# Homogeneous Data Detection Using Mercury for the Cloud

Roy Thomas
University College of Engineering, Nagercoil

K.L.Neela
Assistant Professor, University College of Engineering, Nagercoil

**Abstract:-Cloud computing could definitely benefit from the rapid growth of multicore system. Handling of huge data and decreasing data relocation requires the computation infrastructure of the cloud to place and manage cached data efficiently. Cloud needs to significantly improve the cache utilization and efficiently support the data placement. The management of multi-level caching hierarchy is a critical and challenging task. Although there exists 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, this project titled "Homogeneous Data Detection using MERCURY for the Cloud" proposes MERCURY, a cost-effective and lightweight hardware support to coordinate with OS-based cache management schemes. Its basic idea is to pull data similarity to reduce data migration costs and deliver high performance. A multicore enabled Locality-Sensitive Hashing (MC-LSH) in the computation infrastructure is used for the cloud to accurately capture the similarity among the data. It efficiently partitions data into L1 cache, L2 cache and main memory based on their distinct localities in a multicore system environment. This helps to optimize cache utilization and minimize the replacement of a last level cache element with a less useful one. Experiments based on real-world applications demonstrate the effectiveness and efficiency of Mercury.**

**Index Terms - Cloud computing, multicore processor, cache management,homogeneous data, cache coloring.**

## 1 INTRODUCTION

Cloud computing provides a convenient, on-demand network access to a shared pool of configurable computing resources like networks, servers, storage, applications, services etc. that can be rapidly provisioned and released with minimal management effort or service provider interaction. Cloud consists of large volume of data and these data are heterogeneous. Even though the data sets are very large, the data streams associated are slow, of the speed of Gigabits per second. The demands for data accessibility on clouds are increasing [14, 19]. One of the challenges that the cloud service providers face is to serve the needs of the users in the shortest possible time. The services provided to the users should guarantee high quality and it must avoid duplication. Commercial companies like Google, Microsoft, Yahoo!, Facebook etc. generally handle very large volume of data everyday [15]. The computation environment for the cloud should provide efficient processing and analysis of these data to obtain the required quality of service. An interesting behavior exhibited by these applications is that there exists high degree of similarity across the data of various instances of the application.

Multi core processors are increasingly used today. The cores are fully functional with computation units and caches and hence support multithreading. In a many-core processor the number of cores is large enough that traditional multi-processor techniques are no longer efficient. Cores on a multi-core device can be coupled tightly or loosely. They may share or may not share a cache. They also implement inter-core communications methods or message passing. Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, SIMD or multi-threading. Chip Multiprocessors (CMPs) use relatively simple single-thread processor cores to exploit only moderate amounts of parallelism within any one thread, while executing multiple threads in parallel across multiple processor cores. If an application cannot be effectively decomposed into threads, CMPs will be underutilized. A multicore processor is more adequate for parallel processing. Each microprocessor manufacture has its specific implementation of multicore technology. The number of cores on a single chip is increasing to hundreds of cores per chip. The performance potential from this technology is to be greatly explored. One of the challenges in using multi core technology is to get lost to think of all new ways to explore the technology as traditional methods look comfortable [4].

A multicore architecture contains several CPU cores each equipped with a private first level cache. This is common with all multicore designs. Together with the core and the L1 cache, the chip also contains a second level (L2) cache or a few L2 caches, depending on the concrete implementation. The former can be typically found in the Intel multicore

processors and the latter is adopted by AMD. Through a memory controller, the second level cache is combined with main memory. Additionally, some products have an off-chip L3 in front of the main memory. Hence it is important to find methods to improve the cache utilization and to efficiently place the data within the cache. These problems are very difficult and challenging to address in the case of multicore systems.

