ACTION: Adaptive Cache Block Migration in Distributed Cache Architectures

CHANDRA SEKHAR MUMMIDI and SANDIP KUNDU, University of Massachusetts Amherst, USA

Chip multiprocessors (CMP) with more cores have more traffic to the **last-level cache (LLC)**. Without a corresponding increase in LLC bandwidth, such traffic cannot be sustained, resulting in performance degradation. Previous research focused on data placement techniques to improve access latency in **Non-Uniform Cache Architectures (NUCA)**. Placing data closer to the referring core reduces traffic in cache interconnect. However, earlier data placement work did not account for the frequency with which specific memory references are accessed. The difficulty of tracking access frequency for all memory references is one of the main reasons why it was not considered in NUCA data placement.

In this research, we present a hardware-assisted solution called **ACTION (Adaptive Cache Block Migration)** to track the access frequency of individual memory references and prioritize placement of frequently referred data closer to the affine core. ACTION mechanism implements cache block migration when there is a detectable change in access frequencies due to a shift in the program phase. ACTION counts access references in the LLC stream using a simple and approximate method and uses a straightforward placement and migration solution to keep the hardware overhead low. We evaluate ACTION on a 4-core CMP with a 5x5 mesh LLC network implementing a partitioned D-NUCA against workloads exhibiting distinct asymmetry in cache block access frequency. Our simulation results indicate that ACTION can improve CMP performance by up to 7.5% over **state-of-the-art (SOTA)** D-NUCA solutions.

CCS Concepts: • **Computer systems organization → Multicore architectures;**

Additional Key Words and Phrases: Cache, Non-uniform cache access (NUCA), frequent pages, Counting Bloom Filter

ACM Reference format:
Chandra Sekhar Mummidi and Sandip Kundu. 2023. ACTION: Adaptive Cache Block Migration in Distributed Cache Architectures. *ACM Trans. Arch. Code Optim.* 20, 2, Article 25 (March 2023), 19 pages.
https://doi.org/10.1145/3572911

1 INTRODUCTION

With the increasing disparity between computing capability and memory access times, memory performance has become one of the critical factors influencing overall system performance. Off-chip DRAM accesses are costly in terms of cycles and energy. As a result, the more data

This research was funded in part by a grant from the National Science Foundation.
Authors' address: C. S. Mummidi and S. Kundu, University of Massachusetts Amherst, P.O. Box 01003, Amherst, Massachusetts, USA, 01003; emails: {cmummidi, kundu}@umass.edu.

1544-3566/2023/03-ART25 $15.00
https://doi.org/10.1145/3572911

requests the **last-level cache (LLC)** captures, the fewer the off-chip accesses, making LLC performance crucial. Innovations in process technologies allow integrating more cores on a single chip [22], letting **Chip Multi-core Processors (CMPs)** execute an increasing number of applications in parallel. Chip data demands increase as the number of concurrent applications rises, putting even more strain on the memory system. Limited pins for off-chip DRAM accesses become a bottleneck for growing core count. CMPs employ a large LLC that takes up majority of the chip area to meet memory demands, and to reduce the number of off-chip memory accesses. For a single port cache, concurrent access requests need to be queued. Consequently, access requests to LLC in a single port large cache array spend more time in queues, reducing cache throughput. This problem could be mitigated by dividing LLC into numerous smaller banks connected by a **Network-on-Chip (NoC)**, allowing parallel banks to service concurrent requests [15].

Applications in CMPs share the same LLC; hence they compete for capacity replacing each others' cache blocks. **Cache Partitioning Techniques (CPT)** divide a shared cache into partitions and allocate each process a partition based on its memory requirements [6, 7, 19, 21, 30]. Partitioning isolates applications LLC accesses, thus eliminating interference misses.

In distributed LLC [15], cache access latency consists of cycles spent in network traversal and array access. Hence, cache blocks located in banks experience different access latencies based on proximity to the accessing core, known as **Non-Uniform Cache Architectures (NUCA)**. Furthermore, network traversal takes more cycles than cache array accesses in distributed caches, making data placement critical. Therefore, prior research in NUCA has aimed to place cache blocks nearer to the accessing core to reduce access latency [3, 8, 11].

The new memory technologies such as embedded DRAM allow cache implementation in heterogeneous technologies, enabling larger LLC capacity with high access latency trade-off [14, 26, 27, 29]. However, with limited bandwidth and high latency in DRAM cache architecture's cache block placement is even more critical.

Previous research proposed CPT for the shared cache and placement techniques for the distributed cache. However, an application's access pattern is not uniform across its memory references [2]. It accesses few memory references more frequently than others. Placing frequently accessed data in the farthest banks injects frequent access requests into the network, increasing network load. With the growing non-uniformity of LLC, the location of these frequent memory references in LLC is becoming increasingly important.

To address these problems, we present a solution called **ACTION (Adaptive Cache Block MigraTION)**, which tracks the access frequency for dynamic data placement in NUCA. ACTION uses minimal hardware to monitor the LLC access stream to identify frequent pages. In addition, we leverage partitioned NUCA to migrate frequently used pages to cache blocks nearest to the affine core.

The main contributions of this work are:

- Demonstrate that applications have a non-uniform memory access pattern where some page references are accessed more frequently than others.
- Demonstrate how the most frequently accessed pages in a program changes as the program goes through various program phases.
- A low-cost hardware mechanism that samples the LLC access stream to identify the most frequently accessed pages in rank-order for migration to a local bank, as well as a *refresh policy* for tracking page recency.
- A method to dynamically migrate frequently accessed pages to the local bank of the affine core by selectively invalidating cache pages.

We evaluate ACTION on a 5x5 mesh LLC network, four core CMP, against SPEC CPU 2006 benchmark using the ZSIM simulator [25].

The applications that are likely to benefit the most are those with large cache demands and frequent page accesses. Hence, we filtered applications with high cache demands and non-uniform memory access patterns. From this set, we randomly generate 40 multiprogrammed workloads consisting of four applications. Our results demonstrate that ACTION improves throughput by 4–8% over **state-of-the-art (SOTA)** approaches.

The remainder of this article is structured as follows: Section 2 presents earlier work related to distributed cache partitioning and placement. Section 3 analyzes the memory access patterns of workloads and presents the motivation behind our work. Section 4 details the ACTION hardware and policy mechanisms, Section 5 describes the experimental setup. Experimental results are presented in Section 6.

2 BACKGROUND

This section summarizes previous relevant research on multi-core LLC management. Partitioning and data placement in **non-uniform cache architecture (NUCA)** have been studied in the past for LLC data management in a shared and distributed cache.

In CMPs, LLC organization is either private or shared. In the first scenario, each core has its private fixed size partition. Since these partitions are exclusive, a private LLC organization has the advantage of isolation between cores' cache accesses. On the other hand, applications with small cache requirements waste the available space while others starve due to the partition's fixed size. Shared LLC provides the necessary flexibility for varying cache demands. Jaleel et al. [13] and Bienia et al. [5] demonstrated that application threads share a high percentage of data. Because of the ease of sharing cache blocks between cores, CMPs benefit from shared cache over private. However, as applications compete for the shared cache space, they replace each other's data, resulting in interference misses. In the prior research, cache partitioning techniques have been proposed that provide both benefits of private cache isolation and shared cache flexibility.

