Abstract—The cloud is emerging for scalable and efficient cloud services. In order 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 multi-level 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. In order to accurately and efficiently capture the data similarity, we leverage 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 Multi-Core-enabled LSH (MC-LSH) that accurately captures the differentiated similarity across data. The similarity-aware MERCURY hence partitions data into 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 datasets demonstrate the efficiency and efficacy of our proposed schemes.

Keywords—Cloud computing; multi-core processor; cache management; data similarity.

1 INTRODUCTION

We are entering the era of the cloud that contains massive and heterogeneous data. The datasets have the salient feature of a volume of Petabytes or Exabytes and data streams with a speed of Gigabits per second. These datasets often have to be processed and analyzed in a timely fashion. According to a recent International Data Corporation (IDC) study, there exists more than 1.8 Zettabytes information created and replicated in 2011 [1]. Science poll [2] shows that about 20% respondents often use more than 100GB datasets. Moreover, from small hand-held devices to huge data centers, we are collecting and analyzing massive information. Commercial companies, like Google, Microsoft, Yahoo! and Facebook, process large amounts of data everyday [3]–[5]. For the computation infrastructure of the cloud, it is important and challenging to perform efficient processing and analysis for these data.

The computation infrastructure typically consists of multi-core 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]–[9]. Moreover, prevalent CMPs are widely used in cloud applications. In practice, it is still a daunting task to accurately and efficiently perform the multi-core caching for high performance cloud systems. This paper is to optimize the data placement of the multi-level 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. In order 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 multi-level cache hierarchy. Since the CPU caches are at the hardware level while the buffer cache is a part of 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 multi-core amounts and increasingly large capacity of main memory, severe performance degradation may occur once the inconsistency gap exists. These two layers hence become inefficient to work cooperatively. Moreover, by leveraging a shared cache, a thread, which cooperatively works with multiple co-running 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. Multi-core 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, existing policies in the multi-core processors become neither efficient nor scalable to address the data placement problem. In order 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 LLC Pollution.** Last Level Cache (LLC) [6], [13], [14] 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. In order 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], [15]. 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], [15]. 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 multi-level cache hierarchy. We implement MERCURY and manage the similarity at a granularity of pages by leveraging operating system mechanisms. MERCURY is compatible with 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 multi-level cache hierarchy. Specifically, we make the following contributions.

**First,** (for Challenge 1), in order to narrow the inconsistency gap and quantify the data correlation, MERCURY employs multi-type, rather than conventional homogeneous, membership management. Here, the membership refers that an item belongs to a given dataset. The data in the similarity-aware multi-core 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 Challenge 2 & 3), in order to address the performance bottleneck and alleviate the LLC pollution, MERCURY explores and exploits the access locality by using a Multi-Core-enabled Locality-Sensitive Hashing (MC-LSH). MC-LSH uses a self-contained and space-efficient signature vector, rather than many hash tables in a standard LSH [16], 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 existing hardware devices. In order to further examine and evaluate the efficacy and efficiency of the proposed scheme, we not only examine MERCURY in a multi-core simulation [7], [17], [18], but also implement it in a system by patching PostgreSQL [19]. The extensive experiments use real-world traces and datasets, and examine the performance in multiple evaluation metrics.

The remainder of this paper is organized as follows. Section 2 presents the datasets analysis and problem statement. Section 3 describes the proposed MERCURY architecture and caching schemes. Section 4 shows the cached data management schemes in the multi-level hierarchy. Section 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 Datasets Analysis and 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], [20] [24]. Main benefits include throughput improvements and the reduction of LLC cache miss rates, query latency and data migration overheads. Hence, the motivation of MERCURY design comes from the observations of data similarity widely existing in 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 $a_d$ and $b_d$. If the geometric distance between vectors $a_d$ and $b_d$ is
smaller than a pre-defined threshold, they are similar. The data similarity often hides behind the locality of access patterns [8].