The challenges we have to address are *inconsistency between CPU and operating system caches*, *performance bottleneck in cloud system memory access* and *handling of last-level cache (LLC) pollution.*

Caches are used to bridge the speed gap between CPU and memory. The two types of caches used are CPU caches – L1 cache, L2 cache etc. and operating system (OS) buffer cache. The CPU caches are at the hardware level where as the OS cache is at the software level. These two are developed independent of each other. Hence inconsistency exists between these two caches. The degree of inconsistency is higher in the case of multicore systems. This leads to performance degradation in systems where a cache is shared by several cores [6]. In cloud systems the performance bottleneck has been shifted from slow I/O access to high memory access latency. Performance bottleneck is related to the placement of data in the cache. Hence suitable placement of data in the cache is necessary to improve the overall performance of the cloud systems. The existing policies in multicore systems are neither efficient nor scalable. To address these problems we need to find the data similarity and to optimize the utilization of cache- both private and shared caches. The last level cache (LLC) is dynamically shared among all the cores in a multicore system. Each core has its lowest level of cache hierarchy. Cache pollution – the replacement of a cached data with a less useful one – occurs when a non-reusable cache line is installed into a cache set. The conventional approaches to alleviate the LLC pollution use recent ordering as it serves as good prediction for subsequent behaviors of cache accesses.

The proposed system alleviates these limitations. Mercury is implemented and the similarity is managed at the page levels by using the operating system mechanisms. This is based on the observation that pattern analysis at the page level incurs less overhead than at the block level. Pattern analysis at the page level requires less state and storage space compared to the block level policies. The system is also compatible with the existing cloud computing systems and can further improve by providing an efficient and scalable caching scheme. It plays an important role in multilevel cache hierarchy. It employs multitype membership management to narrow the inconsistency gap between CPU and operating system caches. The data in the multicore caches are classified into three types. They are *frequently accessed and correlated*, *frequently accessed but not correlated*, and *infrequently accessed* data sets. The system uses a multicore enabled locality sensitive hashing (MC-LSH) scheme to address the performance bottleneck and to alleviate the LLC pollution. Locality sensitive hashing (LSH) is used to accurately capture the data similarity in the computation infrastructure for the cloud. A conventional LSH scheme suffers from the problem of homogeneous data placement and also it has the disadvantage of space inefficiency. A Multi-Core-enabled LSH (MC-LSH) is used to address these problems. In MC-LSH a signature vector is used instead of many hash tables in a standard locality sensitive hashing. The significant improvements in using signature vector are the space savings and accuracy in measuring the data similarity. It minimizes the cache conflicts and reduces the amount of migrated data. This in turn significantly reduces the low-speed memory accesses. We have implemented the components and functionalities of the system in a software layer and evaluated the efficiency using a database system.

## 2. RELATED WORK

Efficient management of multilevel cache hierarchy is important to obtain high performance in the cloud. There exists a wide range of proposals to improve cache performance. R-NUCA obtains near-optimal cache block placement by online classification of blocks and placing data close to the core [8]. Affinity scheduling [13] reduces cache misses by judiciously scheduling a process on a recently used CPU. A common class of workloads for shared-memory multiprocessors is multi-programmed workloads. Data prefetching [5] is an effective way to bridge the increasing performance gap between processor and memory. Data Access History Cache (DAHC) is a cache architecture which is capable of supporting many well-known history-based prefetching algorithms, especially adaptive and aggressive approaches. Simulation experiments can be carried out to validate DAHC design and DAHC-based data prefetching methodologies and to demonstrate performance gains. Similarity search [12] methods are widely used as various data mining and machine learning. Nearest neighbor search (NNS) algorithms are often used to retrieve similar entries, given a query. A Ternary Content Addressable Memory (TCAM) can query for a bit vector within a database of ternary vectors, where every bit position represents 0, 1 or *. TCAM access and storage is nearly linear in the size of the database.

Bounded Locality Sensitive Hashing (Bounded LSH)[9] method is used for similarity search in P2P file systems. Bounded LSH makes improvement on the space saving and quick query response in the similarity search, especially for high-dimensional data objects that exhibit non-uniform distribution property. Bounded-LSH is a simple and space-

efficient mapping of non-uniform data space into load-balanced hash buckets that contain approximate number of objects. The Mergeable [4] cache architecture that detects data similarities and merges cache blocks, resulting in substantial savings in cache storage requirements. This leads to reductions in off-chip memory accesses and overall power usage, and increases in application performance. This technique provides a scalable solution and leads to significant speedups due to reductions in main memory accesses. Summary Cache [7] is a protocol in which each proxy keeps the URLs of cached documents of each participating proxy. Two factors contribute to the low overhead: the summaries are updated only periodically, and the summary representations are economical as low as 8 bits per entry.

