Searching Mechanisms for Dynamic NUCA in Chip Multiprocessors

A Thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

Kartheek Vanapalli
(124101040)

under the guidance of

Dr. Hemangee K Kapoor

to the

Department Of Computer Science And Engineering

Indian Institute Of Technology Guwahati
Guwahati - 781039, Assam
May 2013

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Searching Mechanisms for Dynamic NUCA in Chip Multiprocessors**" is a bonafide work of **Kartheek Vanapalli** (**Roll No. 124101040**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Hemangee K Kapoor**
Associate Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Assam - 781039, India.

To My Family Members and Friends

Acknowledgements

I would like to take this opportunity to express my gratitude to my advisor **Dr. Hemangee K Kapoor** for her valuable guidance. without her this work would not be possible. I would like to thank her for her continuous support, patience, motivation and knowledge. It has been a really great learning experience for me to work under her guidance.

I would like to express my sincere thanks to Mr. Shirshendu Das research scholar, Dept. of CSE and Mr. Shounak Chakraborty research scholar, Dept. of CSE with whom I had several useful discussions. I would like to thank the Deptartment of CSE, IIT Guwahati for providing all learning resources. Special thanks to my family members for their support and blessings.

I am thankful to all my friends, seniors and teachers who have always helped me. Also, I would like to thank Ministry of Human Resource Development (MHRD), Government of India for funding my M.Tech course.

Kartheek Vanapalli

Abstract

Rapid growth in the cache sizes of Chip Multiprocessors (CMPs) will lead to increase in wire-delays and unexpected access latencies. Non-uniform access latencies to on-chip caches led to Non-Uniform Cache Architecture (NUCA) design, but to get data proximity features we need to map data blocks dynamically to banks and migration of data blocks. Because of dynamic mapping and migration schemes employed by D-NUCA we can't keep track of data blocks. In D-NUCA mapping and migration schemes, increasing data proximity which reduces access latencies, but if we don't have good searching mechanism for a data block in NUCA cache we strain ourselves in searching whole cache in case of a miss. So in order to get maximum benefit from D-NUCA we need an efficient searching mechanism.

In this paper we discussed existing searching mechanisms and highlighted overheads and limitations to those algorithms. We proposed a novel searching algorithm for D-NUCA designs in CMPs, which is called HKState-NUCA (Home Knows where to find data and its state within the NUCA cache). This algorithm provides fast and power efficient access to data lines than many of the existing searching mechanisms. We have shown that using HKState-NUCA as data search mechanism in a D-NUCA design reduces search requests (hop count) about 50.89%, and achieves an average performance improvement of 7.04% compared to HK-NUCA searching algorithm [13] by introducing 0.8% space overhead.

Contents

List of Figures

Chapter 1

Introduction

1.1 Domain of Research

Rapid improvements in integrated circuit technology led to micro architectural innovation. But, due to parallelism, circuit limitations and high power consumptions of microprocessors motivates to efficiently use the existing silicon resources . It is very costly to design a single large processor by exploiting the above issues, which led to chip multiprocessors (CMPs) [7]. CMPs consists of multiple processors integrated on a chip as shown in Figure 1.1.These processors in CMPs run at a lower clock rate than the microprocessors which mitigate the power consumption. CMPs can execute multiple applications simultaneously by utilizing Thread-Level Parallelism to improve overall performance of applications, parallelism becomes very critical for improving performance nowadays.

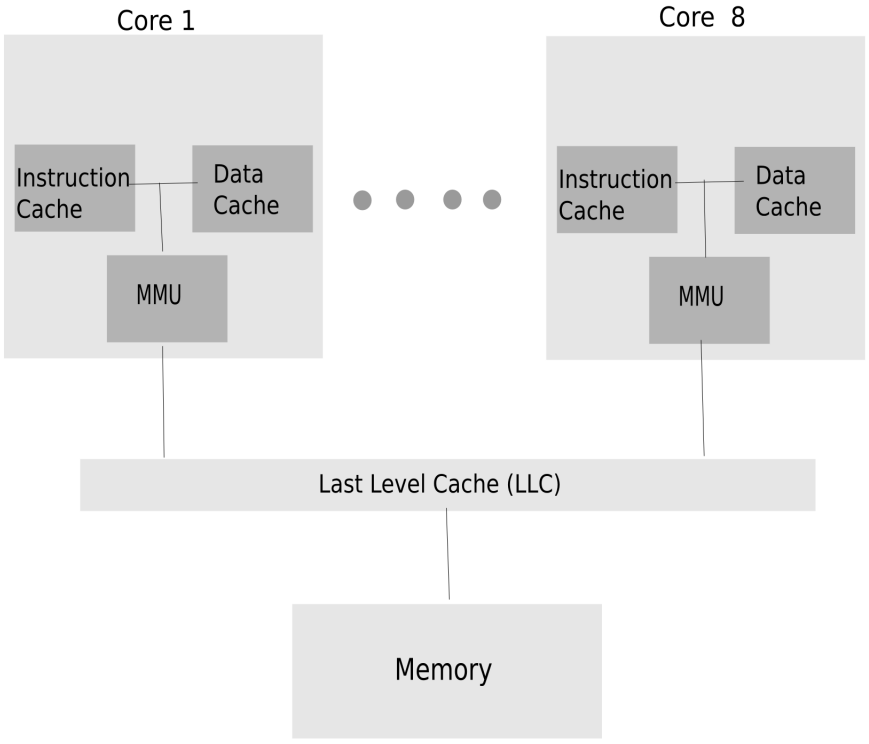


Fig. 1.1 Basic CMP organization.