We study typical large-scale applications [25]–[27] and the main benchmarks from the SPEC2000 evaluation [28], i.e., vpr and twolf. The properties of used traces and datasets are listed. (1): CoverType dataset [25] contains 581,012 data points, each of which has 54-dimensional attributes. (2): EECS NFS server at Harvard [26] collected I/O accesses. This dataset contains concurrent requests with a total of 4.4 millions operations. (3): HP file system provides a 10-day 500GB trace [27] that records the accesses from 236 users. (4): vpr & twolf benchmarks show the CPU performance in the SPEC2000 evaluation [28]. vpr leverages combinatorial optimization technique to automatically synthesize the mapped circuits. twolf makes use of TimberWolfSC placement and global routing package.

In order 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 dependencies 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. 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 2000 million accesses.

![Fig. 1. Average locality ratios from real-world datasets.](image)

We use Locality Ratio as a measure to represent the locality in the access pattern. Figure 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 dataset. The time interval comes from the used traces, in which all accesses are listed in the order of time. For instance, a dataset contains a 20-hour trace record and we select a 25% interval, i.e., 5-hour access record. For a file, if it has 8 accesses within a randomly selected 25% 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% interval), the locality ratio becomes 8/8=100%. We observe that there exists 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 Problem Statement and Basic Idea

The hardware design of cloud computation infrastructure still work for scenarios they are designed for, but the lack of flexibility can be an unavoidable issue and inherent weakness, particularly for multi-core or many-core processors with an increasingly large number of cores. Cache optimization and cache resource management at different levels of software, such as operating systems, compilers, and application programs have shown their efficiency and 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 policy [21], [29]–[31]. 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 multi-threads. Researchers have evaluated OS-based cache partitioning methods in multicore processors [32]. 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, OS-based methods can inevitably cause non-trivial software overhead, and are not ready to be used as computation infrastructure of the cloud. In order to offer efficient computation infrastructure for the cloud, we investigate the problem of the cached data placement in the multi-level cache hierarchy when executing multiple parallel instances as shown in Figure 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. 2. Problem description.](image)
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 multi-core 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], [20], which unfortunately is non-trivial due to expensive operation costs of comparing arriving data with all existing cache lines.

We identify the problem of homogeneous data placement that overlooks the distinct properties and multi-type memberships of cached data. In order 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 Figure 2(b). 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 time-consuming and computation-intensive work due to high dimensions and heterogeneous types. Hence, in order to accomplish a suitable tradeoff between similarity accuracy and operation complexity, we propose to use a hash-based approach, e.g., Locality Sensitive Hashing (LSH) [16], due to its locality-aware property and ease of use. 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. 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 [33]–[35]. 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 Multi-Core-enabled LSH (MC-LSH) to identify similar data and carry out differentiated data placement. MERCURY represents the strength of data similarity respectively as Family, Friend and Foreigner as shown in Figure 2(b). 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 similar data that are placed closely can also significantly reduce the migration costs. MERCURY hence offers the scalable, flexible and load-balanced caching schemes in a multi-level cache hierarchy.

MERCURY is implemented in a hybrid scheme to address the limitations of both hardware solutions and OS-based methods. Specifically, our multicores 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 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 multi-core systems.

3 MERCURY Architecture

MERCURY uses MC-LSH to identify similar data and leverages an LRU replacement in each cache to update stale data. Figure 3 shows the MERCURY architecture in the multi-level 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, in order to explicitly represent the differentiated memberships identified by MC-LSH, we use different flags to label each cache line and obtain holistic optimization in the multi-level cache hierarchy.

3.1 Caches in a Multi-core Processor

The caching schemes in a multi-core 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: In order to partition the shared L2 cache, we leverage the well-known page color [36] 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 and the physical page number, which is indicated as a page color. One can divide a physically addressed cache into non-intersecting 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 identify the similar data by using the results from the MC-LSH computation.