Cache Partitioning Techniques (CPT) dynamically partition shared cache between cores [6, 7, 19, 21, 30]. To perform dynamic partition- CPTs use a monitoring mechanism to observe and estimate cache demands for each application. Then, the CPT controller performs block placement according to partitioning decisions.

Qureshi et al. [20] proposed a hardware monitoring approach called **Utility Monitor (UMON)** for cache partitioning. Each core has a UMON circuit that samples the LLC stream and increments the corresponding way's counter value with each hit. Each counter value represents the number of misses saved by assigning a related way to the partition. Miss curves constructed using these counter values represent misses for various partition sizes. The software controller periodically reads the miss curve data, traversing the convex hulls to determine the partition sizes that provide the best overall benefit.

CPT schemes enforce the partition decisions. These schemes implement partition sizes by restricting a cache line's locations. Chiou et al. [6] proposed *way-partitioning* that assigns ways to applications and places an application's cache line in allotted ways. However, its associativity suffers because an application can only use the allotted subset of ways. Sanchez et al. proposed zcache [23] to increase the associativity by increasing the replacement candidates with limited physical ways. Sanchez et al. [24] also proposed the Vantage replacement technique that partitions the high associativity caches like zcache.

Ranganathan et al. [21] and Varadarajan et al. [30] proposed the *set-partitioning* technique that assigns set lines to applications and restricts placement of cache blocks to allotted sets.

Set-partitioning necessitates significant changes to the cache arrays and is incompatible with the hash-indexed cache found in commercial CPUs.

Large monolithic caches suffer from high access latency and energy consumption, as noted by Hardavellas et al. [12]. Cache bandwidth becomes a bottleneck with increasing the number of processors on a single die. As a result, a large LLC is divided into smaller banks, distributed on a chip, and connected by a scalable network. Taylor et al. [28] and Zhang et al. [32] used small cache arrays with faster access times and multiple banks, resulting in higher cache throughput.

A cache request or response must travel the network from the core to the cache bank and vice versa in network-connected tiled cache banks. Hence, cache block access time has two components: cache array access times and network traversal times.

Different banks have different network latencies when accessing a block depending on their proximity to the requesting core. Therefore, implementing uniform access latency for all cache banks will result in the worst-case access latency for all accesses. Instead of having a fixed access latency, Kim et al. [15] modified a single processor cache pipeline so that the lower level cache waits until it receives the LLC block. This technique allows using varying access times, called **Non-Uniform Cache Architecture (NUCA)**. Kim et al. [15] proposed the **Static NUCA (S-NUCA)** approach, which uses fixed line-bank mapping and spreads the data across all banks. The block closest to the nearest bank has the shortest network latency, whereas access to the farthest bank has the longest, making block placement crucial. Prior research focused extensively on NUCA placement techniques to reduce average access latency.

In **Dynamic-NUCA (D-NUCA)**, a cache block can reside in any bank. For example, in Reactive-NUCA proposed by Hardavellas et al. [11], the dynamic aspect stems from distributing ways across banks, with each bank comprising entire set lines. As a result, the D-NUCA placement can place anywhere in the LLC, thereby reducing network latency.

Cho et al. [8] presented a D-NUCA strategy that employs a fixed-line to bank mapping but dynamically places the cache via indirection in virtual to physical memory translation. The **Memory Management Unit (MMU)** page allocator is modified to map physical pages to assigned banks, called page coloring. This approach has a simple lookup mechanism because of fixed mapping. However, page color reconfiguration involves costly page table changes and **Translation Lookaside Buffer (TLB)** shootdowns.

A complicated search mechanism is required since a cache block can reside in any bank in D-NUCA placement. D-NUCA placement procedures proposed by Beckman et al. [3], and Kim et al. [15] perform a search on all the banks allocated to the core to locate a cache block. However, this multicast search imposes a significant load on network and bank throughput. Kim et al. [15] also explored the idea of maintaining a partial tag directory structure to keep track of banks containing cache blocks to avoid the multicast search. As the number of cores increases, so does the size of the directory structure, resulting in high storage costs.

Beckmann et al. [4] proposed Jigsaw, which solves the above problem by locating cache blocks using a small buffer and hash functions. Moreover, it creates logical caches called *shares* and maps pages to shares using buffer and hash functions.

A few prior works have looked into the effect of cache access patterns on NUCA placement. For example, **Reactive-NUCA(R-NUCA)** [11] places and migrates the cache blocks at page granularity level based on read-shared, write-shared, private data access. In R-NUCA, different data classes have different block placement and data replication policies, like placing private data in the nearest bank of requesting core with no replication. However, R-NUCA has less flexibility in mapping cache lines to banks. On the other hand, Whirlpool [18] proposed by Mukkara et al., combines static profiled data with dynamic placement policies. First, Whirlpool statically profiles applications and classifies data into pools. Then, during application execution, MMU dynamically puts

data pools using profiled data, with each data pool having different placements during different execution intervals.

To compare our work, we implemented a version of Jigsaw [4] described by Beckmann et al. In this implementation, Utility-based monitoring [20] and Vantage replacement [24] performs *way-partitioning*. NUCA placement assigns physical cache space to each application based on partitioning capacity using a greedy approach. Until the cache budget determined by partitioning runs out, each core takes turns grabbing a small fixed amount of available physical space. Each core has a **Share Translation Buffer (STB)** with bank and partition numbers. Jigsaw fills STB entries to inform the partitioning and NUCA decisions. The block address' hash value provides the STB entry index, which contains the bank number of the cache block. This STB lookup happens in parallel with the L1 access.

In summary, previous works on large distributed cache have focused on cache requirements and NUCA-aware cache allocation, but have not considered frequency of access to remote cache blocks for page placement. With the exception of Whirlpool [18], previous authors did not consider page access frequency in cache placement to reduce access latency. Whirlpool [18] uses *static* profiling information for placement. By contrast, our work proposes a dynamic solution both for profiling and placement, which reduces network traffic and access latency.

3 MOTIVATION

In this section, we present the motivation behind ACTION design. CPTs identify applications' cache requirements and select partition sizes to provide isolation. NUCA placement seeks to reduce network latency by bringing data closer to the core. However, previous placement solutions are agnostic to the applications' memory access behavior. The following section presents results on memory access patterns to show that some pages are accessed more frequently than others. This information can be utilized to rank order pages for closer placement if the frequency of access can be captured during runtime.

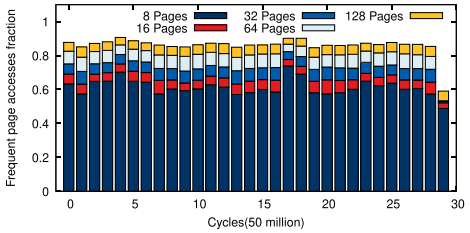

Fig. 1. 464.h264ref accesses to Top-K frequent pages.