For any multiprocessor [20] memory system plays very important role in improving performance. CMP architecture typically consists of very large and complicated cache hierarchies, recent CMP architectures from Intel and AMD consists of very large Last Level Cache (LLC) almost 50% of the on-chip area [18]. Future applications like big data processing will require more number of cores [8], which implies increase in the off-chip memory bandwidth. To mitigate this external bandwidth problems we need to increase the on-chip cache memory. However the exponential increase in on-chip cache sizes associate with on-chip wire delays[15] and access latencies. To incur high performance from CMPs we need to manage the limited on-chip cache resources typically LLC shared by multiple cores [3] [12].

1.2 Motivation

As we said earlier to get high performance from CMPs we need to manage LLC in better way. Among primitive organizations of LLC, shared organization gives good performance compared to private, because majority of LLC accesses are to shared cache blocks. So a shared LLC always dominates an LLC that is composed of private LLCs in performance. In most processors until very recently, a cache structure is designed to have uniform access latency for every cache block in the system. As cache becomes larger Uniform Cache Access (UCA) scheme gives less performance, hence a new scheme with Non Uniform Cache Access (NUCA) evaluated, which will give different access latency for different cache blocks based on the distance from the requester core.

Based on the mapping schemes NUCA architecture divided into Static and Dynamic. If we map cache blocks statically to one cache bank, we will call it as S-NUCA. In S-NUCA we don't need a searching mechanism, because there is no movement to cache blocks, hence there is no movement if a core want to access a cache block which is far away from requester core we will get more access latency for every access. Where as in D-NUCA cache blocks can be mapped to any of the cache banks, so that we can move these cache blocks from one cache bank to another. In D-NUCA if a core is accessing a cache block which is far away from requester core, we will move that cache block nearer to requester core upon every access by that core. In this way D-NUCA will reduce access latencies to far away banks, but because of migration movements we need a good searching algorithm for locating the cache blocks. Because of migration movements we can not predict the location of the cache blocks, recent works have proposed predicting data location [1] within the NUCA cache, unfortunately predicting data location within the NUCA cache has proved to be an extremely difficult task. As mentioned in the paper [3] upto now "efficient search for D-NUCA is an Open Problem". Hence if we don't have good searching mechanism for a data block in NUCA cache we strain ourselves in searching whole cache in case of a miss. So in order to get good performance from D-NUCA we need an efficient searching mechanism. We will explain in detail about all the architectures and mechanisms described here in the next two chapters.

- The main focus of our thesis is making efficient search mechanism by fulfilling required constraints.

In our algorithm we logically divided the cache into clusters and banksets, which contains cache banks. We assigned data blocks logically to the banks and each cache bank will maintain the information about the cache blocks which belongs to it. And also we are introducing state for each cache block based on the movement of the cache block. By using state bit we are reducing number of searches (multicast searches) to cache banks.

1.3 Organization of report

Rest of the report is organized as follows:

- In Chapter 2 we explain different cache architectures for CMPs, their advantages and disadvantages.
- In Chapter 3 we explain related work done, their advantages and disadvantages. We explain the existing approcahes to solve the problem.
- In Chapter 4 we discuss about our HKState-NUCA algorithm. We discuss how it performs better than existing approaches.
- In chapter 5 we discuss about our simulator, results and analysis. We compare the cache parameters and discuss about the performance of our algorithm.
- In chapter 6 we discuss about main improvements and future scope of the work done so that we can make it more efficient.

Chapter 2

Cache Architectures

2.1 CMP Cache types

CMP cache architectures are primitively divided into two types: Shared and Private [3]. Modern CMPs consists of multiple processors and multilevel cache hierarchy within a single chip. Typically each core has its own private L1 cache which intern divided into private L1 data and L1 instruction caches, and a shared L2 cache which is LLC.

Shared L2 Cache

A single large LLC shared by multiple cores on a chip as shown in Figure 2.1. When a core finds a miss in L1 the request goes to shared LLC , which will search for the block, there are no duplicates in shared LLC but the block may be present in multiple private caches. Coherence must be maintained among L1 and LLC. Hit latency of the shared LLC is high compared to private LLC due to its large size. And capacity misses are less because there are no duplicates only single copy of shared data exist in LLC. Because only one copy is maintained and available, space is dynamically allocated among the cores which lead to better space utilization and better cache hit rate. But the disadvantage with the shared cache is if the data is shared by multiple cores there will be more coherence misses.

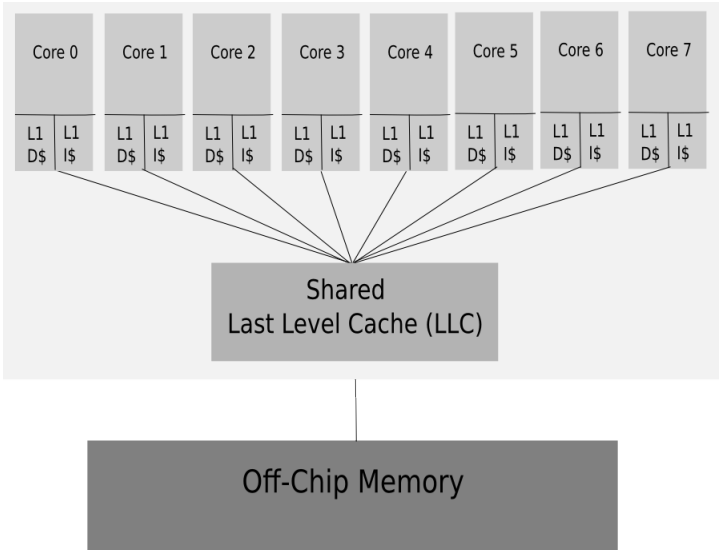


Fig. 2.1 Shared LLC Organization.