### 3.2 Operating System

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 multi-core caching. Specifically, MC-LSH uses a space-efficient signature vector to maintain the cached data and utilizes a coding technique to support differentiated placement policy for the multi-type data. We will describe the design details of MC-LSH in Section 4.

**Updating Locality-aware Data:** In order 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 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 multi-core computation and efficiently update data. Since the cached data are locality-aware, **MERCURY** hence decreases migration costs and minimizes cache conflicts.

In order to satisfy query requests and provide flexible use, we design an interface between high-performance applications and operating system as shown in Figure 3. Its main function is to wrap high-level operation requests to low-level system calls with the aid of the page coloring technique [36]. 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 operating system can't 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. In order 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

In order 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 multi-type memberships. We finally describe the scheme of updating data in **MERCURY**.

#### 4.1 The MC-LSH Scheme

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

**Limitations of Standard LSH.** An LSH [16] 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., \( H = \{ h : S \rightarrow U \} \) is called \( (R,c,R,p_1,p_2)-\text{sensitive} \), if for \( \forall p,q \in S \):
- If \( ||p,q|| \leq R \) then \( Pr_{LB}[h(p) = h(q)] \geq p_1 \)
- If \( ||p,q|| > cR \) then \( Pr_{LB}[h(p) = h(q)] \leq p_2 \)

where \( c > 1 \) and \( p_1 > p_2 \).

In \( H \), \( h_{a,b}(v) = \frac{a\cdot v+b}{a+1} \). \( a \) 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 \( \omega \) is a constant. By using the LSH functions, similar data have a higher probability of colliding than the data that are far apart [37].

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

**Space-efficient Signature Vector.** 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 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 Figure 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 pre-defined 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 large 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%, 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. In order 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 multi-probe LSH [33].

In order 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 [38]. 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.

**Multi-type Membership Coding.** The memberships in MC-LSH include Family, Friend and Foreigner, which respectively represent different similarities among cached data. 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. In order to address this problem, we use a coding technique to guarantee membership consistency and integrity.

![Fig. 4. Signature vector for maintaining page-level data similarity.](image)

![Fig. 5. Differentiated membership coding technique.](image)
4.2 Updating Data

In the multi-level hierarchy of MERCURY, we need to update cached data and their memberships in the signature vector.

**For updating cached data.** In order to update actual data, we make use of a label-based technique to update stale data in multi-level caches. The reason comes from the fact that similar data are potentially re-used by corresponding caches in the near future. In order to decrease re-caching 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 [38]. 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 multi-level 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 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.** In order 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 simultaneously 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 method [33], [37], [40], [41] to obtain the suitable size.

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 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 multi-core simulation [7], [17], [18]. 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 Section 3 and 4. The size of each OS page is 8KB. 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.

MERCURY leverages MC-LSH to identify similar data that are respectively placed into L1 and L2 caches 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 8 colors and each color has 128 cache sets. We hence allocate 1MB cache for 4-core system, 2MB cache for 8-core system, and 4MB cache for 16-core system. Table 1 shows the parameter settings in the simulations.

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

The used traces and datasets include Forest CoverType dataset [25], EECS NFS server at Harvard [26], HP file system trace [27], and vpr and twolf in SPEC2000 [28]. Moreover, by using the proposed sampling approach [33], [37], [40], [41] described in Section 4.2, the suitable sizes of signature vectors are 7.6KB in vpr, 7.9KB in twolf, 8.3KB in CoverType, 8.7KB in EECS, and 9.2KB in HP.