A dynamic mechanism using multiple memory controllers [2] is used that takes the memory access latency variation into account when placing data in appropriate slices of physical memory. An adaptive first-touch page placement and dynamic page-migration mechanisms are used to reduce DRAM access delays for multi-MC systems. Multi-probe LSH [11] is built on the well-known LSH technique, but it intelligently probes multiple buckets that are likely to contain query results in a hash table. The multi-probe LSH method is implemented and evaluated the implementation with two different high-dimensional datasets.

Scavenger [3] is an architecture for last-level caches. Scavenger divides the total storage budget into a conventional cache and a victim file architecture, which employs a skewed Bloom filter in conjunction with a pipelined priority heap to identify and retain the blocks that most frequently missed in the conventional part of the cache in the past. An adaptive cache compression [1] policy is implemented that dynamically adapts to the costs and benefits of cache compression. This policy is based on a two-level cache hierarchy where the L1 cache holds uncompressed data and the L2 cache dynamically selects between compressed and uncompressed storage. MCC-DB [10] makes use of different query execution patterns to minimize cache conflicts.

## 3. PROPOSED MODELLING

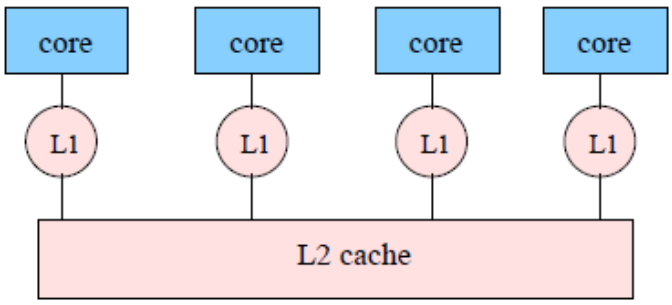
The property of data similarity is helpful to perform an efficient and scalable caching. The main benefits obtained from such a scheme include throughput improvements and the reduction of the last level cache misses rates, query latency, and data migration overheads.

The system is designed based on the observations that data similarity widely exists in the real-world applications. The problem of similarity search refers to finding objects that have similar characteristics to the query object. When data objects are represented by d-dimensional feature vectors, the goal of similarity search for a given query object q, is to find the K objects that are closest to q according to a distance function in the d-dimensional space. The search quality is measured by the fraction of the nearest K objects that are able to retrieve.

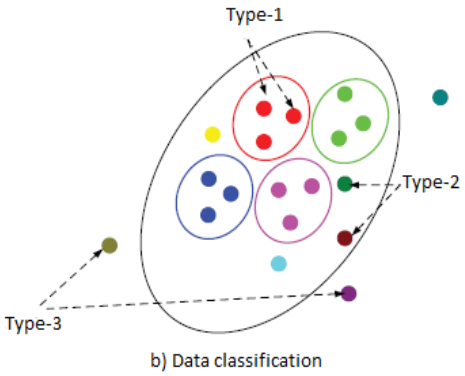
Data similarity is defined based on a predefined threshold value. For two data with point representations as a and b, having d-dimensional attributes that are represented as vectors $\vec{a}_d$ and $\vec{b}_d$, if the geometric distance between vectors $\vec{a}_d$ and $\vec{b}_d$ is smaller than the predefined threshold, they are similar. The data similarity often hides behind the locality of access patterns.

### 2.1 Multitype Membership

The cached data placement in the multilevel cache hierarchy for a multicore architecture to offer efficient computation infrastructure for the cloud is shown in Fig 2.1(a). The term "cache-member," determines the data memberships in each cache based on the given constraints. The constraints include migration costs and data access latency. It is analyzed based on the architecture where multicore processors share a common L2 cache and each core has its own private L1 cache.




a)Private L1 caches and shared L2 cache




b) Data classification

Figure 2.1 Cache-member Description

It is needed to identify and aggregate similar data into the same or adjacent private L1 caches, and then allocate the data

accessed by more than one core into a shared L2 cache. Thus the cached data is managed in both L1 and L2 caches. Moreover, an ideal multicore architecture is scalable and flexible to allow dynamic and adaptive management on the cached data. The premise is to accurately capture the similar data.