Private L2 Cache

Every core is associated with private L1 cache and a private L2 LLC (small relatively compared to shared one) as shown in Figure 2.2. When a core finds a miss in L1 cache it will search in its own private LLC. So here LLC hit latency is low because each core has its own LLC that to small size. But due to small sized LLC there can be high capacity misses which lead to expensive memory accesses. Due to replication of shared data in LLCs we need a complex coherence protocol for LLCs. The LLC capacity may not utilized efficiently because of the static allocation of the cache space among cores.

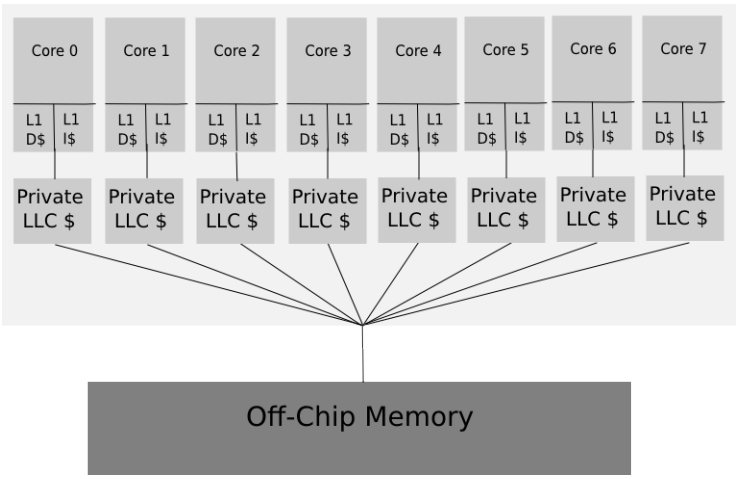


Fig. 2.2 private LLC Organization.

Hybrid L2 Cache

Based on these two types of CMP caches by taking their advantages and disadvantages into consideration , combination of both private and shared cache as shown in Figure 2.3 will give good results in future. Like dividing cores into groups and associating one private LLC to each group which is shared among the cores of that groups.

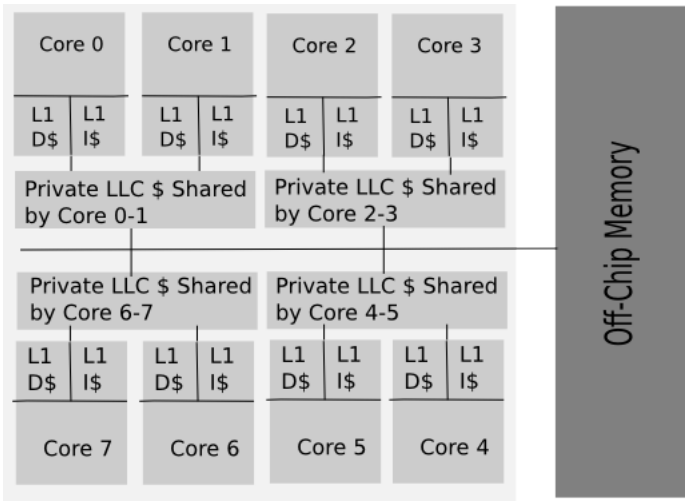


Fig. 2.3 Hybrid LLC Organization

2.2 Centralized vs Distributed Shared Cache

CMP with shared LLC is subdivided into two types: Centralized and Distributed [3].
In centralized shared cache LLC occupies contiguous area on the chip as shown in Figure 2.4. A large LLC is logically divided into banks and surrounded by multiple cores , the core which is nearer to a bank can access it fast than the farthest bank. By keeping the cache banks in close proximity data movement between banks is very fast there is no interconnect required here.

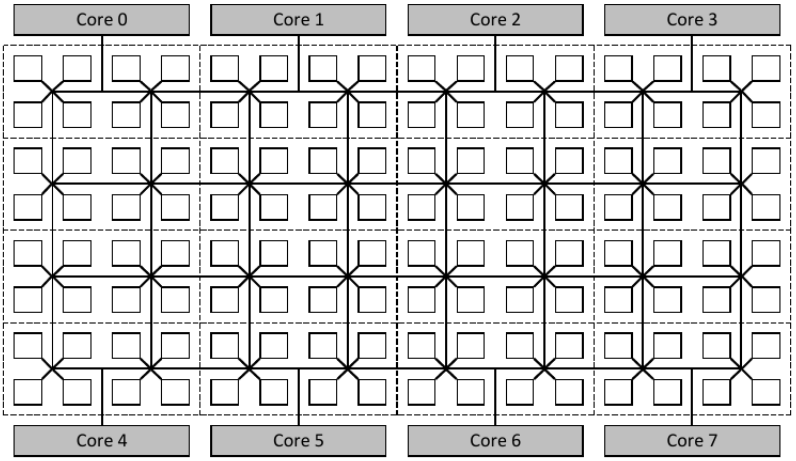


Fig. 2.4 A shared LLC with centralized layout.[13]

In distributed shared cache LLC is divided into banks and distributed among the cores, that is LLC is logically shared and physically distributed among the cores by using interconnects as shown in Figure 2.5. Each bank is local to a core so, each core has quick access to its local bank.The main disadvantage with the distributed system is higher cost in moving data between LLCs.