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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>$\sum_{i=1}^{n} \frac{IPC_{\text{scheme}}(i)}{IPC_{\text{base}}(i)}$</td>
</tr>
<tr>
<td>Weighted Speedup</td>
<td>$\sum_{i=1}^{n} \frac{IPC_{\text{scheme}}(i)}{IPC_{\text{base}}(i)}$</td>
</tr>
<tr>
<td>Fair Speedup</td>
<td>$n/ \sum_{i=1}^{n} \frac{IPC_{\text{base}}(i)}{IPC_{\text{scheme}}(i)}$</td>
</tr>
</tbody>
</table>

### 5.2 Results

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

#### 5.2.1 Throughput

Figure 6 shows the throughput results from executing real-world applications with the increase of multi-core number from 4 to 16. The average throughputs on 4-core systems with private cache, shared cache, PCM, Mergeable and **MERCURY** are respectively 1.352, 1.563, 1.815, 1.925 and 2.162. 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% increase than other applications. The main reason is that the SPEC2000 benchmarks have better similarity in the access pattern, thus allowing 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. 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% respectively on 4-core, 8-core and 16-core systems.

#### 5.2.3 Fair Speedup

Fair speedup computes the harmonic mean of the normalized IPCs while taking into account both fairness and performance. Figure 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% (4-core), 8.31% and 9.52% (8-core), and 8.67% and 9.96% (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 multi-core processor based on the multi-type 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. Figure 9 shows 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 respectively 1.22 and 1.14. Mengeable requires a little larger latency than PCM mainly due to merged writebacks. Compared with PCM and Mengeable schemes, MERCURY using simple hash computation requires the smallest time than others and decreases the update latency on average by 48.26%, 46.57% and 43.82% respectively on 4-core, 8-core and 16-core systems.

We observe that the average percentages of migrated data are 13.2% and 11.9% respectively in private and shared caches. Compared with Mengeable, 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% on 4-core, 8-core and 16-core systems. The main reasons are twofold. One is that MC-LSH provides high accuracy of identifying correlated data, thus reducing the number of migrated data on average by 48.26%, 46.57% and 43.82% 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 re-placement in the caches of other cores. Figure 10 shows the percentage of migrated data in PCM, Mengeable and MERCURY. Mengeable 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% respectively in private and shared caches. Compared with Mengeable, 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% on 4-core, 8-core and 16-core systems. The main reasons are twofold. One is that 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 key metrics to evaluate cache efficiency is the hit rate that defines the probability of obtaining queried data within limited cache space for requests. Figure 11 shows the cache hit rate of MERCURY scheme compared with private, shared, MERCURY, Mengeable and PCM respectively. MERCURY, Mengeable and PCM are respectively 65.22%, 67.15%, 86.92%, 77.48%, 69.27% on the 4-core system, 61.68%, 63.73%, 83.87%, 74.16% and 65.12% on the 8-core systems, and 59.51%, 61.34% 81.73%, 70.52% and 62.35% on the 16-core systems. MERCURY has the better performance in this metric than Mengeable and PCM since 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
Execution time in our performance evaluation includes the identification and placement of the correlated data in the L1 caches. We evaluate MERCURY, PCM, Mengeable and standard LSH schemes in terms of time overhead as shown in Figure 12(a). The time overhead is normalized to those in Mengeable scheme. MERCURY makes use of hashing computation to identify correlated data, thus requiring smaller execution time than Mengeable that needs to carry out the extra operations of merging cache blocks. Compared with Mengeable, MERCURY decreases the execution time by 28.73%, 21.84% and 19.56% 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 Mengeable requires temporary storage space, e.g., content-addressable memory to find similar data. Figure 12(b) shows the comparisons among MERCURY, PCM, Mengeable and standard LSH schemes in terms of space overhead. We observe that compared with the Mengeable scheme, MERCURY obtains significant space savings and decreases the space overhead by 47.35%, 45.26% and 40.82% 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 [42]. Specifically, we set up 100 clients 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 dataset sizes: 500MB, 1GB, and 10GB.
for TPC-H; 1GB, 5GB, and 10GB 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 2.66GHz CPUs, 8GB memory, and four 250GB disks. Each processor has four cores, and every two cores share a 4MB L2 cache. The DBMS used in our experiments is the PostgreSQL. We measured the performance by three metrics, L2 cache miss rate, query execution time (Cycles Per Instruction) and patching costs.