3.1 Memory Access Pattern

Application run has phases over which it accesses different parts of the memory. Even during a single application phase, its access pattern is not uniform over the memory references. It accesses a few data structures more often than others. We illustrate this behavior by presenting the page access patterns. Figure 1 shows the page reference pattern for LLC accesses for the benchmark 464.h264ref from the SPEC-CPU 2006 benchmark suite. This pattern shows that only the top eight frequently visited pages account for more than 50% of all page accesses.

3.2 NUCA Placement

As shown in Figure 2(a), typical NUCA placement may distribute the frequently accessed pages across LLC, thereby increasing the number of remote page accesses, which in turn increases network traffic and access latency. Our

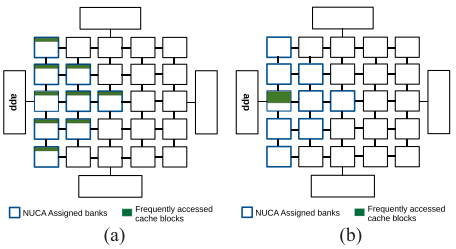

Fig. 2. (a) NUCA placement, (b) Ideal placement.

working hypothesis is that placing these pages closer to the requesting core will ultimately improve performance. In Figure 2(b), we illustrate the ideal placement for the data. However, such a solution will require access frequency tracking, which is a hard problem.

ACTION incorporates a *lightweight, approximate,* solution for tracking page access frequency using simple hardware and adaptively places frequently accessed pages in the nearest banks.

4 PROPOSED WORK

This section presents the ACTION design and explains how it reduces LLC network data movement.

The local bank is the nearest LLC bank to a core in the CMP, whereas the remaining banks are referred to as remote banks. We refer to pages frequently accessed as local pages since ACTION stores them in the local bank. Also, pages formerly classified as local pages and placed in the local bank which are no longer frequent become remote pages.

Figure 3 shows the complete overview of the ACTION. When there is an L1 request, ACTION checks the page number in the **Frequent Page List (FPL)**. On L1 miss, if this page belongs to FPL, L2 request will be sent to the local bank of the core otherwise to the remote bank. This procedure is described in detail in the Section 4.2.

ACTION monitor records every L2 access and tracks top-K frequent pages in the **FTB (Frequency Tracking Buffer)**. On an L2 request, if the page belongs to the FTB, monitor increments the page FTB count value and also **Counting Bloom Filter (CBF)** counter values. If there is a page miss in FTB, the incoming page number access count from CBF, a replacement candidate, and its access count from FTB are computed. If the incoming page access count is greater than the replacement candidate, the incoming page replaces the FTB entry and updates its count value. Finally, incoming page entries are incremented in the CBF.

Apart from monitoring and placement, as shown in Figure 3, ACTION features a reconfiguration mechanism to dynamically adapt to program phase changes. In the following subsections, we explain these components in detail.

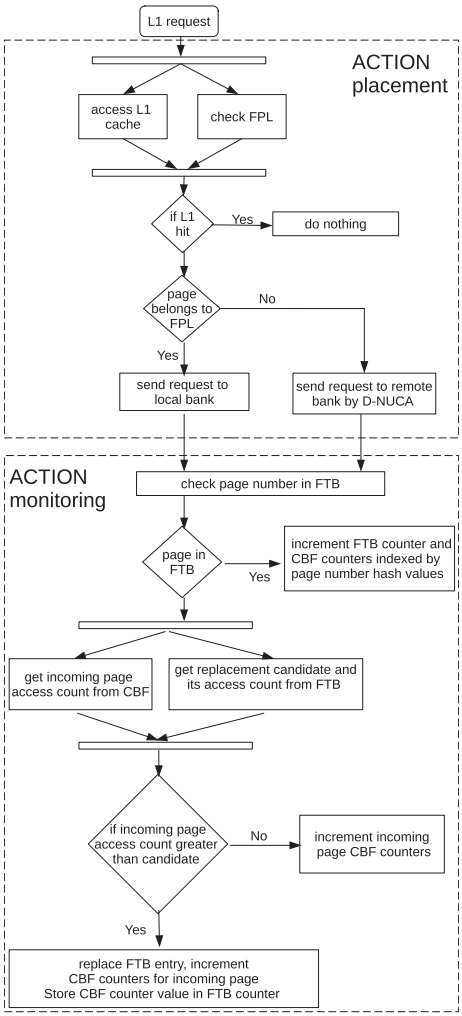

Fig. 3. ACTION overview.

4.1 ACTION Monitoring

ACTION monitor, illustrated in Figure 4, identifies the top-K local pages of the application run. It maintains a list of local pages. ACTION checks if the accessed page belongs to the local page list on every LLC access by comparing the incoming page number's access count to a local page access count. We need access counts for every accessed page to keep track of local pages. For this purpose, we employ a Counting Bloom Filter, which

is an approximate counting structure that records and counts every LLC access and page access. Einziger et al. proposed TinyLFU [9] in which a Counting Bloom Filter is used for identifying frequent items for cache replacement.

The Frequency Tracking Buffer (FTB) is a small set-associative list of page numbers. FTB keeps track of the top-K local pages at all times during application execution. We update FTB with a simple **LFU (Least Frequently Used)** mechanism on every LLC access.

We check if the accessed block's page number is present in FTB. If the page number is in FTB, we record the access in CBF. Else a replacement candidate is selected from FTB by LFU policy. If the incoming page access count is greater than the candidate access count, it replaces the candidate in FTB.

To implement the LFU technique, we need access counts of all the application pages. Unfortunately, an application accesses many pages during execution, making it difficult to keep track of all page accesses. Therefore, we use CBF, a minimal hardware approximate counting structure that keeps track of page access counts. In the next section, we explain in detail the operation of CBF in the ACTION monitoring mechanism.

4.1.1 Counting Bloom Filter (CBF). A CBF, shown in Figure 5, is a Bloom Filter in which each entry is a counter instead of a single bit. CBF records every LLC access, and CBF counters track the LLC access count of a page number. A CBF supports insertion and estimation functions.

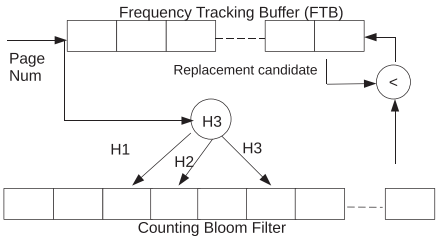

Fig. 4. ACTION Monitor.

Insertion: Figure 6(a) shows how CBF inserts a page LLC access. On an LLC access, CBF computes hash values of the page number. The CBF is then indexed using hash values, and the corresponding counter values are incremented.

Access Frequency Estimation: During page replacement in FTB, we have to estimate the incoming page access frequency. The incoming page number is hashed to compute the indices to the counters; the minimum value among these counters corresponds to the approximate frequency count, as shown in Figure 6. Based on hash function indices, multiple pages share array counters. However, no two pages share all hash indices except for hash collisions. This Bloom filter property is the basis for approximating the access frequency.

CBF is a minimal hardware counting structure with fast operations. With CBF and FTB, we collect information about the top-K frequent pages during the application's run. We use this information to place cache blocks in the local bank. In the following subsection, we describe the placement procedure of the ACTION.