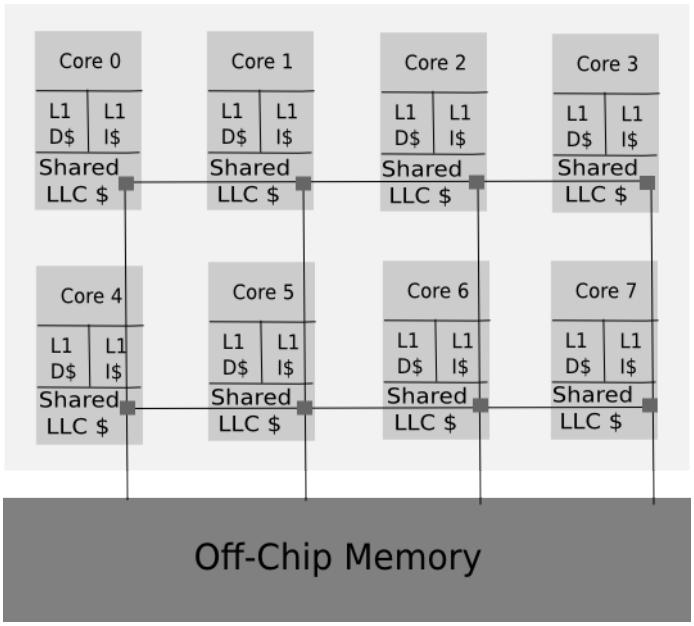


Fig. 2.5 A shared LLC with distributed layout.

2.3 Non-Uniform Cache Architectures (NUCA)

In traditional uniform cache access (UCA) design data nearer or farther to the processor are accessible with same latency. Due to large on-chip caches it is very difficult to manage uniform access latency, so data residing near processor in large cache is fast accessible than the far away data. Non-Uniform Cache Architecture (NUCA) [12] designs have been proposed to address this situation.

NUCA Organizations

NUCA cache divides the cache memory into smaller cache banks that are distributed along the chip and each bank can be accessed independently. In this architecture the access latency not only depends on the size of the cache but also depend on the distance to the requested processor. So NUCA cache gives lower access latency in accessing nearer blocks than the farther banks which reduces the effect of the cache wire delays.

There are two most typical NUCA organizations for CMPs those are heavy-banked NUCA cache, and the tiled-CMP architecture. The former is the centralized shared LLC, the later is the distributed shared LLC. NUCA design has been classified based on the placement policy as static (S-NUCA) and dynamic (D-NUCA).

In S-NUCA mapping of data to bank is statically determined by a set of bits (index) ,based on associativity. Here cache sets are distributed across the banks , one set is associated with one or more banks. Because the mapping of data blocks is static it doesn't support movement of data blocks. S-NUCA doesn't require searching mechanism for a cache block, because blocks are statically associated with cache banks by using index bits we can determine which bank the block belongs to. The main disadvantage of the S-NUCA is movement of the cache blocks are not possible due to which we can't move a block nearer to requesting core lead to high access latencies. By improving proximity of data with respect to cores we can mitigate access latencies due to distance from core, this can be done in D-NUCA.

2.4 Dynamic-NUCA

In this organization data can dynamically mapped to cache banks and frequently accessed data can dynamically migrated to closer banks of the requesting core. NUCA Cache is fully described by addressing these four issues [3] to manage the data.

1. `Mapping:` How the data are mapped to the banks and which banks can accommodate a data block ?
2. `Movement:`How the data are migrated from one bank to another and under what conditions the data should be migrated ?
3. `Replacement:`Which data block should be evicted in order to provide space for a data block (if the bank is full) and what is the final destination of the evicted data block.
4. `Searching:`In which cache banks we need to be searched to find a data block ?

1. Mapping

To provide dynamic migration constraint to D-NUCA here ways and sets are distributed across banks, All the ways of a set can be distributed across different banks, so that a data block of a particular set can be placed or migrated to any bank in the cache.

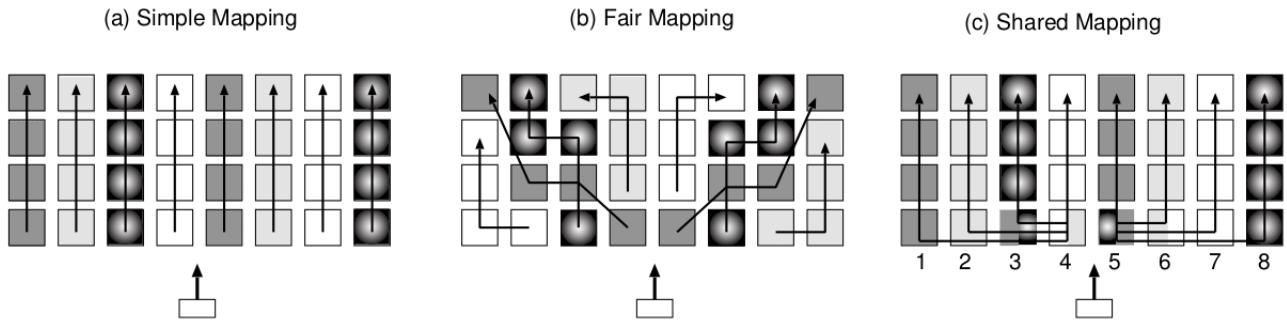


Fig. 2.6 Mapping bank sets to banks [12].