The homogeneous data placement is based on the distinct properties and multitype memberships of cached data. A differentiated placement policy is used to improve the homogeneous data management. In this policy the cache memberships are classified into three types as shown in Figure 2.1(b). The accessed data are classified into three categories. They are i) Frequently accessed and correlated data (Type-1), ii) Frequently accessed but loosely correlated data (Type-2), and iii) Infrequently accessed data (Type-3).

After the classification of data they are placed in different locations. The frequently accessed and correlated data are placed in private L1 cache, and the frequently accessed but loosely correlated data are placed in L2 cache. The infrequently accessed data are placed in the main memory. In this way, the strength of access locality is differentiated to facilitate the efficient placement of cached data.

2.2 Locality Sensitive Hashing (LSH)

Data stored in the cloud are heterogeneous and of high dimensions. Hence finding the data similarity is a time consuming and computation-intensive work. To accomplish a suitable tradeoff between similarity accuracy and operation complexity, a hash-based approach, the LSH is used due to its locality-aware property and ease of use. The LSH can identify and place similar data together with low complexity. Locality-Sensitive Hashing (LSH) is a method which is used for determining which items in a given set are similar. Rather than using the naive approach of comparing all pairs of items within a set, items are hashed into buckets, such that similar items will be more likely to hash into the same buckets. As a result the number of comparisons needed will be reduced; only the items within any one bucket will be compared. Locality-sensitive hashing is often used when there exist an extremely large amount of data items that must be compared. In these cases, it may also be that the data items themselves will be too large, and as such will have their dimensionality reduced by a feature extraction technique beforehand.

The main application of LSH is to provide a method for efficient approximate nearest neighbour search through probabilistic dimension reduction of high-dimensional data. This dimensional reduction is done through feature extraction realized through hashing (eg. minhash signatures), for which different schemes are used depending upon the data. LSH is used in fields such as data mining, pattern recognition, computer vision, computational geometry, and data compression. It also has direct applications in spell checking, plagiarism detection, and chemical similarity. Locality Sensitive Hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items).

A locality sensitive hashing scheme is defined with respect to a universe of items $U$ and a distance metric $\phi : U \times U \rightarrow [0,1]$. An LSH scheme is a family of hash functions $H$ coupled with a distribution $D$ over the functions such that a function $h \in H$ chosen according to $D$ satifies the property that $Pr[h(a)=h(b)]=\phi(a,b)$ for any $a,b \in U$.

3. 1 System Architecture

MERCURY uses the MC-LSH to identify similar data and leverages an LRU replacement in each cache to update stale data. Figure 3.1 shows the MERCURY architecture in the multilevel hierarchy. It is assumed that each core has one private L1 cache and all processor cores share an L2 cache.

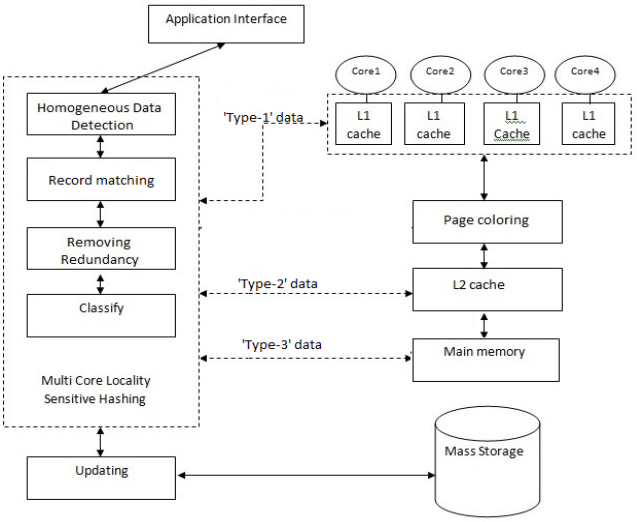



Figure 3.1 Mercury multicore caching architecture

The MERCURY scheme is tightly associated with two parts. One is the processor architecture and the other is the operating system. Furthermore, to explicitly represent the differentiated memberships identified by the MC-LSH, different flags are used to label each cache line and obtain holistic optimization in the multilevel cache hierarchy.