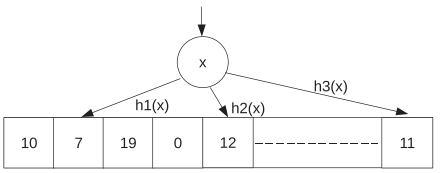

Fig. 5. ACTION Monitor.

4.2 ACTION Placement

ACTION placement mechanism is responsible for NUCA and access behavior aware placement. We applied the ACTION placement procedure along with JIGSAW [4], shown in Figure 7. JIGSAW partitions the cache space among applications based on their requirements and assigns the physical cache among banks based on the proximity to the core. Jigsaw places cache blocks in the assigned bank partitions uniformly. ACTION uses a **Frequently-accessed Page List (FPL)** that

puts the frequent page cache blocks in the local bank of the application core utilizing memory access behavior captured by the ACTION monitor.

FPL contains local pages that are to be placed in the local bank. So, on L1 access, we search FPL parallel to L1 lookup to find if the cache block belongs to a local page. If the cache block belongs to an FPL page, we request the local bank. Otherwise, we send it to the **Static Translation Buffer (STB)** bank.

We modified cache controller logic to send frequent page LLC requests to the local bank for implementing the ACTION placement procedure.

Application's memory access pattern changes over its run. As a result, frequent pages change over different application phases. To capture these phase changes, ACTION migrates pages in LLC dynamically. Below, we describe the ACTION reconfiguration mechanism in detail.

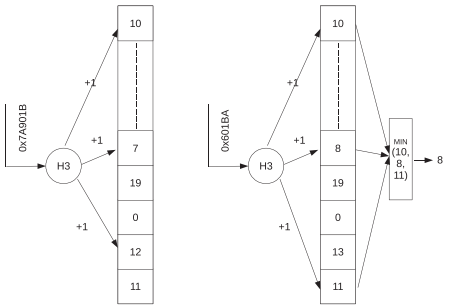

Fig. 6. (a) CBF insertion, (b) CBF estimation.

4.3 ACTION Reconfiguration Mechanism

ACTION places FPL pages in the local bank. ACTION's FTB keeps track of frequent pages from the LLC access stream. For every fixed interval, called Reconfiguration interval, we compare FTB and FPL pages. If there is a change in the application's access pattern, FTB and FPL pages disagree on some entries. The FPL pages which are not in FTB are no longer frequent and need to move them to remote banks. New FTB pages not present in FPL are the newly identified pages in the recent interval and need to migrate to the local bank.

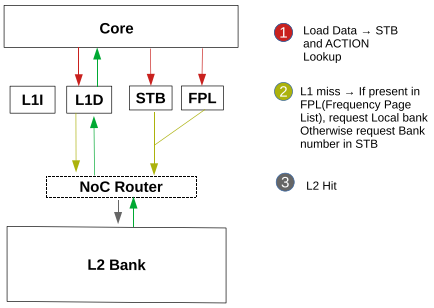

Fig. 7. ACTION LLC flow.

We assume a similar method for migration as in the base D-NUCA model. Specialized hardware walks through each bank cache array and invalidates the cache blocks that belong to local and remote pages. It is generally recognized that cache miss penalty is more severe than remote cache block access latency. The reconfiguration interval in D-NUCA was optimized to maximize gain from reconfiguration taking losses from cache miss penalty into account. Since ACTION is built atop D-NUCA, we use the same reconfiguration interval of 50M cycles. When the invalidated cache block is accessed in the next interval of the application run, the cache controller places it in the new location. Hence, reconfiguration has a penalty of compulsory misses and invalidation overhead with hardware. Because of the large reconfiguration interval, cycles saved by ACTION page migration outweigh penalty cycles.

We copy FTB entries to FPL to begin accessing new local page cache blocks from the local bank from the next interval. Finally, we clear FTB entries and reset CBF counters to monitor the next interval of the application run.

5 EXPERIMENTAL SETUP

To illustrate the ACTION's benefit in large LLC systems, we simulated a CMP configuration shown in Table 1. ZSim [25], an Intel Pin-based trace-driven x86-64 microarchitectural simulator, is used to run simulations.

Table 1. Configuration of Simulated 4 Core CMP

Cores	4 cores, x86-64 ISA, in-order IPC = 1 except on memory accesses, 2 GHz
L1 Caches	32KB, 8-way set-associative, split D/I, 4-cycle latency
L2 Caches	512KB per bank, 4-way 52-candidate zcache, inclusive, 9-cycle latency
Coherence protocol	MESI, 64 B lines, in-cache directory, no silent drops; sequential consistency
NUCA NoC	5x5 mesh, 512-bit flits and links, X-Yrouting, 5/10/15/20 cycle hop delay
Memory	1MCU, 1 channel/MCU, 120 cycles zero-load latency, 12.8GB/s per channel

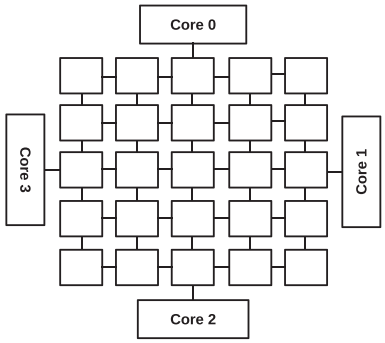

Fig. 8. Simulated 4 core CMP.

We use four simple x-86 **Instruction Set Architecture (ISA)** in-order cores with IPC=1, except on memory accesses in the simulated configuration. Memory hierarchy consists of a split L1 cache, a second-level large LLC, and dram memory. A cache miss in private L1 sends a request to the shared LLC.

5.1 Last Level Cache

Similar to Oracle Sparc [1], CMP configuration has a large centralized LLC. The LLC has a total capacity of 12.5MB distributed across cache banks. There are 25 cache banks arranged in a 5x5 mesh **network-on-chip (NoC)**, as shown in Figure 8. Each node in the mesh has a cache of 512 KB size and a router to send /receive network messages. We used the network timing model from Minnow [31] proposed by Zhang et al. In this model, as the network size increases, average access time for remote banks also increases, thereby increasing the gap between local and remote access times. Since the benefit of ACTION comes from arbitrage of the disparity between local and remote access times, ACTION is expected to perform better for larger networks. The network has a flit size of 512 bits and uses X-Y routing to direct the network traffic. A cache block request from one of the core's private L1 cache travels to the shared LLC location through the network.

Each LLC bank array is Zcache [23] with four physical ways and 52 candidates for cache line replacement. Thus, Zcache offers higher associativity required for D-NUCA without increasing physical array ways.

Each bank array comprises all the set lines of the entire cache. Hence, any bank can contain a cache line, implementing D-NUCA.

Vantage replacement policy [24] partitions the shared LLC between applications. Jigsaw [4] makes the partition decision based on capacity requirements and physical cache allocation to applications based on proximity. **STB (Static Translation Buffer)** has bank numbers and partition numbers and gives cache block location for D-NUCA.

MESI coherence protocol is implemented with an in-cache directory. Each cache line is of size 64B.

Off-chip main memory accesses happen through a **memory control unit (MCU)** with a single channel. Each off-chip access has a latency of 120 cycles. We used the memory management architecture from Liu et al. [16].

