2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems

MERCURY: A Scalable and Similarity-aware Scheme in Multi-level Cache Hierarchy

Yu Hua
Huazhong University of Science and Technology
Wuhan, China
csyhua@hust.edu.cn

Xue Liu
McGill University
Montreal, Quebec, Canada
xueliu@cs.mcgill.ca

Dan Feng
Huazhong Univ. of Sci. and Tech.
Wuhan, China
dfeng@hust.edu.cn

***Abstract*—The management of multi-level caching hierarchy is a critical and challenging task. Although there exist many hardware and OS-based schemes, they are difficult to be adopted in practice since they incur non-trivial overheads and high complexity. In order to efficiently deal with this challenge, we propose MERCURY, a cost-effective and lightweight hardware support to coordinate with OS-based cache management schemes. Its basic idea is to leverage data similarity to reduce data migration costs and deliver high performance. Moreover, in order to accurately and efficiently capture the data similarity, we propose to use 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 efficiently 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. Experiments through real-world benchmarks further corroborate the efficacy and efficiency of MERCURY.**

***Keywords*-Multi-core processor; cache management; system scalability; data similarity.**

I. Introduction

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 parameters at each level. Proper policies help minimize the latency to frequently accessed data [1]–[4]. The focus of 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 system performance in the CMPs.

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 high performance applications. Despite the popularity, it is still a daunting task to accurately and efficiently perform the multi-core caching for high performance computing.

A good design on efficient cache hierarchy management needs to answer the questions, such as "*how to significantly improve the cache utilization and how to efficiently support the cached 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 [5], [6].

Challenge 2: Performance Bottleneck Shift in High Performance Systems. Multi-core based hardware advancements bring new challenges to the design and the implementation of high performance systems [7]. 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 comes is correlated with the placement problem of cached data. The optimization of cached data placement hence becomes important to improve the overall 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

1526-7539/12 $26.00 © 2012 IEEE
DOI 10.1109/MASCOTS.2012.49

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) [1] 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 [5], [8]. 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.

Although existing hardware based schemes work for meeting the design requirements, their lack of flexibility is becoming an inherent weakness and potential performance bottleneck, especially for future multicore processors that have an increasingly large number of cores. On the other hand, the OS based cache schemes have demonstrated their efficiency and effectiveness to offer competitive solutions to the hardware's problems, with the aid of the predictable access patterns and well-analyzed data similarity. For examples, the simple LRU cache replacement policy [9] can successfully support the cache management in the single processor. Unfortunately, in the context of multicore processors, the OS based cache management becomes more challenging than ever due to the separate management schemes in the hardware and the OS. Hardware caches are beyond the scope of the OS management. Multiple threads are hence difficult to share and schedule. In the meanwhile, the OS-based schemes incur non-trivial software overheads.

In order to alleviate the limitations in the hardware solutions and the OS-based methods, this paper proposes a cost-effective and efficient scheme, called MERCURY. The rationale behind MERCURY comes from the observation that performing the state maintenance and reference pattern analysis at page granularity generally incurs less overheads than at block [1], [5], [8]. *MERCURY* hence plays a significant role in managing the multi-level cache hierarchy. The cost-effective MERCURY is able to provide hybrid functionalities. One is to provide a lightweight hardware 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-efficiency structures. MERCURY hence allows the OS control over the shared LLCs, while minimizing the software overheads. 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 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, 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 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 implemented *MERCURY* in a PolyScalar [10]. The extensive experiments make use of multiple real-world traces and datasets under multiple performance evaluation metrics.

The remainder of this paper is organized as follows. Section II describes the proposed *MERCURY* caching scheme. Section III shows the cached data management schemes in the multi-level hierarchy. Section IV shows the performance evaluation results through extensive experiments. We present the related work in Section V. Finally, we conclude our paper in Section VI with summaries of findings.

II. MERCURY Architecture

MERCURY uses MC-LSH to identify similar data and leverages an LRU replacement in each cache to update stale data. Figure 1 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.

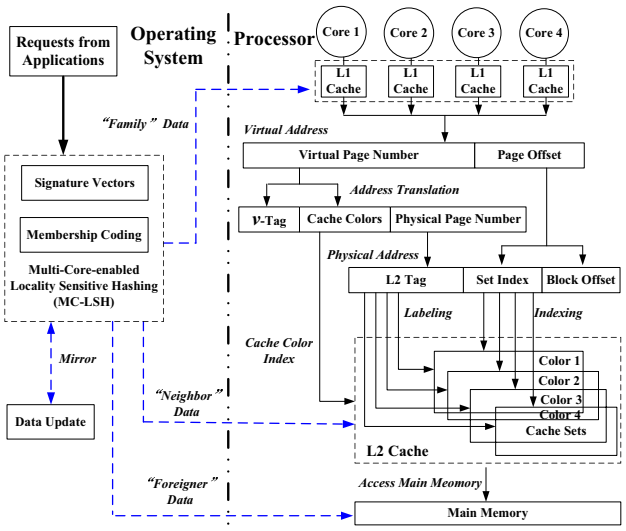

Figure 1. *MERCURY* multi-core caching architecture.

