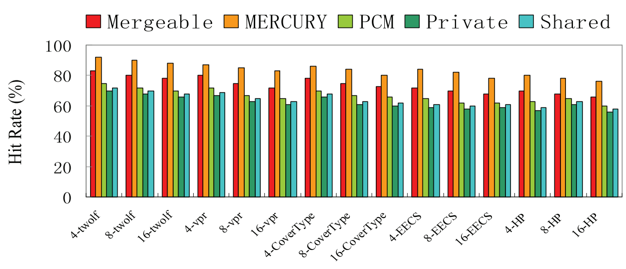


Fig. 11. Hit rates in multicore caches.

the update latencies that are normalized to those in the PCM scheme. We observe that the average normalized latencies of private and shared caches are 1.22 and 1.14, respectively. Mergeable requires a little larger latency than PCM mainly due to merged writebacks. Compared with the PCM and Mergeable schemes, *MERCURY* using the simple hash computation requires the smallest time than others and decreases the update latency on average by 48.26, 46.57, and 43.82 percent, respectively, on 4-core, 8-core, and 16-core systems.

5.2.5 Migration Cost

Hit misses or updates in caches often lead to data migration among multiple caches, which incurs relatively high costs in terms of data transmission and replacement in the caches of other cores. Fig. 10 shows the percentage of migrated data in PCM, Mergeable, and *MERCURY*. Mergeable is able to detect and merge similar data to guarantee that many correlated data are stored in a single cache, thus producing the smaller number of the migrated data than PCM.

We observe that the average percentages of migrated data are 13.2 and 11.9 percent, respectively, in private and shared caches. Compared with Mergeable, *MERCURY* can obtain better performance in this metric and decrease the number of migrated data on average by 35.26, 32.57, and 31.73 percent on 4-core, 8-core, and 16-core systems. The main reasons are twofold. One is that the MC-LSH provides high accuracy of identifying correlated data, thus reducing the number of migrated data. The other is that the fast identification of similar data in *MERCURY* produces low computation complexity.

5.2.6 Hit Rate

One of the key metrics to evaluate cache efficiency is the hit rate that defines the probability of obtaining queried data within limited cache space for requests. Fig. 11 shows the cache hit rate of the *MERCURY* scheme compared with private, shared, PCM and Mergeable. The average hit rates in private, shared, *MERCURY*, Mergeable, and PCM are respectively 65.22, 67.15, 86.92, 77.48, and 69.27 percent on the 4-core system, 61.68, 63.73, 83.87, 74.16, and 65.12 percent on the 8-core systems, and 59.51, 61.34, 81.73, 70.52, and 62.35 percent on the 16-core systems. *MERCURY* has the better performance in this metric than Mergeable and PCM since the MC-LSH accurately identifies correlated data within constant-scale execution complexity. The improved accuracy significantly decreases potential migration costs that possibly occur due to hit misses. The quick identification also alleviates the effects of staleness in the caches.

5.2.7 Time and Space Overheads

The execution time in our performance evaluation includes the identification and placement of the correlated data in the L1 caches. We evaluate the *MERCURY*, PCM, Mergeable, and standard LSH schemes in terms of time overhead as shown in Fig. 12a. The time overhead is normalized to those in Mergeable scheme. *MERCURY* makes use of hashing computation to identify correlated data, thus requiring smaller execution time than Mergeable that needs to carry out the extra operations of merging cache blocks. Compared with Mergeable, *MERCURY* decreases the execution time by 28.73, 21.84, and 19.56 percent on 4-core, 8-core, and 16-core systems.

MERCURY needs to use the signature vector to reveal the similarity of correlated data and leverage counting Bloom filters to maintain the memberships of cached data, while Mergeable requires temporary storage space, for example,