5.2 Workload and Performance Metric

We generated 40 multiprogrammed mixes from the SPEC CPU 2006 benchmark suite for workloads. Each workload runs for 1 billion instructions after skipping 20 billion cycles. Each workload

simulates until all four applications in the mix complete 1 billion instructions. The results include only the first 1 billion instructions for each application, ensuring that each executes at least 1 billion. We use NUCA + Cache Partitioning Jigsaw for our base model. We reported the performance gain by the ACTION model in terms of **harmonic mean (HMean)** IPC gain.

$$IPC_{hmean} = \frac{\text{Number of Applications}(n)}{\sum_{i=0}^{n} IPC_{\text{base},i}/IPC_i} \tag{1}$$

IPC values from Jigsaw [4] is used as IPCbase, i used to normalize ACTION IPC_i values. The performance gain gives Hmean of normalized IPC compared to the base model.

6 EVALUATION

This section summarizes the ACTION performance over state-of-the-art D-NUCA and presents the results. We first present results to show performance gain by migrating frequent pages. Next, we analyze the ACTION model sensitivity to parameters. Finally, we compare the ACTION model results with the oracle model for page migration.

6.1 Migrating Frequent Pages

Workload generation: We selected applications from the SPEC-CPU 2006 benchmark suite to generate workload mixes. The following section describes the procedure to identify applications from the benchmark suite that benefit from ACTION in large LLC systems.

Application sensitivity to cache size: SPEC-CPU 2006 benchmark suite has applications that have high and low cache demands. Applications with small cache requirements will not benefit from dynamic cache reconfiguration as the initial placement alone may be good enough, where all frequently accessed pages fit within local bank of size 512KB (baseline). Applications whose demand exceeds 512KB, must place some pages in remote banks. To identify cache-sensitive applications, we simulated the applications in single-core, single-bank configuration with LLC sizes varying from 512KB to 8MB. Table 2 shows the performance gain of target applications for various LLC sizes compared against baseline of 512KB. We narrowed the target pool of applications based on cache size sensitivity in Table 2. The selected pool of applications are shown in Figure 10.

Applications with frequent pages: Applications have varying page access patterns. For some applications, a few pages are accessed more frequently than others as shown in Section 3. These applications are the most likely beneficiaries of dynamic page migration. To identify such applications with frequent page access characteristics, we first simulated each application in single-core configuration. Figure 9(a) shows the total number of pages accessed by each application in intervals of 50 million cycles. Figure 10 shows the fraction of accesses that happen to the top-8, 16, 32, 64, 128 pages compared to the total accesses. For applications such as *416.gamess, 458.sjeng, 444.namd, 447.dealII, 445.gobmk, 400.perlbench, 464.h264ref, 401.bzip2, 471.omnetpp, 403.gcc, 454.calculix* most of the accesses made are to the frequent pages. For example, *439.mcf* accesses an average of 20,000 pages, but most of the accesses happen to top-128 pages. Hence, in large heterogeneous LLC systems, placing these frequent page cache blocks in the nearest bank reduces network latency by reducing remote accesses.

We chose applications with both high cache demand and frequent page access patterns for our simulations as they are most likely to benefit from dynamic page migration to the nearest bank. Intersection of applications from Table 2 and Figure 10 are selected for this purpose. The chosen benchmarks are *401.bzip2, 435.gobmk, 445.dealII, 403.gcc, 464.h264ref*, and *434.zeusmp*.

Next, we ran each application on a single-core, multiple cache-bank configuration as shown in Figure 12 with a reconfiguration interval of size 50 million cycles. In the four core configuration, as shown in Figure 8, each application starts with six banks. Table 3 shows the fraction of remote

Table 2. Applications Normalized Gain with Cache Capacity

Benchmark	512KB	1MB	2MB	4MB	8MB	16MB
450.soplex	1	1.0943	1.29031	1.71199	2.68193	3.8903
471.omnetpp	1	1.00019	1.24878	2.50451	2.50455	2.50455
482.sphinx3	1	1.02025	1.0303	1.04487	1.9077	2.26118
403.gcc	1	1.56567	1.892	1.93788	1.93903	1.93903
473.astar	1	1.35238	1.6842	1.7105	1.75151	1.89734
454.calculix	1	1.51013	1.5772	1.58131	1.59191	1.60323
429.mcf	1	1.00349	1.00836	1.1667	1.36878	1.59781
470.lbm	1	1.00001	1.00002	1.00062	1.01085	1.5519
401.bzip2	1	1.15496	1.35515	1.44687	1.46826	1.46826
456.hmmer	1	1.05502	1.2	1.31472	1.33364	1.33364
437.leslie3d	1	1.06389	1.07315	1.13603	1.18518	1.2656
436.cactusADM	1	1	1.00001	1.05056	1.1795	1.22885
464.h264ref	1	1.0202	1.15029	1.17566	1.17567	1.17567
434.zeusmp	1	1.02448	1.04008	1.06714	1.10182	1.16148
410.bwaves	1	1.00739	1.05852	1.09051	1.10532	1.12406
459.GemsFDTD	1	1.00268	1.00647	1.10904	1.10993	1.11163
447.dealII	1	1.03534	1.09235	1.09625	1.09625	1.09625
400.perlbench	1	1.00903	1.01707	1.02148	1.06334	1.06334
445.gobmk	1	1.03277	1.04578	1.04979	1.05327	1.05831
444.namd	1	1.00671	1.00939	1.0097	1.0097	1.0097
433.milc	1	1	1.00021	1.00173	1.00299	1.00848
458.sjeng	1	1.00143	1.00251	1.00381	1.00545	1.00696
462.libquantum	1	1	1	1	1	1
416.gamess	1	1	1	1	1	1
483.Xalan	1	1	1	1	1	1

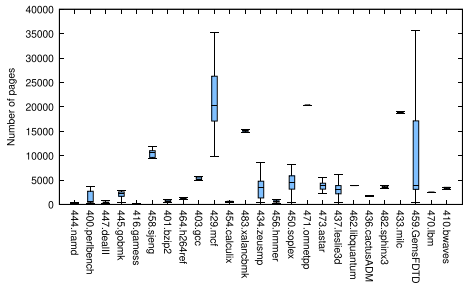

Fig. 9. Pages accessed by each application.

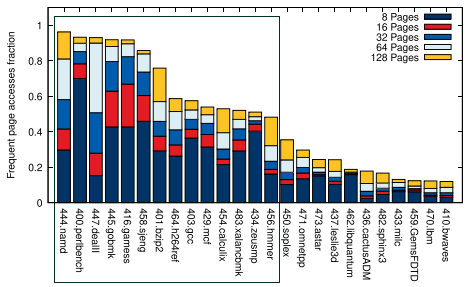

Fig. 10. Access to top-k frequent pages.

accesses as a result of ACTION. As can be seen from this table, *401.bzip2, 435.gobmk, 445.dealII*, and *403.gcc* exhibit significantly reduced remote accesses. The reduction in number of remote accesses results in IPC gain as shown in Figure 11. Hence, these applications are the target for our multiprogrammed workload generation.