A. 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 [11] 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 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 from 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.

B. Operating System

Operating system functionalities includes the MC-LSH computation and the locality-aware data update schemes.

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 III.

Locality-aware Data Update: 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 [2]. 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 support data update operations. Since the cached data are locality-aware, *MERCURY* hence decreases migration costs and minimize cache conflicts.

C. Interface to Applications

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 1. Its main function is to wrap high-level operation requests to low-level system calls with the aid of the page coloring technique [11]. Page color manages the bits between the cache index and the physical page number in the physical memory address. Specifically, the applications 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 utilize the page coloring technique to complete cache partitioning among different processes and support the queries.

III. 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 a data update scheme in *MERCURY*.

A. 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 that pursue precise results, MC-LSH provides a near-accurate and fast approach to obtaining 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 [12] can identify the similar data by allowing them to be placed into the same hash buckets with a high probability. In practice, in order to support similar queries, LSH needs to hash a query point q into the buckets of multiple hash tables. All data items in the chosen buckets are then united and ranked according to their distances to the query point q in a geometric space (e.g., Euclidean space). By using the LSH functions, the similar data have a higher probability of colliding than the data that are far apart [13].

Definition 1: Given a data domain S and a distance function $||*||$, 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 practice, for the used hash functions 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 large constant. In order to enlarge the gap between P_1 and P_2, we make use of multiple hash tables that may incur the space inefficiency.

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 hamper the use of the multi-core benefits for high performance systems. Unlike existing work, *MERCURY* enables LSH to be space-efficient and leverages the differentiated multi-type policy.

Space-efficient Signature Vector. MC-LSH leverages a space-efficient signature vector 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.

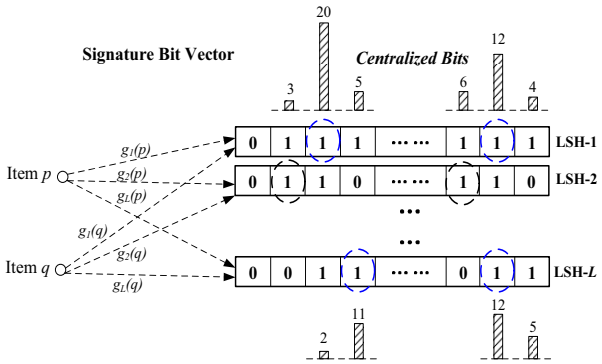

Figure 2. Signature vector for maintaining page-level data similarity.

A signature vector is able to maintain the data similarity as shown in Figure 2. 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 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%, meaning that similar data are still possible to be placed into adjacent bits. False negative hence takes place 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 totally $2t$ bits (t left and t 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 [14] by probing $\delta \in \{+1, -1\}$ neighbors.

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 in bulk 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 [15], rather than bits, in the Bloom filters. Each indexed counter is incremented when adding an item and is decremented 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*, *Neighbor* 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.

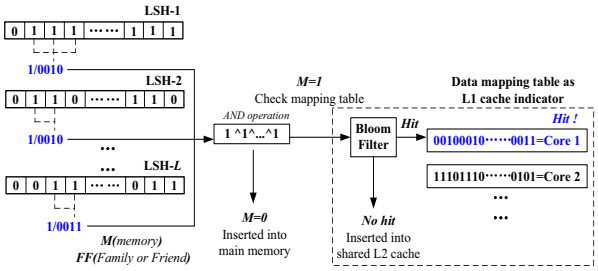

Figure 3. Differentiated membership coding technique.

We use an example to illustrate the differentiated membership coding as shown in Figure 3. 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 existing L1 caches. Based on the conclusion in [14], 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 multi-core 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 multi-core 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 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.

B. Data Update

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, which have been inserted into corresponding counting Bloom filters, into the shared L2 cache, in which we label them by using the page colors of the correlated cores to facilitate the cache update. Once free space is available in an L1 cache, *MERCURY* reloads these labeled data into the corresponding L1 cache.

The cache update 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. The L1 caches employ the simple LRU replacement to update stale data. When the data in the L1 caches become stale, they are transferred into the shared L2 cache among multiple cores. Moreover, if the data in the L2 cache become stale, they move to the main memory. Therefore, our cache update is actually a multi-level migration process from the L1 cache, then the L2 cache, finally to the main memory.