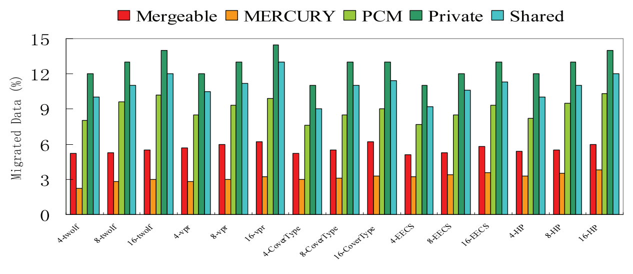
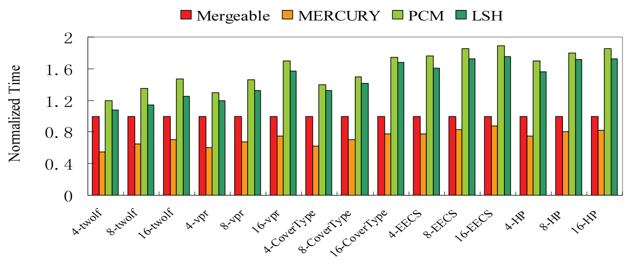
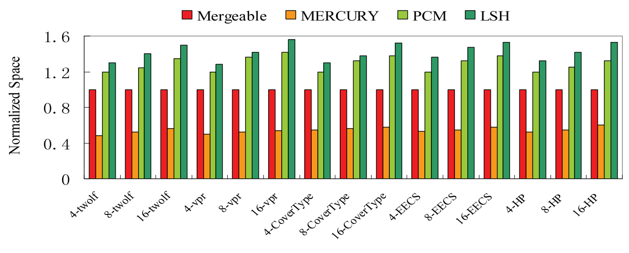


Fig. 10. Percentage of migrated data.

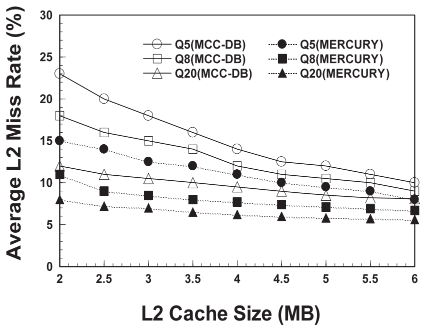


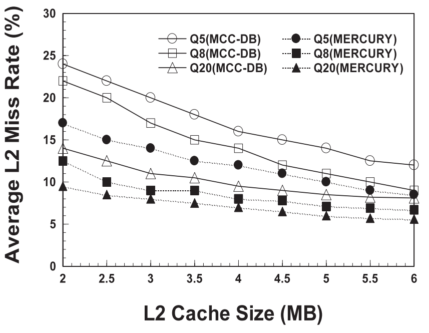
(a) Normalized time overhead.



(b) Normalized space overhead.

Fig. 12. Normalized time and space overheads.


(a) 500MB dataset.


(b) 1GB dataset.

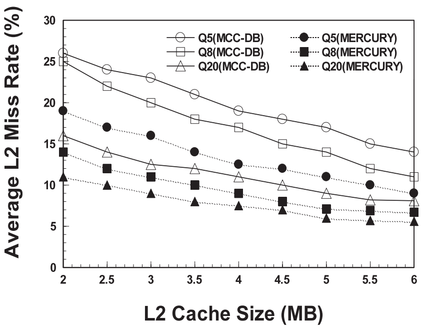
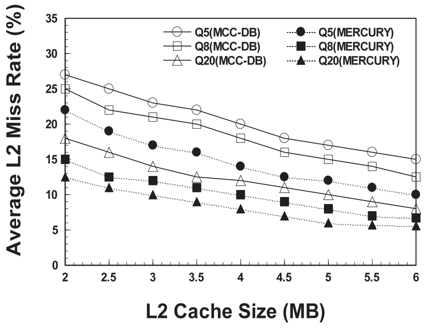
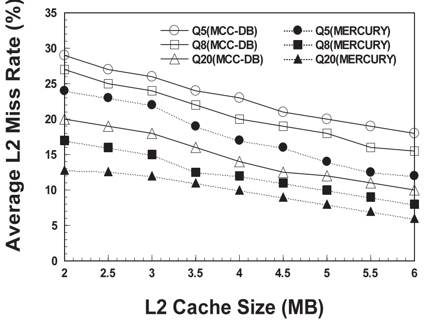

(c) 10GB dataset.

Fig. 13. L2 miss rates of TPC-H workload under three data set sizes.


(a) 1GB dataset.


(b) 5GB dataset.

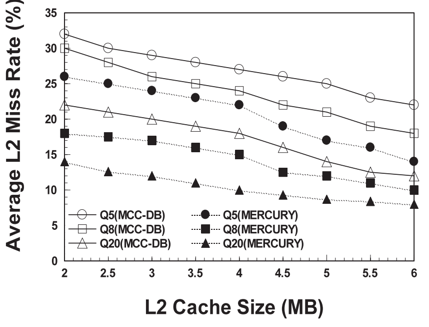

(c) 10GB dataset.