Multiprogrammed workload generation: We generated 40 multiprogrammed workloads from the above application pool. For each workload, an application is chosen randomly from *401.bzip2, 435.gobmk, 445.dealII, and 403.gcc* with replacement. Hence, a workload mix might have multiple instances of the same application.

6.2 Sensitivity to Parameters

The ACTION model has two sets of parameters: FPL parameters to decide the number of frequent pages that ACTION will migrate, and the CBF parameters that determine accuracy of the

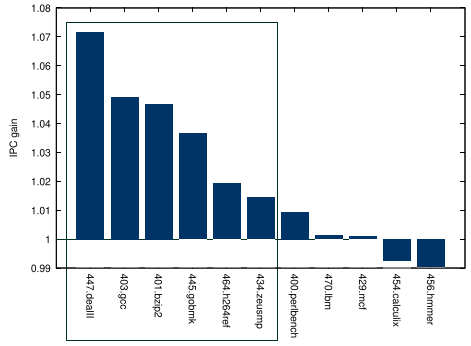

Fig. 11. Application gain with page migration over D-NUCA.

Table 3. Fraction of Remote Bank Accesses to Total LLC Access

Benchmark	D-NUCA	Page Migration
447.dealII	0.83	0.20
401.bzip2	0.80	0.41
445.gobmk	0.81	0.58
403.gcc	0.81	0.59
464.h264ref	0.81	0.66
434.zeusmp	0.82	0.77

monitor. We describe the experiments and results in the following subsections for obtaining these parameters.

FPL parameters: FPL maintains the most frequently accessed page numbers over the application's run. FPL in turn has two parameters: *reconfiguration interval* to maximize performance gain with minimum overhead, and *number of frequent pages* to be migrated to the local bank.

(a) Reconfiguration interval: D-NUCA reduces capacity and interference misses by allocating optimum cache capacity to applications. However, a cache miss has a higher penalty than accessing a cache block from a remote bank. Hence, we use the same reconfiguration interval of 50 million cycles as base model D-NUCA so that our implementation uses optimum interval that reduces misses than remote bank accesses.

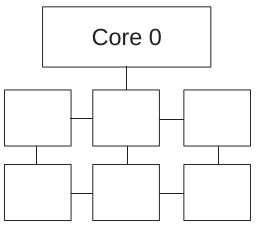

Fig. 12. Single core configuration with LLC mesh network.

(b) Page list size: In the configuration Figure 12, a maximum of 128 pages can fit in the local bank of capacity 512KB. Figure 13 shows the applications gain due to frequent page migration in the banked cache. Figure 13 presents the performance comparison between placing frequent pages in the local bank to uniformly distributing the cache blocks across banks. In D-NUCA, a hash function places the cache blocks across cache banks uniformly based on block address.

By placing frequent pages in the local bank, ACTION reduces remote bank access. However, migrating page blocks that are not as frequent as uniformly placed cache blocks increases remote block access. Furthermore, reconfiguration has a penalty of compulsory misses due to invalidations. Hence, we need to migrate an optimum number of frequent pages with a net advantage of saved network cycles over the reconfiguration penalty. From the Figure 13, migrating 64 and 128 frequent pages to the local has the maximum gain for all the applications.

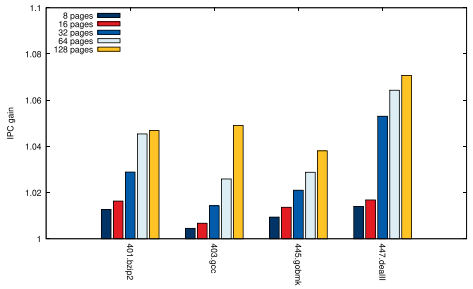

Fig. 13. Application gain with frequent page migration.

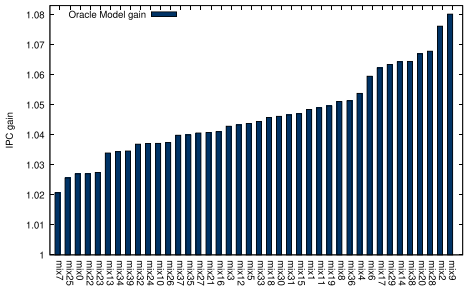

Fig. 14. Application mixes gain with page migration.

Oracle gain: We simulated 40 workloads on four core mixes. We migrated the top-64 frequent pages to the local bank for 50 million intervals. Figure 14 shows the performance gains for each workload mix compared to the D-NUCA base mosedel. Figure 14 represents the maximum gain achieved for each mix by page migration.

The following sections explain the procedure to determine ACTION hardware parameters that achieve performance closest to the Oracle Model.

CBF parameters: ACTION monitoring hardware uses CBF for counting page accesses, with FPL using the LRU policy to keep only top-k frequent pages during the application run. CBF records every page's LLC access and counts page accesses. FPL uses CBF page counts to make LFU replacement decisions on a page LLC access.

Large CBF has higher accuracy but also has a significant hardware overhead. We present experiments to determine CBF parameters with minimum hardware overhead without losing the performance gain.

Counters in CBF have a fixed width W that saturates after $(2W-1)$ insertions. ACTION monitors LLC stream by CBF recording all the LLC accesses. Figure 15 shows the total number of LLC accesses per interval. Recording the large number of LLC accesses into the CBF saturates counters, reducing frequent page prediction accuracy.

The following parameters are used in the context of CBF:

Monitoring interval: Interval length to record LLC accesses and track frequent pages.

Window size: Periodic interval for refreshing CBF.

CBF Parameters: Counter width, m = CBF array size/ number of entries, n = Maximum number of CBF insertions, k = Number of hash functions

The main objective for finding the optimum parameters is to reduce the hardware size without losing much of the performance gain.

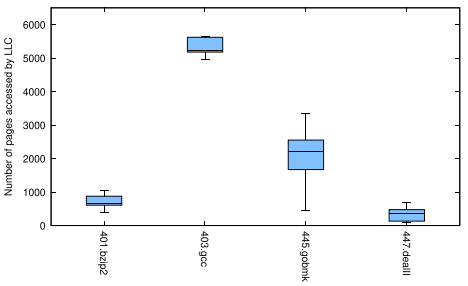

Fig. 15. Number of pages accessed by each application per phase.

Monitoring interval: Instead of monitoring the entire execution phase, recording LLC accesses in a small interval at the end of each interval accurately predicts the frequent pages. Monitoring over a small interval reduces the number of insertions into the CBF but affects the prediction accuracy. Thus, the ideal value is a short monitoring interval that does not significantly lose prediction accuracy. Since performance improvement is a proxy for prediction accuracy, we measure ACTION performance over various intervals to determine ideal monitoring interval. This has been shown in Figure 16. It may be observed from this figure that monitoring 3 million cycles at the end of the interval captures most of the gain for the targeted applications.

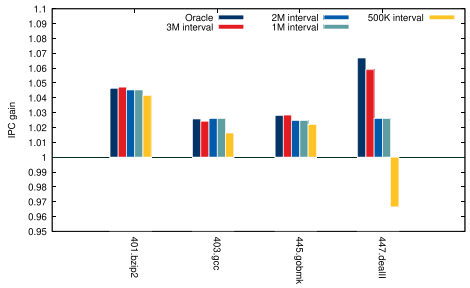

Fig. 16. ACTION prediction accuracy with different monitoring intervals.