In simple mapping policy each column of banks in the cache becomes one bank set , each bank-set consists of all the ways of a set. Here the latencies to access bank-sets are different from each other. To mitigate the problem of non-uniform latencies to each bank in Fair mapping policy each banks are associated with bank-sets in such a way that the average access latency across all the bank sets are equilized. But the disadvantage is the complex routing path from one bank to another bank within a bank-set, which lead to longer latencies. In shared mapping policy closest banks are shared among multiple bank-sets. If a bank is shared by all the bank-sets then that bank can accommodate for any cache line.

2. Movement

Every time a hit occurs the cache block moves closer to the requested core, that is heavily used blocks are gradually migrated towards the closer banks.

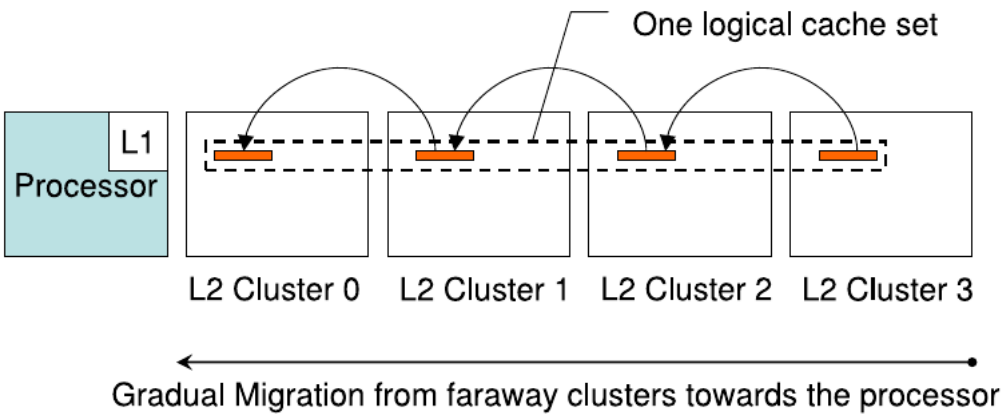


Fig. 2.7 Migration policy for single processor.[11]

A data is initially placed to the farthest bank and move closer gradually by swapping with LRU data. If there is no space for initial placement in farthest bank we need to evict a cache block (LRU). We can also place a data in nearest bank but if the space is not available evict a LRU block to nearer cache banks.The big challenge in the migration policy is finding the best location for a particular data block.

3. Replacement

There are two situations in which we need replacement policy, one is initial placement of the data, if the cache bank doesn't have free space for the data then it need to replace an existing cache block (victim), another situation arises because of movement of data. As discussed earlier D-NUCA adopts LRU policy for replacement. Here we need to take care of evicted data (victim). There are two approaches to victim migration, one policy is zero copy policy here the data is evicted to the off-chip memory, another policy is one copy policy here victim data not removed from the cache but placed in a lower priority bank, if the none of the bank is free then data is evicted to off-chip memory.

4. Searching

In D-NUCA data blocks mapped to multiple cache banks, and because of the migration policy adopted by D-NUCA, data blocks are constantly changing their location to optimize future accesses. So cache controller don't know where exactly the block is located at current time. Hence to search for a block we need to look whole cache (bank-set) in case of miss. The biggest challenge in searching is to find requested data block in less time, by resulting less traffic messages, and by consuming less power.

The main focus of this paper is finding efficient search mechanism by fulfilling above constraints. Some of the existing search mechanisms will be discussed in the next chapter.

Chapter 3

Related Work : Searching mechanisms for D-NUCA

In this chapter we will discuss various searching mechanisms proposed in these papers [12],[13],[17],[9],[11], to improve access latency, power consumptions and network traffic.

3.1 Primitive search mechanisms

The simplest searching techniques is the incremental search [12], in which cache banks are searched sequentially starting from the closest bank of the bank-set until the block is found or a miss occurs. Since it is sequential the hit latency is very high but, the energy consumption is very low.

The second method is multicast search [12], in which the requested block is simultaneously searched across all the cache banks of the bank-set by multicasting the address. Because of the multicasting it consumes high bandwidth intern consumes high energy but, the hit latency is very low.

There are some hybrid intermediate policies, limited multicast [12], in which initially some banks of the bank-set are searched simultaneously using multicast search if a miss occurs the remaining banks are searched sequentially using incremental search. Another hybrid policy is partitioned multicast, in which a bank-set is broken into subsets of banks, each subset is searched sequentially but the banks of a subset are searched in parallel.

There are some smart search mechanisms [12] [10] to improve the performance and energy consumption. A partial tag structure is stored in the cache controller. Here the tag bits of the requested block are compared in parallel with the stored partial tag bits to find a miss or hit, which will reduce the miss resolution time. Another variant of the smart search can be first the local bank is searched, if a miss occurs the partial tag bits are searched in parallel. This mechanism has high accuracy, but this can lead to extremely high storage overheads. The storage overhead for a 6 bit partial-tag is almost 10% of the cache space which is very high and searching for a partial tag in it again lead to same problems. A similar approach will be discussed in bloom filters for smart search paper.