Fig. 14. L2 miss rates of TPC-C workload under three data set sizes.

content-addressable memory to find similar data. Fig. 12b shows the comparisons among the *MERCURY*, PCM, Mergeable, and standard LSH schemes in terms of space overhead. We observe that compared with the Mergeable scheme, *MERCURY* obtains significant space savings and decreases the space overhead by 47.35, 45.26, and 40.82 percent, respectively, on 4-core, 8-core, and 16-core systems. The main reason is that *MERCURY* leverages the simple bit-aware signature and space-efficient Bloom filters to demonstrate and maintain the memberships of correlated data, thus obtaining space savings.

6 System Implementation Study

We present the experimental results of running standard workloads provided by both TPC-H and TPC-C benchmarks [40]. Specifically, we set up 100 clients (i.e., the maximum number of clients allowed in PostgreSQL) to send out queries concurrently. For each client, queries are randomly drawn from the pool of all TPC-H queries. We repeat the same experiments under three different data set sizes: 500 MB, 1 GB, and 10 GB for TPC-H; and 1 GB, 5 GB, and 10 GB for TPC-C. Due to space limitation, the evaluation mainly shows the results of the TPC-H workload. We run all experiments in the cloud. Each cloud server has two Intel Core2Quad 2.66-GHz CPUs, 8-GB memory, and four 250-GB disks. Each processor has four cores, and every two cores share a 4-MB L2 cache. The DBMS used in our experiments is the PostgreSQL 8.3.1 running on Linux kernel 2.6.14.1. We measured the performance by three metrics, L2 cache miss rate, query execution time (cycles per instruction), and patching costs.

For a given data set, the hash functions come from the random selection of the LSH function family. More hash functions provide higher accuracy of identifying similar data, which however incurs higher computation complexity and space overhead. In our experiments, we select $L = 7$ hash functions based on the presample estimation, in which we randomly extract a subset from the used trace and make the estimation, which has been successfully used in the real-world applications [31], [35], [39], [41]. Moreover, for the data with different types or dimensionalities, we use the normalization method to compute data similarity in the same metric measure. Normalized value is equal to the measure: *(ActualValue-Minimum)/(Maximum-Minimum)*. For instance, for the attribute of file size, we assume that the size range is from 10 KB to 200 KB, (i.e., range: 10-200). For a file with 120 KB, its normalized value is $(120\text{-}10)/(200\text{-}10) = 0.58$.

We evaluate the real implementation performance by comparing with MCC-DB [17]. MCC-DB exploits access and caching patterns from query analysis. The reason for making this comparison is threefold. First, both MCC-DB and MERCURY work well as patches to PostgreSQL [16] for concurrent queries. Second, the essential property behind two techniques is to use cache partitioning in multi-core processors to enhance system performance. Third, MCC-DB has provided standard experimental results by using TPC-H [40] to facilitate fair comparisons with other methods. TPC-H [40] benchmarks have large volumes of data for decision support systems when executing 22 different types of queries. We perform extensive experiments on a physical testbed based on the PostgreSQL system using the workloads generated from the TPC benchmarks.

6.1 L2 Cache Miss Rate

Figs. 13 and 14, respectively, show the L2 miss rates when using TPC-H and TPC-C workloads with different data set sizes. We first examine the rates of three typical queries, i.e.,

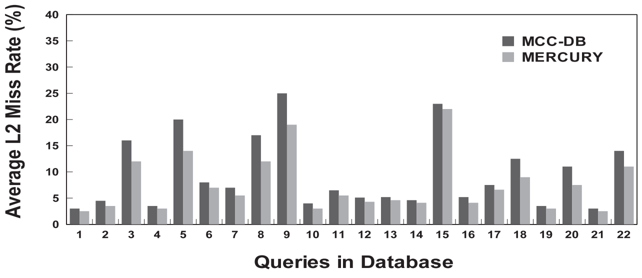
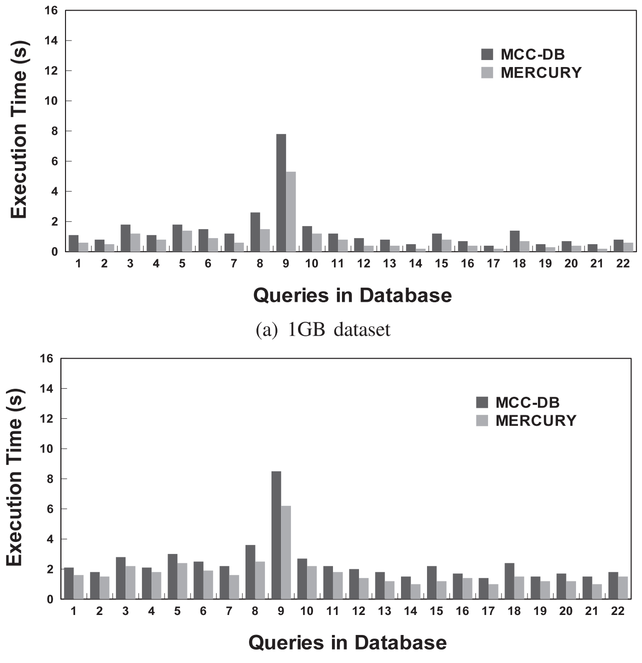