For a given dataset, 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 pre-sample estimation, in which we randomly extract a subset from the used trace and make the estimation, which has been successfully used in real-world applications [33], [37], [41], [43]. 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: \((\text{ActualValue-Minimum})/\text{(Maximum-Minimum)}\). For instance, for the attribute of file size, we assume that the size range is from 10KB to 200KB, (i.e., range: 10-200). For a file with 120KB, its normalized value is \((120-10)/(200-10)=0.58\).

We evaluate the real implementation performance by comparing MERCURY with MCC-DB [20]. MCC-DB explores and exploits data access patterns and caching behaviors to efficiently support cache partitioning from query analysis. Due to similarity property, MCC-DB is comparable to MERCURY. Specifically, the reasons for making this comparison are threefold. First, both MCC-DB and MERCURY work well as patches to PostgreSQL [19] for concurrent queries. Second, the essential property behind two techniques is to use cache partitioning in multi-core processors to improve system performance. Third, MCC-DB has provided standard experimental results by using TPC-H [42] to facilitate fair comparisons with other methods. TPC-H 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

Figures 13 and 14 respectively show the L2 miss rates when using TPC-H and TPC-C workloads with different dataset sizes. We first examine the rates of three typical queries, i.e., Q5, Q8 and Q20, in both MCC-DB and MERCURY schemes. Q5 and Q8 are dominated by multi-way hash joins and Q20 is dominated by nested sub-query executions. We observe that MERCURY obtains on average 42.6%, 48.1% and 51.2% miss decrease compared with MCC-DB in the TPC-H workload with 500MB, 1GMB and 10GB 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%, on average 21.8% for all 22 queries as shown in Figure 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.

![Fig. 12. Normalized time and space overheads.](image)

![Fig. 15. L2 miss rates in 10GB dataset with 4MB L2 cache.](image)

### 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 perfmon tool to check hardware counters. Figures 16(a) and 16(b) respectively show the executing time of queries in 1GB and 10GB dataset 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% improvements upon MCC-DB respectively in 1GB and 10GB sizes.
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 datasets under different sizes. MERCURY requires less time and space costs than MCC-DB because MERCURY makes use of fast MCLSH hashing computation to place similar data together. We also observe that with the increase of dataset 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

Multi-level cache hierarchy has been studied in the high-performance cloud architecture and software communities. There exist a wide range of proposals to improve caching performance (e.g., hit rate, access latency and space overhead) [11], [22], [23], [44]–[46]. We argue that suitable management of the multi-level cache hierarchy is becoming more important to deliver high performance in the cloud.

Locality-based Optimization. The state-of-the-art work, R-NUCA [47], obtains near-optimal cache block placement by classifying blocks online and placing data close to the core. In order to mitigate the loss of reusing cached states
when rescheduling a process, affinity scheduling [48] helps reduce cache misses by judiciously scheduling a process on a recently used CPU. In order to improve the performance in “multi-execution” 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. 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 [20] 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.** In order to reduce the cache pollution caused by LRU that inserts non-reuseable items into the cache while evicting reusable ones, ROCS [6] employs hardware counters to characterize cache behaviors and introduces a pollute buffer to host not reused cache lines of pages before eviction. Moreover, in order to address the problems of increased capacity interference and longer L2 access latency, CloudCache [49] 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 [50] helps regulate fairness in modern multi-core processors while demonstrating the relative benefits of the various resource control mechanisms. Moreover, in order to reduce the large amounts of misses in LLC between the eviction of a block and its reuse, Scavenger [51] divides the total cache storage into a conventional cache and a file architecture to identify and retain high-priority cache blocks that are more likely to be reused. MERCURY bridges the gap between multi-core architecture and operating systems. Existing hardware acceleration approaches can use MERCURY to simplify operations and optimize system calls.