3.2 Cache data Management

The caching schemes in a multicore processor include L1 and L2 cache management [18], and virtual-physical address translation. L1 cache management deals with frequently accessed and closely related data placement. Each core has one associated cache that contains frequently visited data to increase the access speed and decrease the required bandwidth. L2 cache management deals with frequently accessed and loosely related data placement.

L2-cache is a shared cache and to partition the shared L2 cache, the well-known page color [16],[17] scheme is used due to its simplicity and flexibility. Page coloring is an extensively used OS technique for improving cache and memory performance. Page colouring is illustrated as shown in Figure 3.2.

A physical address contains several common bits between the cache index and the physical page number, which is indicated as a page color. One can divide a physically addressed cache into nonintersecting regions (cache color) by page color, and the pages with the same page color are mapped to the same cache color. A shared cache is divided into N colors, where N comes from the architectural settings. The cache lines are represented by using one of N cache colors.

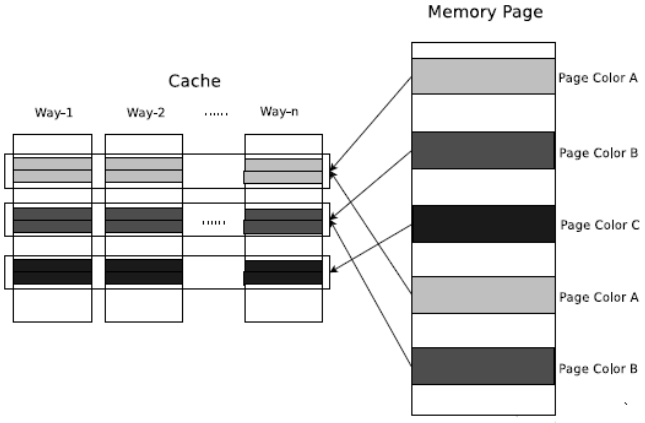



Figure 3.2 Page colouring technique

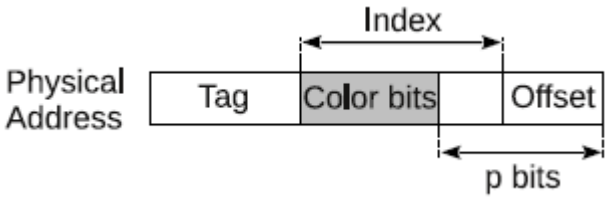



Figure 3.3 Page Colour bits

Page color manages the bits between the cache index and the physical page number in the physical memory address as shown in Figure 3.3. Page coloring is a software technique that controls the mapping of physical memory pages to a processor's cache blocks. Memory pages that map to the same cache blocks are assigned the same color. By controlling the color of pages assigned to an application, the operating system can manipulate cache blocks at the granularity of the page size times the cache associativity. This granularity is the unit of cache space that can be allocated to an application. The maximum number of colors that a platform can support is the cache line size multiplied by the number of sets and divided by the page size.

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 memory pages of the same color can be mapped to the same cache region. To efficiently partition the cache space, different page colors are allocated to memory threads. MERCURY can hence leverage the page coloring technique to complete cache partitioning among different processes and support the queries.

The operating system functionalities support the MC-LSH computation and update the locality-aware data. 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, Multicore Locality Sensitive Hashing, to offer efficiency and scalability to the multicore caching. Specifically, the MC-LSH uses a space-efficient signature vector to maintain the cached data and utilizes a coding technique to support a differentiated placement policy for the multitype data.

A key function in MERCURY is to identify similar data with low operation complexity for updating locality-aware data. This identification and updating is to be done accurately as well as fast. MC-LSH scheme is used to identify similar data and avoid brute-force checking between arriving data and all valid cache lines. The similar data are then placed in the same or close-by caches to facilitate multicore computation and efficiently update data. Since the cached data are locality-aware, MERCURY, hence, decreases migration costs and minimizes cache conflicts.

3.3 SIMILARITY Detection

Similarity detection is a main component in the MERCURY designs to detect the homogeneous data. The algorithm used for this purpose is based on Locality Sensitive Hashing. In Locality Sensitive Hashing the similarity between two items, C1 and C2, are determined as follows.