Fig. 15. L2 miss rates in 10-GB data set with 4-MB L2 cache.



(a) 1GB dataset

(b) 10GB dataset

Fig. 16. Execution time of queries.

Q5, Q8, and Q20, in both MCC-DB and *MERCURY* schemes. Q5 and Q8 are dominated by multiway hash joins and Q20 is dominated by nested subquery executions. We observe that *MERCURY* obtains on average 42.6, 48.1, and 51.2 percent miss decrease compared with MCC-DB in the TPC-H workload with 500 MB, 1 GMB, and 10 GB sizes. These benefits come from the fast and accurate hashing-based computation to identify similar data to efficiently support concurrent queries. We further examine the L2 miss rates by executing all 22 queries in TPC-H. Compared with MCC-DB, *MERCURY* has the decrease of miss rates from 16.1 to 26.7 percent, on average 21.8 percent for all 22 queries as shown in Fig. 15. In addition, it is also observed that Q1, Q4, Q14, and Q21 show the comparable values with MCC-DB due to their relatively weak locality characteristic.

6.2 Query Execution Time

Shared L2 cache plays a key role in determining query execution performance. When concurrent query execution processes access common tuples and index data, *MERCURY* enables multiple cores for data sharing to reduce unnecessary memory accesses. In *MERCURY*, a query execution process reuses cache lines. We evaluate query execution time by using cycles per instruction, which is examined by using the performance tool to check hardware counters. Figs. 16a and 16b, respectively, show the executing time of queries in 1 GB and 10 GB data set sizes. Due to different functionalities of all 22 queries in TPC-H, the execution time shows significant fluctuation. *MERCURY* also obtains on average 21.7 and 23.1 percent improvements upon MCC-DB in 1-GB and 10-GB sizes.

TABLE 3
Normalized Time and Space Costs as a Patch

		TPC-C			TPC-H		
		1GB	5GB	10GB	500MB	1GB	10GB
Time	MCC-DB	1	1	1	1	1	1
	MERCURY	0.78	0.76	0.71	0.75	0.71	0.65
Space	MCC-DB	1	1	1	1	1	1
	MERCURY	0.86	0.83	0.81	0.84	0.82	0.79

6.3 Patching Cost

Both MCC-DB and *MERCURY* are implemented as a patch in PostgreSQL. The new component may introduce extra patching costs. We examine these costs in terms of time and space as shown in Table 3 by being normalized to MCC-DB when taking into account both TPC-C and TPC-H data sets under different sizes. *MERCURY* requires less time and space costs than MCC-DB because *MERCURY* makes use of the fast MC-LSH hashing computation to place similar data together. We also observe that with the increase of data set sizes, *MERCURY* obtains more benefits in terms of time and space overheads over MCC-DB. In the meantime, this observation demonstrates the scalability of the *MERCURY* scheme.

7 Related Work

Multilevel cache hierarchy has been studied in the high-performance cloud architecture and software communities. There exists a wide range of proposals to improve caching performance (e.g., hit rate, access latency, and space overhead) [11], [19], [20], [42], [43], [44]. We argue that suitable management of the multilevel cache hierarchy is becoming more important to deliver high performance in the cloud.

Locality-based optimization. The state-of-the-art work, R-NUCA [45], obtains near-optimal cache block placement by classifying blocks online and placing data close to the core. To mitigate the loss of reusing cached states when rescheduling a process, affinity scheduling [46] helps reduce cache misses by judiciously scheduling a process on a recently used CPU. To improve the performance in "multiexecution" applications, *Mergeable* [7] captures data similarities and merges duplicate cache lines owned by different processes to obtain substantial capacity savings. Nevertheless, performing the explicitly merging operations on cache blocks demands relatively longer execution time and increases computation complexity. The process-level cache management policy (PCM) [12] has the assumption that all memory regions belonging to a running process exhibit the same access pattern. MCC-DB [17] makes use of different locality strengths and query execution patterns to minimize cache conflicts. This improvement works under the assumption when there are multiple candidate plans that are accurately estimated in advance. However, this assumption does not always hold in many practical applications because performing accurate estimate generally requires high-computation overheads. Unlike them, *MERCURY* explores and exploits the locality property by lightweight hashing approach, thus obtaining significant performance improvements.