In the paper [12] it is showed that these searching mechanisms increases accesses to single-processor D-NUCA cache, but if we considered CMP architecture the implimentation of these searching mechanisms are impracticle [4] they won't accelerate accesses.

3.2 HK-NUCA: Boosting Data Searches in D-NUCA for CMPs

The baseline architecture of this paper [13] is centralized shared CMP as shown in Figure 2.4. Author partitioned the shared LLC into small banks, and one bank holds one way of the bank-set. A bank-cluster consists of one way (one bank) of every bank-set, so that we can place a data block in any of the bank-cluster. In this paper author provides a fast and energy-efficient access to data by exploiting migration features and home knowledge. Here the term home bank is predetermined based on the lower bits of the data block's address which are denoted as home select bits. A data block can stay in any of the banks that form a bank-set, but the initial position of the block is determined based on the home select bits and the block will be placed in home bank.

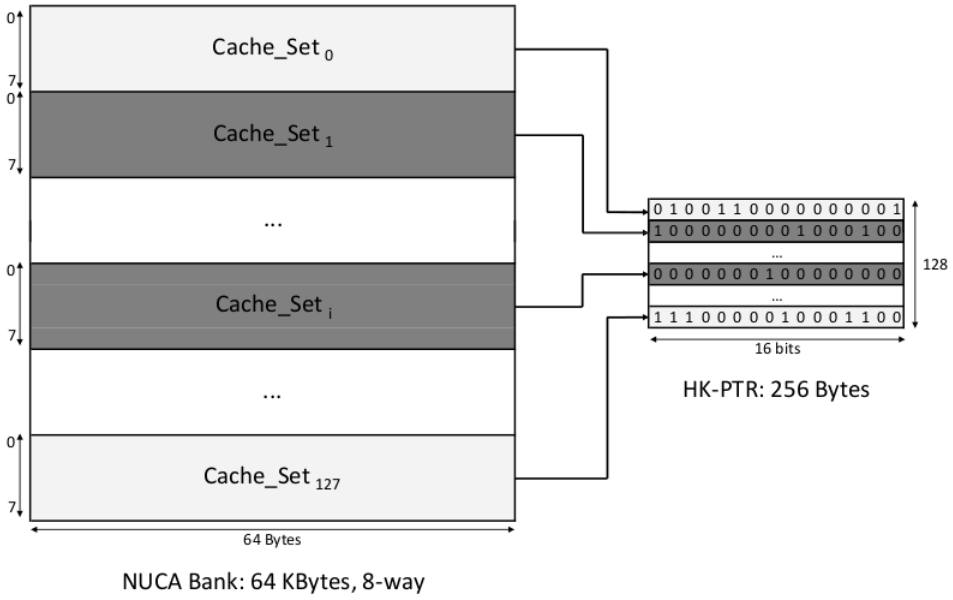


Fig. 3.1 HK-NUCA ptr list(HK-PTR) for a 64KB cache bank.[13]

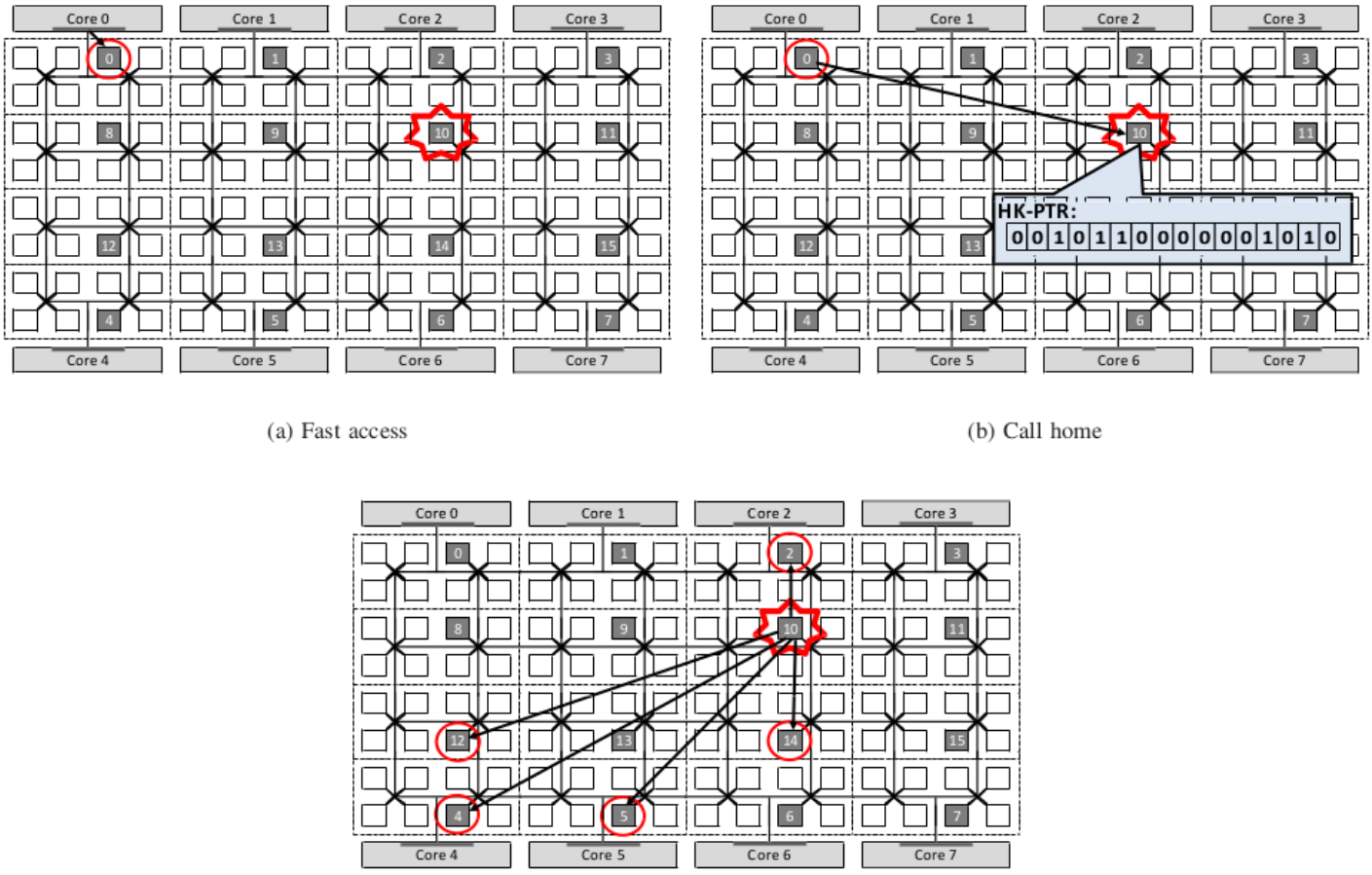
The basic function of home is to know which other NUCA banks have at least one of the data block that it manages. To configure this functionality to home, every bank has a set of HK-NUCA pointers (HK-PTRs) as shown in Figure 3.1. HK-PTR has the bits that are equal to number of ways of a bank-set, which means it will assign a single bit to each bank of the bank-set. If a bit in HK-PTR is 1 then the corresponding cache bank contains at least one block of the current home bank, if the bit is set to 0 then the corresponding cache bank doesn't contains a single block of the home bank.