The *similarity* of C1 and $C_2$ = *Sim* $(C_1,C_2)$ = is the ratio of the sizes of the intersection and union of $C_1$ and $C_2$.

- *Sim* $(C_1,C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$ (3.1)
- The probability that $h(C_1) = h(C_2)$ is the same as *Sim* $(C_1, C_2)$

The value computed using Equation (3.1) gives the similarity between C1 and C2. It is the ratio of intersection to the union.

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

The standard LSH has the limitation that it 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., IH={h:S→U} is called $(R,cR,P_1,P_2)$-sensitive, if for $\forall p,q \in S$ :

If $\|p,q\| \leq R$ then $P_{rIH}[h(p)=h(q)] \geq P_1$, (3.2)
If $\|p,q\| > cR$ then $P_{rIH}[h(p)=h(q)] \leq P_2$ (3.3)
where $c>1$ and $P_1>P_2$.

A value 1 in the probability calculated using the Equation (3.2) shows that the items are completely similar. A value less than or equal to 1 can be used for computational purposes and this is taken as the threshold value. If the distance calculated using the probabilities in Equation (3.2) and Equation (3.3) is less than the defined threshold value, the items are considered as dissimilar. Otherwise they are treated as similar items. Thus the distance function can be used to determine the similarity.

By using the LSH functions, similar data have a higher probability of colliding than the data that are far apart. Although the LSH has been recently used in many applications, it is difficult to be used in the multicore systems due to heavy space overhead and homogeneous data placement. These limitations have severely hampered the use of the multicore benefits for high-performance systems. Unlike the existing work, MERCURY enables the LSH to be space-efficient by leveraging signature vectors.

3.4 SIGNATURE Vector

A space-efficient signature vector and a simple coding technique help maintain and represent the multitype memberships. The MC-LSH uses space-efficient signature vectors to store and maintain the locality of access patterns. A signature vector is an m-bit array where each bit is initially set to 0. LSH functions are used to map data points into bits. 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. This technique significantly decreases space overhead. A data point is given as an input to a hash function $g_i$ , which is then mapped into a bit that is thus set to 1.If a bit is set more than once only the first setting takes effect.

The signature vector is used to maintain the data similarity. The bits that receive more hits than its left and right neighbours are known as centralized bits. The hit numbers are much larger than a predefined threshold value. The centralized bits become the centers of correlated data. They are selected to be mapped and stored in the L1 caches. When hashing data into the signature vector, the hit numbers of bits are counted and carefully select the centralized bits. 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, a mapping between the centralized bits and L1 caches is constructed to facilitate the data placement. Moreover, 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 bit.

Even though the MC-LSH computation can guarantee similar data to be hashed into one bit with very high probability, it need not be 100 percent accurate. This means 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 neighbours is 1. An extra check to the neighbouring bits besides the hit one can be done to avoid potential false negatives. Although extra checking on neighbouring bits possibly incurs false positives, a miss from the false negative generally incurs the larger penalty than the false positive. MERCURY probes more than one hit bit, i.e., checking left and right neighbours, besides the hashed bit when the hit bit is "0".

MERCURY offers scalable and flexible schemes based on the characteristics of the real-world workloads to efficiently update the signature vectors. Specifically, if the workloads exhibit an operation-intensive (e.g., write-intensive)

characteristic, the operations are carried out on the signature vectors and allow the (re)-initialization in the idle time.

3.5 Membership Coding

Multitype membership coding is used in the scheme to differentiate data. The memberships in the MC-LSH include frequently accessed and correlated, frequently accessed but not correlated and infrequently accessed data. The MC-LSH identifies data memberships and places data into L1 cache, L2 cache, or main memory, respectively. To determine whether the hits in multiple LSH vectors indicate a single cache, a coding technique is used which guarantees membership consistency and integrity.