Hardware acceleration. To reduce the cache pollution caused by LRU that inserts non-reusable items into the cache while evicting reusable ones, ROCS [6] employs hardware counters to characterize cache behaviors and introduces a pollute buffer to host non-reused cache lines of pages before eviction. Moreover, to address the problems of increased capacity interference and longer L2 access latency, CloudCache [47] leverages fine-grained hardware monitoring and control to dynamically expand and shrink L2 caches for working threads by using dynamic global partitioning, distance-aware data placement, and limited target broadcast. Hardware-assisted execution throttling [48] helps regulate fairness in modern multicore processors, while demonstrating the relative benefits of the various resource control mechanisms. Moreover, to reduce the large amounts of misses in the LLC between the eviction of a block and its reuse, Scavenger [49] divides the total cache storage into a conventional cache and a victim file architecture to identify and retain high-priority cache blocks that are more likely to be reused. *MERCURY* bridges the gap between a multicore architecture and an operating systems. The existing hardware acceleration approaches can use *MERCURY* to simplify operations and optimize system calls.

Operations enhancements. MergeSort [50] performed an efficient multiway merge without being constrained by the memory bandwidth for high-throughput database applications. The parallel skyline computation could benefit from multicore architectures, such as parallel version of the branch-and-bound algorithm. Park et al. [51] presented a parallel algorithm based on parallel programming that was evaluated as a case study of parallelizing database operations. A cooperation-based locking paradigm [52] was proposed for efficient parallelization of frequency counting and top-k over multiple streams in the context of multicore processors. In addition, adaptive aggregation [53] demonstrated that a chip multiprocessor with new dimensions could enhance concurrent sharing of aggregation data structures and contentious accesses to frequently used values. Qiao et al. [54] introduced a scheduling technique to cooperate multiple memory scans to reduce the overhead on memory bandwidth. These research projects aim to make a single query benefit from the cache, which is orthogonal to our work. Mining locality can improve parallel queries in multicore CPU [55] and tree-structured data [56]. Two popular join algorithms, such as hash join and sort-merge join, was re-examined in [57] to use multicore cache blocking to minimize access latency, increase compute density and balance load amongst cores, even for heavily skewed input data sets. CATCH [58] can store unique contents in instruction cache by means of hashing, but their proposed system does not support modifications in cached data. In addition, the cache compression technique [59] compresses the L2 data results to reduce the cache space and the off-chip accesses, thus obtaining bandwidth savings. The cooperative caching technique [60] in a multiprocessor can reduce off-chip access through using a cooperative private cache either by storing a single copy of clean blocks or providing a cache-like, spill-over memory for storing evicted cache lines. *MERCURY* can improve the query performance by using locality-aware data placement strategy.

Workloads awareness. An OS-based cache partitioning mechanism [29] presents execution and measure-based strategies for multicore cache partitioning upon multiple representative workloads. A nonuniform cache architecture (NUCA) [61] takes advantage of proximity of data from the accessing processor. To further address the problem of onchip data locality in large shared NUCA, PageNUCA [8] proposed a fully hardwired coarse-grain data migration mechanism that dynamically monitored the access patterns of the cores at the granularity of a page. Subsequently, the NuRAPID proposal [62] decoupled the tag and data placement in a NUCA by augmenting each tag and data block with a forward and reverse pointer to the corresponding data block and tag, respectively. NUcache [63] makes use of the DelinquentPC-Next-Use characteristic to improve the performance of shared caches in multi-cores. The NUcache organization logically partitions the associative ways of a cache set into MainWays and DeliWays. *MERCURY* is orthogonal to the existing schemes. It leverages the lightweight LSH-based computation and obtains significant performance improvements on the LLC by accurately capturing the differentiated locality across data.