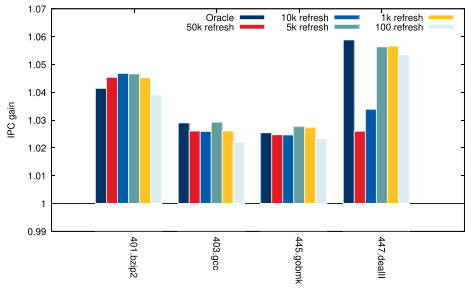

Fig. 17. ACTION prediction accuracy with different refreshing window size.

Table 4. Application IPS Gain with Different Counter Widths of CBF

Benchmark	Counter width = 31	Counter width = 3
401.bzip2	1.050	1.046
403.gcc	1.027	1.028
445.gobmk	1.029	1.025
447.dealII	1.064	1.061

Refresh window size: CBF counters saturate after $(2^W - 1)$ insertions and predict wrong page access counts, where W is the counter width. We use a refresh mechanism that resets the counters for every fixed interval to avoid saturation.

Figure 17 shows the prediction accuracy for different refreshing window sizes. For an FPL size of 64, we simulated for refresh window sizes 2*(FPL size) to 1K*FPL size. From these simulation results, we can observe that frequent refresh operations do not affect the ACTION performance. Furthermore, choosing a smaller refresh window helps to reduce the CBF array size since CBF insertions (CBF n parameter) become small with smaller window sizes. Therefore, we choose a refresh window size of 512, which is not too small for frequent refreshes and not too big to increase the CBF array.

Counter width: In the FTB, we track top-64 frequent pages of the LLC accesses. For our page migration, we need top-64 pages but not their ranking based on frequency. So for a refresh window size of 512 insertions, the counter should have a maximum value of $W/64$.

A comparison of performance gain from ACTION with the counter sizes of 3 and 31 is shown in Table 4. We can observe almost similar IPS gain with a counter of 3 and 31 bits from these results.

CBF parameters *m, n, k*: As we refresh CBF for every 512 insertions, our CBF insertions value n is 512. We simulated ACTION with CBF array size $m = 1K, 2K$ and $4K$ for number of hash functions 4, 6, 8. From Table 5, we observe that $m = 4K$ has the highest performance gain because of the bigger array size. However, we selected $m = 2K$ since it captures almost all the gain with half the array size.

Our ACTION model tracks top-64 pages in an interval of 3 million cycles and refreshes CBF counters for every 512 insertions. A CBF array of size 2048 and 8 hash functions with each array entry has a 3-bit counter. ACTION adds 768B in CBF entries per core. In the following section, we compare the ACTION performance gain with Oracle mode.

Figures 18–21 show the frequent page prediction accuracy for bzip2,gcc,gobmk and dealII applications that CBF achieves.

Table 5. Application IPS Gain with m = 2K and k = 4, 6, 8 of CBF

Benchmark	3M interval	m = 4K, k = 4	m = 2K, k = 6	4m = 1K, k = 8
401.bzip2	1.046883	1.044553	1.044539	1.044539
403.gcc	1.028282	1.017562	1.017586	1.017586
445.gobmk	1.025516	1.020525	1.020525	1.020525
447.dealII	1.061154	1.057705	1.057705	1.057705
Mean	1.040	1.035	1.035	1.035

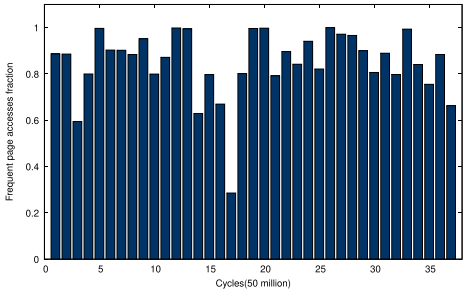

Fig. 18. bzip2 CBF prediction accuracy.

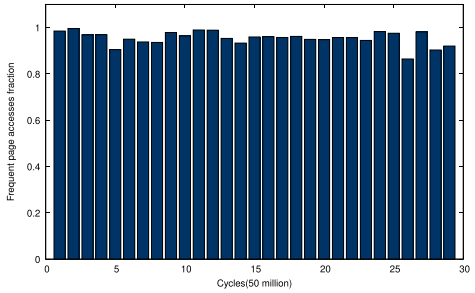

Fig. 19. gcc CBF prediction accuracy.

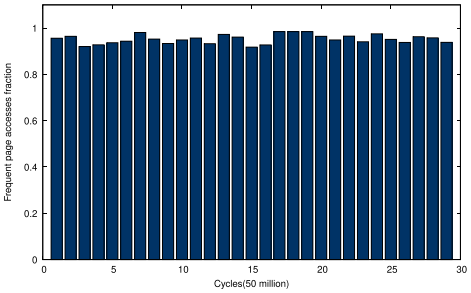

Fig. 20. gobmk CBF prediction accuracy.

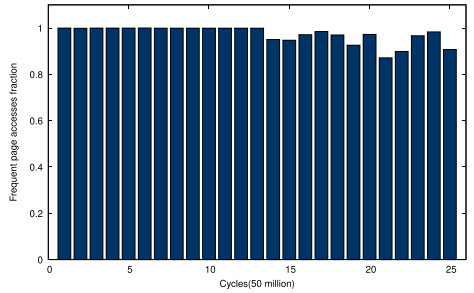

Fig. 21. dealII CBF prediction accuracy.

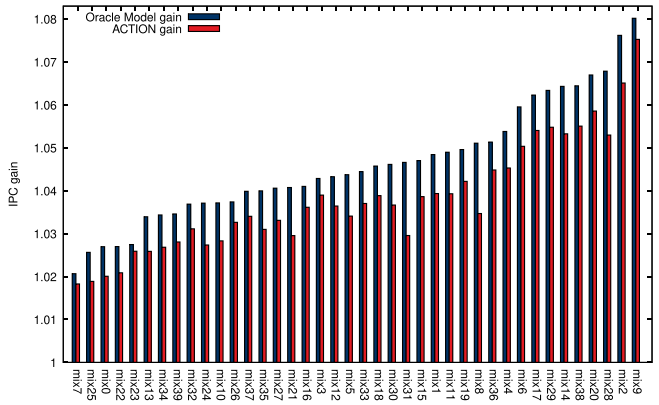

Fig. 22. ACTION performance gain.

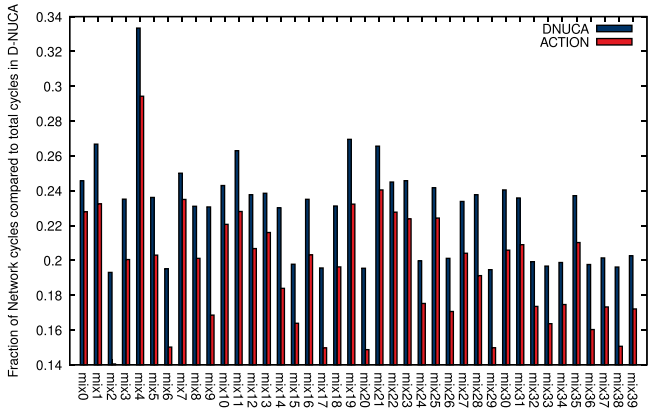

Fig. 23. Network cycles saved by ACTION.