3.5 Identifying and Removing Redundancy

In the multilevel hierarchy of MERCURY, it is needed to update cached data and their memberships in the signature vector. We use a least frequently used (LFU) replacement strategy. With LFU, the cached data are all more popular than others. Moreover the frequency of accessing the data items is available from the signature vector and the counting Bloom Filter. Hence there is no extra computation complexity and no other data structure is needed. To avoid redundancy of cached data, the shared L2 cache is used to place data shared among multiple cores. The private cache L1, of each core contains separate data. These data are periodically scanned to find duplication. The similarity function is used for this. Similar data used by multiple cores are placed in shared L2 cache. Page coloring is used for improving the cache performance. In the shared L2 cache, the data is labelled using page colors of the correlated cores to update caches. For two items C1 and C2, if the ratio of the intersection to the union is greater than or equal to a predefined threshold, they are considered as similar.

The counting Bloom filters are used to update the data membership in the signature vectors. They 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. They also 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 and 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, only less storage space is required. When an item is deleted, the hit counters are decreased by 1. If all counters become 0, there are no cached data. In that situation the associated caches are initialized by sampling data to determine the locality-aware representation in the signature vector. The size of a signature vector depends on not only the amounts of data to be inserted, but also their distribution.

## 4 RESULTS AND DISCUSSIONS

MERCURY leverages the MC-LSH to identify similar data that are placed into L1 and L2 caches, respectively, with an LFU replacement policy. Specifically, each core has its own private L1 cache whereas an L2 cache is shared by multiple cores. The scalability of MERCURY is evaluated by increasing the number of cores. The experiments were done in a software simulation environment using a web server and clients. A database was used to simulate the cloud storage. In the simulation model a number of clients are setup to send out queries concurrently. For each client, queries are randomly drawn from the pool of all TPC-H queries. The experiments are repeated under different data set sizes and measured the performance by the metrics migration cost and hit rate. The outputs for these metrics are compared with other existing systems to show the efficiency of the Mercury.

4.1 Migration Cost

Hit misses or updates in caches often lead to data migration among multiple caches. This incurs relatively high costs in terms of data transmission and replacement in the caches of other cores. Fig. 4.1 shows the percentage of migrated data in Private, Shared, PCM, and MERCURY. The average percentages of migrated data are 13.2 and 11.9 percent, respectively, in private and shared caches as shown in Table 4.1.

| **Architecture** | Private | Shared | Mercury | PCM |
|---|---|---|---|---|
| **Avg percent of migrated data** | 13.2 | 11.9 | 2.8 | 8.5 |

Table 4.1 Average percentage of migrated data

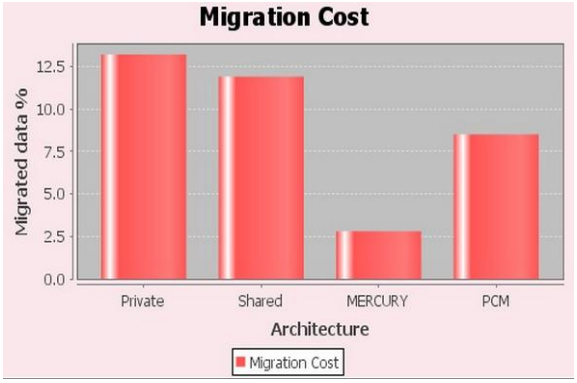



Figure 4.1 Percentage of migrated data

MERCURY can obtain better performance in this metric and decrease the number of migrated data on average by 35.26 on 4-core systems. One of the main reasons for this is that the MC-LSH provides high accuracy of identifying correlated

data, thus reducing the number of migrated data. The other reason is that the fast identification of similar data in MERCURY produces low computation complexity.

4.2 Hit Rate

One of the key metrics to evaluate cache efficiency is the hit rate that defines the probability of obtaining queried data within limited cache space for requests. Fig.4.2 shows the cache hit rate of the MERCURY scheme compared with private, shared, and PCM. The average hit rates in private, shared, MERCURY, and PCM are respectively 65.22, 67.15, 86.92, and 69.27 percent on the 4-core system, which is tabulated in Table 4.2.

| **Architecture** | Private | Shared | Mercury | PCM |
|---|---|---|---|---|
| **Avg percent of hit rate** | 65.22 | 67.15 | 86.92 | 69.27 |

Table 4.2 Average percentage of migrated data

MERCURY has the better performance in this metric than others since the MC-LSH accurately identifies correlated data within constant-scale execution complexity. The improved accuracy significantly decreases potential migration costs that possibly occur due to hit misses. The quick identification also alleviates the effects of staleness in the caches.