Scheduling. Age-based scheduling for heterogeneous multiprocessor [64] allows a thread with the larger remaining execution time to run in a faster core given the prediction of remaining execution time. A thread-based preloading technique for simultaneous multithreading processors was proposed in [65] to use the *helper thread* to perform aggressive data preloading. To improve the utilization of on-chip memory and reduce the impact of expensive DRAM and remote cache accesses, O^2 scheduling [66] schedules objects and operations to caches and cores. To decrease the unnecessary sharing of network control state at all stack layers, the IsoStack architecture [67] offloads network stack processing to a dedicated processor core. Moreover, integrated processor-cache partitioning [9] divides both the available processors and the shared cache in a chip multiprocessor among different multithreaded applications. The existing scheduling strategies can further help optimize *MERCURY* performance.

8 Conclusion

MERCURY, as an infrastructure of the cloud, plays a significant role in managing the multilevel cache hierarchy. By exploring and exploiting data similarity that is derived from locality-aware access patterns, *MERCURY* alleviates homogeneous data placement and improves system performance by the low-complexity MC-LSH computation. The cost-effective *MERCURY* is able to provide hybrid functionalities. One is to provide a lightweight mechanism for allocating cache resources. The other is to support the OS-based dynamic cache allocation and capture data similarity with the aid of space-efficient structures. *MERCURY*, hence, allows the OS control over the shared LLCs, while minimizing software overheads. Experiments using the real-world data sets demonstrate the *MERCURY's* efficiency.

Since modern microprocessors increasingly incorporate multiple memory controllers [68], [69], our future work will consider the locality-aware schemes for implementing efficient data placement in multiple memory controllers.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 61173043; National Basic Research 973 Program of China under Grant 2011CB302301; NSFC under Grant 61025008, 61232004; Fundamental Research Funds for the central universities, HUST, under grant 2012QN098; the NSERC Discovery Grant 341823; US National Science Foundation Award 1116606. This is an extended version of our paper published in the Proceedings of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2012, pages: 371-378.