Access Policy

In HK-NUCA bank access policy consists of three stages.

1. **Fast access:** Because of the migration policy adopted by D-NUCA frequently accessed data moved closer to the requested core, so most of the times the block can be found in Local bank-cluster (nearer bank-cluster to a core) as shown in Figure 3.2(a). Hence first access the corresponding cache bank in the local bank-cluster. If there is a hit then the

block is accessed with a minimum access latency, otherwise the request is forwarded to home bank by entering into next stage.



(a) Fast access

(b) Call home

(c) Parallel access

Fig. 3.2 Stages of HK-NUCA searching mechanism.[13]

2. **Call home:** In this stage the request is forwarded to the home bank by using home select bits. In home bank data is searched if it is found the data is send back to the requested core as shown in Figure 3.2(b), otherwise home bank will get the corresponding HK-PTR then find out which cache banks can have the requested data, then request is forwarded to the banks which are set in HK-PTR in parallel by entering into next stage.

3. **Parallel access:** This is the final stage request is forwarded in parallel to all the banks that are set in HK-PTR as shown in Figure 3.2(c), here only few banks (which are set) are accessed because of the home knowledge which will reduce the network traffic and power consumption. If the block is not found in any of the requested banks the request is finally sent to the off-chip memory.

Managing Home Knowledge

Keeping updated HK-PTRs is very critical for ensuring accuracy. There are three actions that may arise update in HK-PTR:

- When a new data enters into the cache.
- An eviction from a NUCA bank.
- A migration movement.

The main advantage of the HK-NUCA is reducing miss resolution time in case all the bits in HK-PTR set to 0, this can be done in only two cache bank accesses (local bank and home bank). Even if the HK-PTR bits set to 1, we are sending request to fewer cache banks compared to previous search mechanisms.

3.3 Leveraging Bloom Filters for Smart Search Within D-NUCA Caches

This paper [17] presents a complexity-effective solution to the smart search problem. It takes the advantage of the bloom filters in searching cache banks. Compared to partial tag it reduces the storage requirements, power consumptions and network traffic. Basically bloom filter is a structure uses a simple indexing mechanism to update it. Whenever an item inserted into a bloom filter based on some hashing functions it will update it's structure so that , if an item which is already inserted searched again it will surely get hit. In bloom filter there are no false negatives, but there is a small chance of false positives.

The baseline architecture of this paper is centralized shared CMP, like in HK-NUCA cache is divided into banks which form bank-sets and bank-clusters. The bank-clusters which is nearer to a core is called Local bank-cluster, the group of bank-clusters which are not nearer to any of the core are called as Center bank-clusters and remaining clusters which are local to other cores are called as Inter bank-clusters. Here each core maintains a bloom filter for each bank-cluster, So to search a block we will search it's tag in corresponding bloom filter instead of searching it directly in bank-clusters.