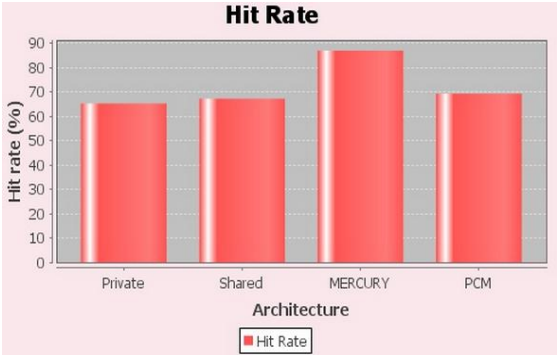



Fig 4.2 Hit rates

## 5 CONCLUSION

Mercury determines data similarity and support efficient data placement. The MC-LSH accurately captures the differentiated similarity among data. MC- LSH scheme alleviates the problem of homogeneous data placement and space inefficiency. The similarity-aware MERCURY plays an important role in multilevel cache hierarchy. It efficiently partitions data into L1 cache, L2 cache and main memory based on their distinct localities in a multicore system environment. L2-cache is a shared cache and to partition the shared L2 cache, the page color scheme is used to control cache partitioning, and consequently to achieve fair and efficient cache utilization. It employs multitype membership management to narrow the inconsistency gap between CPU and operating system caches.

## REFERENCES

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

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

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

[4] Biswas S, Franklin D, Savage A, Dixon R, Sherwood T, and Chong (2009), "Multi-Execution: Multicore Caching for Data-Similar Executions," Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA.

[5] Chen Y, Byna S, and Sun X (2007), "Data Access History Cache and Associated Data Prefetching Mechanisms," Proc. IEEE/ACM Conf. Supercomputing (SC).

[6] Ding X, Wang K, and Zhang X (2011) "SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores", Proc. Sixth Conf. Computer Systems (EuroSys).

[7] Fan L, Cao P, Almeida J, and Broder A (2000), "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," IEEE/ACM Trans. Networking, vol. 8, no. 3, pp. 281-293.

[8] Hardavellas N, Ferdman M, Falsafi B, and Ailamaki A,(2010) "Near Optimal Cache Architectures", IEEE Micro, vol 30, no.1 pp.20-28

[9] Hua Y, Xiao B, Feng D., and Yu B.(2008), "Bounded LSH for Similarity Search in Peer-to-Peer File Systems," Proc. 37th Int'l Conf. Parallel Processing (ICPP), pp. 644-651.

[10] Lee R, Ding X, Chen F,Lu Q, and Zhang X (2009), "MCC-DB: Minimising Cache Conflicts in Multi-core Processors for Databases", Proc. VLDB Endowment, vol.2,no.1,pp.373-384.

[11] Lv Q, Josephson W, Wang Z, Charikar M, and Li K(2007), "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB), pp. 950-961.

[12] Shinde R, Goel A, Gupta P, and Dutta D (2010), "Similarity Search and Locality Sensitive Hashing Using Ternary Content Addressable Memories," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD), pp. 375-386.

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

[14] Wang Y,Veeravalli B (2013) "On Data Staging Algoriths for Shared Data Accesses in Clouds". IEEE Transactions on Parallel and Distributed Systems, VOL.24, No.4

[15] Wu S, Li F,Mehrotra S. and Ooi B (2011) "Query Optimization for Massively Parallel Data Processing," Proc. ACM Symp. Cloud Computing (SOCC).

[16] Ye Y, West R, Cheng Z, and Li Y (2014) "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring" PACT'14, August 24-27,2014

[17] Zhang X, Dwarkadas S, Shen K (2009) "Towards Practical Page Coloring-based Multi-core Cache Management". Eurosys'09.

[18] Zhang Z, Zhu Z, and Zhang X,(2004) "Design and Optimization of Large size and Low Overhead Off-Chip caches,: IEEE Trans. Computers, vol.53, no.7, pp.843-855.

[19] Aarti Singh, Manisha Malhotra "Security Concerns at Various Levels of Cloud Computing Paradigm", International Journal of Computer Networks and Applications, Volume 2, Issue 2, Page No: 41-45, April – May (2015).