REFERENCES

[1] J. Gantz and D. Reinsel, "Digital Universe Study: Extracting Value from Chaos," *Proc. Int'l Data Corporation (IDC),* June 2011.
[2] Science Staff, "Dealing with Data—Challenges and Opportunities," *Science,* vol. 331, no. 6018, pp. 692-693, 2011.
[3] M. Armbrust et al., "A View of Cloud Computing," *Comm. ACM,* vol. 53, no. 4, pp. 50-58, 2010.
[4] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud Computing for Everyone," *Proc. ACM Symp. Cloud Computing (SOCC),* 2011.
[5] S. Wu, F. Li, S. Mehrotra, and B. Ooi, "Query Optimization for Massively Parallel Data Processing," *Proc. ACM Symp. Cloud Computing (SOCC),* 2011.
[6] L. Soares, D. Tam, and M. Stumm, "Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-Only Pollute Buffer," *Proc. IEEE/ACM 41st Ann. Int'l Symp. Microarchitecture (MICRO),* pp. 258-269, 2009.
[7] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. Chong, "Multi-Execution: Multicore Caching for Data-Similar Executions," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA),* 2009.
[8] M. Chaudhuri, "PageNUCA: Selected Policies for Page-Grain Locality Management in Large Shared Chip-Multiprocessor Caches," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 227-238, 2009.
[9] S. Srikantaiah, R. Das, A.K. Mishra, C.R. Das, and M. Kandemir, "A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors," *Proc. Conf. High Performance Computing Networking, Storage, and Analysis (SC),* 2009.
[10] X. Ding, K. Wang, and X. Zhang, "SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores," *Proc. Sixth Conf. Computer Systems (EuroSys),* 2011.
[11] Y. Chen, S. Byna, and X. Sun, "Data Access History Cache and Associated Data Prefetching Mechanisms," *Proc. IEEE/ACM Conf. Supercomputing (SC),* 2007.
[12] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Enabling Software Management for Multicore Caches with a Lightweight Hardware Support," *Proc. Conf. High Performance Computing Networking, Storage, and Analysis (SC),* 2009.
[13] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John, "The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies," *Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA),* 2010.
[14] A. Forin, B. Neekzad, and N. Lynch, "Giano: The Two-Headed System Simulator," Technical Report MSR-TR-2006-130, Microsoft Research, 2006.
[15] S. Biswas, D. Franklin, T. Sherwood, and F. Chong, "Conflict-Avoidance in Multicore Caching for Data-Similar Executions," *Proc. 10th Int'l Symp. Pervasive Systems, Algorithms, and Networks (ISPAN),* 2009.
[16] "PostgreSQL," http://www.postgresql.org/, 2013.
[17] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: Minimizing Cache Conflicts in Multi-Core Processors for Databases," *Proc. VLDB Endowment,* vol. 2, no. 1, pp. 373-384, 2009.
[18] T.R.B. Bershad, D. Lee, and B. Chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 1994.
[19] Y. Yan, X. Zhang, and Z. Zhang, "Cacheminer: A Runtime Approach to Exploit Cache Locality on SMP," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 4, pp. 357-374, Apr. 2000.
[20] K. Zhang, Z. Wang, Y. Chen, H. Zhu, and X. Sun, "PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers," *Proc. IEEE/ACM 11th Int'l Symp. Cluster, Cloud, and Grid Computing (CCGrid),* pp. 265-274, 2011.
[21] G. Suh, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," *Proc. ACM 15th Int'l Conf. Supercomputing (ICS),* 2001.
[22] "UCI Machine Learning Repository," *The Forest CoverType Data Set,* http://archive.ics.uci.edu/ml/data sets/Covertype, 2013.
[23] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Proc. Second USENIX Conf. File and Storage Technologies (FAST),* 2003.
[24] E. Riedel, M. Kallahalla, and R. Swaminathan, "A Framework for Evaluating Storage System Security," *Proc. Conf. File and Storage Technologies (FAST),* 2002.
[25] SPEC2000, http://www.spec.org/cpu2000/, 2013.
[26] S. Carr and K. Kennedy, "Compiler Blockability of Numerical Algorithms," *Proc. Supercomputing Conf.,* 1992.
[27] E.E.R.M.S. Lam and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 1991.
[28] M.S.L.T.C. Mowry and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 1992.
[29] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," *Proc. IEEE 14th Int'l Symp. High Performance Computer Architecture (HPCA),* 2008.
[30] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Toward Removing the Curse of Dimensionality," *Proc. 13th Ann. ACM Symp. Theory of Computing (STOC),* 1998.
[31] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB),* pp. 950-961, 2007.
[32] R. Shinde, A. Goel, P. Gupta, and D. Dutta, "Similarity Search and Locality Sensitive Hashing Using Ternary Content Addressable Memories," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD),* pp. 375-386, 2010.
[33] A. Joly and O. Buisson, "A Posteriori Multi-Probe Locality Sensitive Hashing," *Proc. ACM Int'l Conf. Multimedia,* 2008.
[34] G. Taylor, P. Davies, and M. Farmwald, "The TLB Slice-a Low-Cost High-Speed Address Translation Mechanism," *Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA),* 1990.
[35] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *Comm. ACM,* vol. 51, no. 1, pp. 117-122, 2008.
[36] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.
[37] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.
[38] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and Efficiency in High-Dimensional Nearest Neighbor Search," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD),* 2009.
[39] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for Similarity Search in Peer-to-Peer File Systems," *Proc. 37th Int'l Conf. Parallel Processing (ICPP),* pp. 644-651, 2008.
[40] TPC, http://www.tpc.org/, 2013.
[41] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query," *IEEE Trans. Computers,* vol. 61, no. 6, pp. 817-830, June 2012.
[42] Z. Zhang, Z. Zhu, and X. Zhang, "Cached Dram for ILP Processor Memory Access Latency Reduction," *IEEE Micro,* vol. 21, no. 4, pp. 22-32, July 2001.
[43] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," *Proc. IEEE/ACM Conf. Supercomputing (SC),* 2008.
[44] Z. Zhang, Z. Zhu, and X. Zhang, "Design and Optimization of Large Size and Low Overhead Off-Chip Caches," *IEEE Trans. Computers,* vol. 53, no. 7, pp. 843-855, July 2004.

[45] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Near-Optimal Cache Block Placement with Reactive Nonuniform Cache Architectures," *IEEE Micro*, vol. 30, no. 1, pp. 20-28, Jan./Feb. 2010.

[46] J. Torrellas, A. Tucker, and A. Gupta, "Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors: A Summary," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1993.

[47] H. Lee, S. Cho, and B. Childers, "Cloudcache: Expanding and Shrinking Private Caches," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 219-230, 2011.

[48] X. Zhang, S. Dwarkadas, and K. Shen, "Hardware Execution Throttling for Multi-core Resource Management," *Proc. USENIX Ann. Technical Conf.*, 2009.

[49] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A New Last Level Cache Architecture with Global Block Priority," *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 421-432, 2007.

[50] J. Chhugani, A. Nguyen, V. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture," *Proc. VLDB Endowment*, vol. 1, pp. 1313-1324, 2008.