For updating memberships. In order to update the data membership in the signature vectors, we make use of 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 capacity. 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.

IV. Performance Evaluation

A. Experiment Configuration

We use simulation study primarily for the evaluation of *MERCURY*'s scalability. Our simulation uses PolyScalar [10], in which 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 II and III. 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. The simulated memory transactions are pipelined.

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 I shows the parameter settings in the simulations.

Table I
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

The properties of the used traces and datasets are listed.

- **Forest CoverType** dataset [16] contains 581,012 data points, each of which has 54-dimensional attributes.
- **TPC-H** benchmark [17] has a large volume of data for decision support systems when executing 22 different types of queries.
- **EECS** NFS server at Harvard [18] collects the I/O accesses. This dataset contains the requests with a total of 4.4 millions operations.
- **HP** file system provides a 10-day 500GB trace [19] that records the accesses from 236 users.
- **175.vpr** and **300.twolf** show the CPU performance in the SPEC2000 evaluation [20]. 175.vpr leverages combinatorial optimization technique to automatically synthesize the mapped circuits. 300.twolf makes use of TimberWolfSC placement and global routing package.

We use the three metrics to evaluate the performance, i.e., *Throughput*, *Weighted speedup* and *Fair speedup* as shown in Table II. 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 decrements 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.

Table II
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	$n/\sum_{i=1}^{n}(IPC_{base}[i]/IPC_{scheme}[i])$

B. Results

We compare *MERCURY* with baseline approaches, i.e., *private* and *shared* caches, and state-of-the-work, PCM [7] and Mergeable [2] schemes, which we re-implemented for the experiments.

1) Throughput: Figure 4 shows the throughput result from running 6 real-world applications with the increments of multi-core number from 4 to 16. The average throughputs on 4-core systems with private cache, shared cache, PCM and *MERCURY* are respectively 1.352, 1.563, 1.815 and 2.162. For 8-core systems, the average throughputs are 2.481, 2.572, 2.953 and 3.305. For 16-core systems, they are 3.281, 3.469, 3.957 and 4.452.

We observe that two typical SPEC2000 benchmarks obtain the larger throughputs on average by 15.7% increments 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 scheme.

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

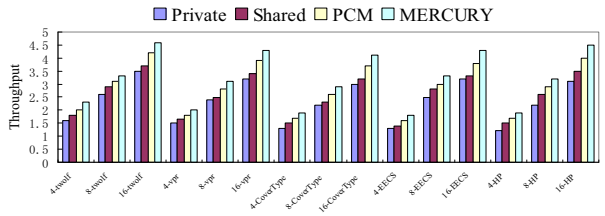

Figure 4. Throughput (Sum of IPCs).

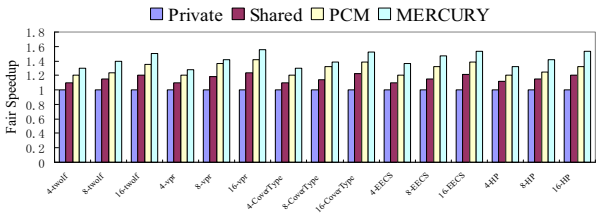

Figure 6. Normalized Fair Speedup.

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 increments of the shared cache are 9.87%, 17.52% and 23.67% respectively on 4-core, 8-core and 16-core systems.

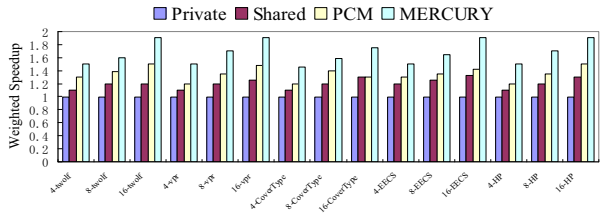

Figure 5. Normalized Weighted Speedup.

The PCM 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. *MERCURY* obtains 1.527, 1.634 and 1.928 weighted speedups, demonstrating better performance. With the increments of cores, *MERCURY* further exhibits its effectiveness and scalability since it makes use of simple hashing to adapt to the workload changes.

3) Fair Speedup: Fair speedup computes the harmonic mean of the normalized IPCs while taking into account both fairness and performance. Figure 6 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, *MERCURY* significantly improves the performance on this metric by 8.35%, 9.52% and 9.96%, respectively on 4-core, 8-core and 16-core systems. The main reason is that *MERCURY* leverages the differentiated placement policy that fairly assigns the data into the correlated caches and improves the utilization of the multi-core processor based on the multi-type memberships.

4) 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. Figure 7 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.