6.3 ACTION Performance

Figure 22 shows the performance of ACTION compared to the Oracle model. ACTION achieves an average of 3.8% IPS improvement over D-NUCA, whereas Oracle achieves 4.8% over D-NUCA, which represents the ceiling of possible gain.

Since ACTION reduces the frequency of request injections into the LLC network, it improves not only the access time of locally placed cache blocks but also the access time of remote cache blocks by lowering the overall network load. Figure 23 shows the ACTION model's fraction of cycles spent in the network traversal compared to the D-NUCA.

Reconfiguration penalty for invalidations: In these experiments, ACTION performs reconfiguration for every 50 million cycles. Like the base D-NUCA, specialized hardware walks through each array invalidating the migrating cache blocks. If a core accesses a migrating cache block, its execution enters into a quiesced state until the invalidation is completed.

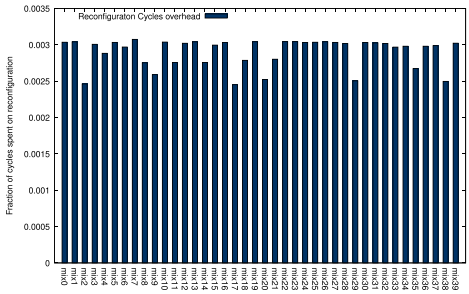

Fig. 24. Fraction of cycles spent invalidating cache blocks for page migration.

Hence, the reconfiguration penalty involves both the cycles spent for invalidations and the compulsory misses of migrated cache blocks. Figure 24 shows the mean fraction of cycles spent in the bulk invalidation process by all four cores for each workload mix.

6.4 Stacked DRAM

Caches are becoming increasingly distributed and non-uniform to fulfill the capacity needs of rising core count in CMPs. Previous research explores the idea of heterogeneous technologies with **Static Random Access Memory (SRAM)** and DRAM cache bank integration [10, 17, 26, 27]. Typically, processors and DRAM are manufactured using different process technologies. For example, DRAM in LLC may be implemented in two ways: (*i*) using a specialized manufacturing process that allows *embedded DRAM* in a monolithic process or (*ii*) manufacturing DRAM and processor separately, but integrating them in a package by stacking one die above another. Each technology has its pros and cons, which are beyond the scope of this article. Regardless of the

Table 6. Workload IPS Gain of ACTION with Different Hop Latencies

Remote cache latencies	Range(percentage)	Mean(percentage)
40 cycles	4.15–18.90	10.5
50 cycles	4.80–21.14	11.61
60 cycles	5.4–13.04	13.05
70 cycles	5.71–26.70	14.33

technology, while DRAM offers increased capacity, it also has limitations, such as higher access latency and limited bandwidth.

Prior works [14] hide the extra latency by checking SRAM and DRAM cache in parallel. While it improves performance, it also adds unnecessary access, increasing the network load. Jenga [29] uses an adaptive cache hierarchy allocation based on application needs.

DRAM cache banks have limited bandwidth which limits access. Jenga [29] monitors bank access latencies and places data incrementally in the banks with the lowest access latency. However, these works do not consider the application access behavior for block placement explained in Section 3.

For a stacked DRAM cache of size 128MB to 2GB, access latency varies from 42 cycles to 74 cycles [29]. Table 6 shows the ACTION significant performance gain with various hop cycle latencies ranging from 40 to 70 cycles for the target multiprogrammed workloads.

7 CONCLUSION

Applications exhibit non-uniform access frequency across the address range. As a result, a core running an application accesses a few pages more frequently than others. In CMPs with a large non-uniform cache, placing frequently accessed pages in the nearest cache location has performance benefits.

ACTION, proposed in this article, is a software-hardware mechanism that dynamically migrates and places the frequently accessed page blocks in the nearest cache bank. ACTION hardware features a Counting Bloom Filter (CBF) that records memory access information by tracking page access counts, making it possible to identify the frequently accessed pages. ACTION places these frequent pages at the nearest LLC location of the affine core in concert with D-NUCA placement. As the frequently accessed pages vary during the execution, ACTION updates this information continuously and migrates cache blocks dynamically after a monitoring period.

Our simulation results indicate that ACTION can improve multiprogrammed applications performance on a CMP from 1.8%–7.5% with an average gain of 3.8% over state-of-the-art (SOTA) D-NUCA. The gain in performance is attributable not only to the reduction in access latencies but also to the reduction in the number of remote access requests. Furthermore, a reduction in the number of network requests reduces network traffic, which in turn enhances the response time of the NoC.

Finally, ACTION exhibits significant performance improvement with increasing LLC heterogeneity. Since DRAM-based LLCs offer large capacities compared to SRAM-based LLCs but with substantial access latency differential and limited bandwidth. ACTION reduces remote access, thereby relieving pressure on the limited bandwidth and reducing average access latency, making it ideal for DRAM-based cache.

REFERENCES

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, et al. 2015. M7: Oracle's next-generation sparc processor. *IEEE Micro* 35, 2 (2015), 36–45.

[2] Rajeev Balasubramonian, Viji Srinivasan, Sandhya Dwarkadas, and Alper Buyuktosunoglu. 2003. Hot-and-cold: Using criticality in the design of energy-efficient caches. In *International Workshop on Power-Aware Computer Systems*. Springer, 180–195.

[3] Bradford M. Beckmann and David A. Wood. 2004. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 319–330.

[4] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 213–224.

[5] Christian Bienia, Sanjeev Kumar, and Kai Li. 2008. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 47–56.

[6] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. 2000. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference*. 416–419.

[7] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 357–368.

[8] Sangyeun Cho and Lei Jin. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 455–468.

[9] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[10] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. 2014. Haswell: The fourth-generation Intel core processor. *IEEE Micro* 34, 2 (2014), 6–20.

[11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 184–195.

[12] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. 2007. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*.

[13] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. 2006. Last level cache (llc) performance of data mining workloads on a CMP-a case study of parallel bioinformatics workloads. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*. IEEE, 88–98.

[14] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 404–415.

[15] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 211–222.

[16] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[17] Gabriel H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 453–464.

[18] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 113–127.

[19] Moinuddin K. Qureshi. 2009. Adaptive spill-receive for robust high-performance caching in CMPs. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 45–54.

[20] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[21] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 214–224.

[22] Karl Rupp, M. Horovitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. 42. Years of microprocessor trend data. *by Karlrupp. Net.[Online* (42).

[23] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 187–198.

[24] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. 57–68.

[25] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 475–486.
[26] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro* 36, 2 (2016), 34–46.
[27] Jeff Stuecheli. 2013. POWER8. In *Hot Chips Symposium*. 1–20.
[28] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2 (2002), 25–35.
[29] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 652–665.
[30] Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni, and Donald Newell. 2006. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 433–442.
[31] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices* 53, 2 (2018), 593–607.
[32] Michael Zhang and Krste Asanovic. 2005. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 336–345.

Received 25 March 2022; revised 15 September 2022; accepted 10 November 2022