[51] S. Park, T. Kim, J. Park, J. Kim, and H. Im, "Parallel Skyline Computation on Multicore Architectures," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE)*, 2009.

[52] S. Das, S. Antony, D. Agrawal, and A.E. Abbadi, "Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams," *Proc. VLDB Endowment*, vol. 2, pp. 217-228, 2009.

[53] J. Cieslewicz and K. Ross, "Adaptive Aggregation on Chip Multiprocessors," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB)*, 2007.

[54] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman, "Main-Memory Scan Sharing for Multi-Core CPUs," *Proc. VLDB Endowment*, vol. 1, pp. 610-621, 2008.

[55] W. Han and J. Lee, "Dependency-Aware Reordering for Parallelizing Query Optimization in Multi-Core CPUs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[56] S. Tatikonda and S. Parthasarathy, "Mining Tree-Structured Data on Multicore Systems," *Proc. VLDB Endowment*, vol. 2, pp. 694-705, 2009.

[57] C. Kim, T. Kaldewey, V. Lee, E. Sedlar, A. Nguyen, N. Satish, J. Chhugani, A.D. Blas, and P. Dubey, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs," *Proc. VLDB Endowment*, vol. 2, pp. 1378-1389, 2009.

[58] M. Kleanthous and Y. Sazeides, "CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication and Its Application to Instruction Caches," *Proc. Conf. Design, Automation, and Test in Europe (DATE)*, 2008.

[59] A. Alameldeen and D. Wood, "Adaptive Cache Compression for High-Performance Processors," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA)*, 2004.

[60] J. Chang and G. Sohi, "Cooperative Caching for Chip Multiprocessors," *Proc. 33rd Ann. Int'l Symp. Computer Architecture (ISCA)*, 2006.

[61] C. Kim, D. Burger, and S. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated on-Chip Caches," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[62] Z. Chishti, M. Powell, and T. Vijaykumar, "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures," *Proc. IEEE/ACM 36th Ann. Int'l Symp. Microarchitecture (MICRO)*, 2003.

[63] R. Manikantan, K. Rajan, and R. Govindarajan, "NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 243-253, 2011.

[64] N. Lakshminarayana, J. Lee, and H. Kim, "Age Based Scheduling for Asymmetric Multiprocessors," *Proc. IEEE/ACM Supercomputing Conf.*, 2009.

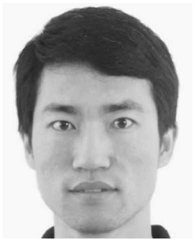
[65] J. Zhou, J. Cieslewicz, K. Ross, and M. Shah, "Improving Database Performance on Simultaneous Multithreading Processors," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB)*, 2005.

[66] S. Boyd-Wickizer, R. Morris, and M.F. Kaashoek, "Reinventing Scheduling for Multicore Systems," *Proc. 12th Conf. Hot Topics in Operating Systems (HotOS)*, 2009.

[67] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, "IsoStack: Highly Efficient Network Processing on Dedicated Cores," *Proc. USENIX Ann. Technical Conf.*, 2010.

[68] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Managing Data Placement in Memory Systems with Multiple Memory Controllers," *Int'l J. Parallel Programming*, vol. 40, no. 1, pp. 57-83, 2012.

[69] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2010.



Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is an associate professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 40 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, USENIX ATC, INFOCOM, SC, ICDCS, ICPP, and MASCOTS. He has been on the program committees of multiple international conferences, including INFOCOM and ICPP. He is a senior member of the IEEE, and a Member of ACM and USENIX.

Xue Liu received the BS and MS degrees in mathematics and automatic control, respectively, from Tsinghua University, China, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2006. He is an associate professor in the School of Computer Science at McGill University. His research interests include computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. He serves as an editor for *IEEE Transactions on Vehicular Technology*, an associate editor for the *IEEE Transactions on Parallel and Distributed Systems*, and an editor for the *IEEE Communications Surveys and Tutorials*. His work has received the Year 2008 Best Paper Award from *IEEE Transactions on Industrial Informatics*, and the First Place Best Paper Award of the ACM Conference on Wireless Network Security (WiSec 2011). He is a member of the IEEE and the ACM.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications to her credit in journals and international conferences, including *IEEE Transactions Parallel and Distributed Systems*, JCST, USENIX ATC, FAST, ICDCS, HPDC, SC, ICS, and ICPP. She serves as the program committee member of multiple international conferences, including SC and MSST. She is a member of the IEEE.