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% 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.

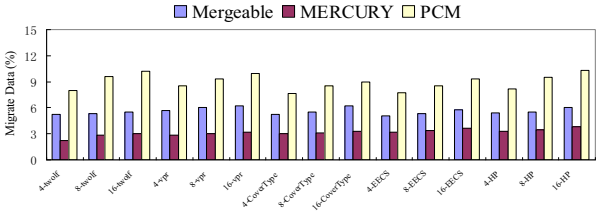

Figure 7. Percentage of migrated data.

V. Related Work

The optimization management problem of multi-level cache hierarchy in CMPs has been studied in the computer architecture and software communities. There exist a wide range of proposals to improve caching performance (e.g., hit rate, access latency and space overhead) [1], [3], [4], [21].

In order to mitigate the loss of reusing cached states when rescheduling a process, affinity scheduling [22] helps reduce cache misses by judiciously scheduling a process on a recently used CPU. In order to improve the performance in the multi-execution applications, *Mergeable* [2] captures data similarities and merges duplicate cache lines owned by different processes to obtain substantial space savings. Performing the explicitly merging operations on cache blocks requires relatively longer execution time and increases computation complexity. Process-level cache management policy (PCM) [7] has the assumption that all memory regions belonging to a running process exhibit the same access pattern. MCC-DB [23] makes use of different locality strengths and query execution patterns to minimize cache conflicts. An OS-based cache partitioning mechanism [24] presents execution- and measure-based strategies for multi-core cache

partitioning upon multiple representative workloads. Moreover, integrated processor-cache partitioning [4] divides both the available processors and the shared cache in a chip multiprocessor among different multi-threaded applications. *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.

VI. Conclusion and Future Work

This paper proposed *MERCURY*, a novel multi-level cache hierarchy designed for high performance systems running on CMPs. *MERCURY* significantly improves overall system performance. It explores data similarity that is derived from locality-aware access patterns to alleviate homogeneous data placement and improve system performance by using the low-complexity MC-LSH computation. We also propose the use of space-efficient signature vectors to obtain significant space savings. A simple coding technique further helps maintain the multi-type memberships for carrying out differentiated data placement. Experimental results demonstrate the efficiency and efficacy of *MERCURY*. Our future work will consider the methods of improving the hashing accuracy for capturing the data similarity.

Acknowledgment

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, 60703046; Fundamental Research Funds for the central universities, HUST, under grant 2012QN098. The authors greatly appreciate anonymous reviewers for constructive comments.

References

[1] 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. MICRO*, pp. 258–269, 2009.

[2] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. Chong, "Multi-execution: multicore caching for data-similar executions," *Proc. ISCA*, 2009.

[3] M. Chaudhuri, "Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," *Proc. HPCA*, pp. 227–238, 2009.

[4] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir, "A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors," *Proc. SC*, 2009.

[5] X. Ding, K. Wang, and X. Zhang, "SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores," *Proc. EuroSys*, 2011.

[6] Y. Chen, S. Byna, and X. Sun, "Data access history cache and associated data prefetching mechanisms," *Proc. SC*, 2007.

[7] 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. SC*, 2009.

[8] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John, "The virtual write queue: coordinating DRAM and last-level cache policies," *Proc. ISCA*, 2010.

[9] T. R. B. Bershad, D. Lee and B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," *Proc. ASPLOS*, 1994.

[10] "PolyScalar," *http://users.csc.calpoly.edu/ franklin/PolyScalar /Home.htm*.

[11] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice-a low-cost high-speed address translation mechanism," *Proc. ISCA*, 1990.

[12] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *Proc. STOC*, 1998.

[13] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *Communications of the ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: Efficient indexing for high-dimensional similarity search," *Proc. VLDB*, pp. 950–961, 2007.

[15] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[16] The Forest CoverType dataset, "UCI machine learning repository," *http://archive.ics.uci.edu/ml/datasets/Covertype*.

[17] TPC *http://www.tpc.org/*.

[18] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Proc. FAST*, 2003.

[19] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *Proc. FAST*, 2002.

[20] SPEC2000 *http://www.spec.org/cpu2000/*.

[21] R. Manikantan, K. Rajan, and R. Govindarajan, "Nucache: An efficient multicore cache organization based on next-use distance," *Proc. HPCA*, pp. 243–253, 2011.

[22] J. Torrellas, A. Tucker, and A. Gupta, "Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary," *Proc. ACM SIGMETRICS*, 1993.

[23] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: minimizing cache conflicts in multi-core processors for databases," *Proc. VLDB*, vol. 2, no. 1, pp. 373–384, 2009.

[24] 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. HPCA*, 2008.