**Operations Enhancements.** MergeSort [52] performed an efficient multiway merge without being constrained by the memory bandwidth for high-throughput database applications. Parallel skyline computation could benefit from multi-core architectures, such as parallel version of the branch-and-bound algorithm. Authors in [53] presented a parallel algorithm based on parallel programming that was evaluated as a case study of parallelizing database operations. A cooperation based locking paradigm [54] was proposed for efficient parallelization of frequency counting and top-k over multiple streams in the context of multi-core processors. In addition, adaptive aggregation [55] demonstrated that a chip multiprocessor with new dimensions could enhance concurrent sharing of aggregation data structures and accesses to frequently used values. Authors in [56] 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 multi-core CPU [57] and tree-structured data [58]. Two popular join algorithms, such as hash join and sort-merge join, was re-examined in [59] to use multi-core cache blocking to minimize access latency, increase compute density and balance load among cores, even for heavily skewed input datasets. CATCH [60] can store unique contents in instruction cache by means of hashing, but their proposed system does not support modifications in cached data. In addition, cache compression technique [61] compresses the L2 data results to reduce the cache space and the off-chip accesses, thus obtaining bandwidth savings. Cooperative caching technique [62] 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 [32] presents execution- and measure-based strategies for multi-core cache partitioning upon multiple representative workloads. A non-uniform cache architecture (NUCA) [63] takes advantage of proximity of data from the accessing processor. In order 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 [64] 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 [65] makes use of the DelinquentPC-Next-Use characteristic to improve the performance of shared caches. The NUcache organization logically partitions the associative ways of a cache set into MainWays and DeliWays. MERCURY is orthogonal to existing schemes. It leverages light-weight LSH based computation and obtains significant performance improvements on LLC by accurately capturing the differentiated locality across data.

**Scheduling.** Age based scheduling for heterogeneous multiprocessor [66] 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 multi-threading processors was proposed in [67] to use the helper thread to perform aggressive data preloading. In order to improve the utilization of onchip memory and reduce the impact of expensive DRAM and remote cache accesses, \( O^2 \) scheduling [68] schedules objects and operations to caches and cores. In order to decrease the unnecessary sharing of network control state at all stack layers, IsoStack architecture [69] 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 multi-threaded applications. Existing scheduling strategies can further help optimize MERCURY performance.
8 Conclusion

**MERCUORY**, as an infrastructure of the cloud, plays a significant role in managing the multi-level cache hierarchy. By exploring and exploiting data similarity that is derived from locality-aware access patterns, **MERCUORY** alleviates homogeneous data placement and improves system cost performance by low-complexity MC-LSH computation. The cost-effective **MERCUORY** 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. **MERCUORY** hence allows the OS control over the shared LLCs, while minimizing software overheads. Experiments using real-world datasets demonstrate the **MERCUORY**’s efficiency.

Since modern microprocessors increasingly incorporate multiple memory controllers [70], [71], our future work will consider the locality-aware schemes for implementing efficient data placement in multiple memory controllers.

Acknowledgment

This work was supported in part by National Basic Research 973 Program of China under Grant 2011CB302301; National Natural Science Foundation of China (NSFC) under Grant 61173043, 61025008; Fundamental Research Funds for the central universities, HUST, under grant 2012QN098; the NSERC Discovery Grant 341823; US National Science Foundation Award 1116606. The authors greatly appreciate anonymous reviewers for constructive comments.

References

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 (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), 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 member of the IEEE, ACM and USENIX.

Xue Liu is an associate professor in the School of Computer Science at McGill University. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2006. He received his B.S. degree in Mathematics and M.S. degree in Automatic Control both from Tsinghua University, China. His research interests are in computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. 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 serves as an associate editor of the IEEE Transactions on Parallel and Distributed Systems and editor of the IEEE Communications Surveys & Tutorials. 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 on Parallel and Distributed Systems (TPDS), JCSS, USENIX ATC, FAST, ICDCS, HPDC, SC, ICS and ICPP. She serves as the program committees of multiple international conferences, including SC and MSST. She is a member of the IEEE.