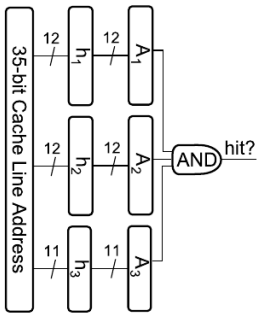


Fig. 3.3 Searching in Bloom filter, showing hash functions h1 through h3 , which are used as indexes into bit array slices A1 through A3. A hit is detected when ones are returned from all array slices.[17]

The bloom filters will direct L1 misses to search in banks which likely to have the required data block. The access mechanism is as follows, whenever a L1 miss happens first it will check in the corresponding bloom filter of the local bank-cluster, if there is a hit it will search the block in local bank-cluster as shown in Figure 3.3. There can be miss in this situation because bloom filters are effected with the false positives. Then in the second stage it will check corresponding bloom filters of the center bank-clusters , search starts in hit areas. If miss happens in center bank-clusters, in the final stage inter bank-clusters are searched based on corresponding hit in bloom filters. Here the main advantage is bloom filters are directing to candidate cache bank , So no need to search the whole cache in case of miss and compared to partial tag array bloom filters reduce space utilization by an order of magnitude.

The main disadvantage of bloom filters is as time goes on filters give more false positives which lead to search whole cache after some insertions. This is because as more items are inserted false positives increases and information becomes stale because of migration and eviction. To eliminate these issues we need to clear an entire filter when it's false positives rate becomes high by putting a counter or we can go for counting bloom filters , but that will again lead to space over head.

3.4 C-AMTE:Constrained Associative Mapping of Tracking Entries

The baseline architecture used in this paper [9] is Tiled CMP architectures which is nothing but distributed shared NUCA cache as shown in Figure 2.5. It uses static home tile concept similar to home bank in HK-NUCA, but here a data block is mapped to a cache bank which is local cache to a core (both cache and core together form a tile) is called static home tile (SHT) of that block. SHT of a block is determined using home select bits (HS) like in HK-NUCA.

According to SHT a block might be mapped to a tile which is far away from the requester core, lead to more access latencies in the further accesses. To mitigate this problem D-NUCA uses migration or replication mechanisms, so after these operations the block is at a tile different than it's home tile which is denoted as host tile. Now we can't able to find the block using HS bits anymore.

To locate a block at current host tile author come up with a robust solution, by maintaining tracking entries at each tile. Tracking entry tracks the copies of a cache block corresponding to that tile. Author maintains two tracking entry tables at each tile those are :

1. `Principal tracking entry (PTR):` Stores pointer to its own data block of home tile, when a requester core asks for the block it will provide current host of that block.
2. `Replicated tracking entries (RTR):` Stores pointer to a data block at requester tile , which also pointing to current host tile to directly locate it.

Assume a CMP organization with PTR and RTR tables, C-AMTE Algorithm for locating a block is as follows as in the Figure 3.4

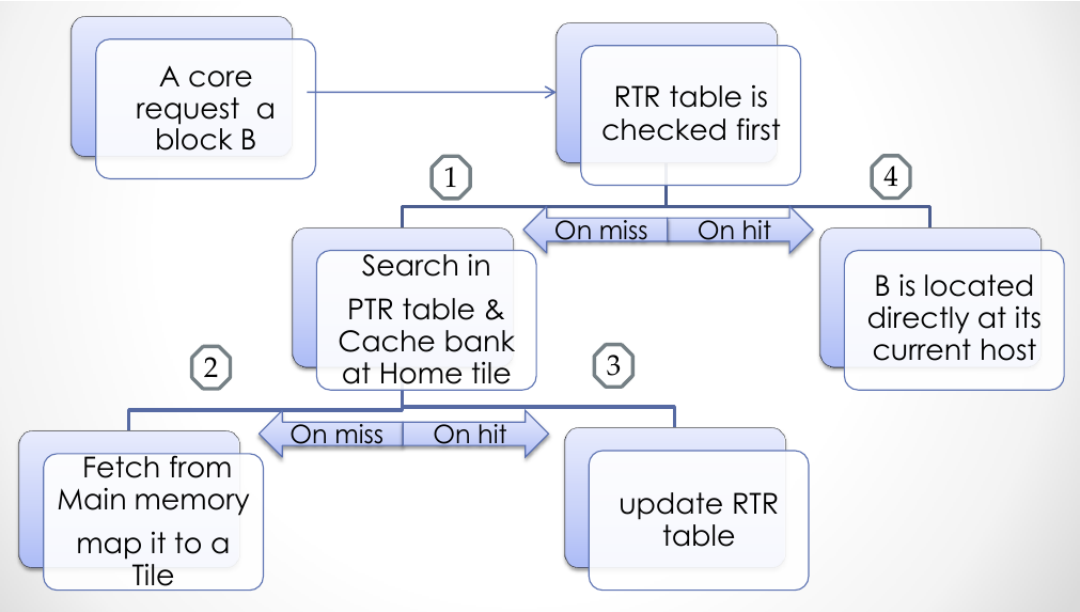


Fig. 3.4 C-AMTE Algorithm for locating a block b

Maintenance and coherence of the tracking entries

The principal and replicated tracking entries need to be kept coherent. We accomplish this by embedding a bit vector with each principal tracking entry at the PTR tables to indicate which cores had cached related replicated tracking entries at their RTR tables. Each time a data block is migrated to a different tile, the principal and replicated tracking entries of the corresponding block updated, which point to the new host tile of that block.

There can be false misses, if the block is in transit, for this reason we need to maintain a copy of the data block kept at current host tile of that block. By using acknowledgement message by new host tile , the current host tile can delete that block.

C-AMTE mechanism is non-scalable to a large number of tiles, because of the bit vector associated with each principal tracking entry.

3.5 A Novel Migration Based NUCA Design

This paper [11] evaluates a novel migration scheme to address the L2 NUCA data placement problem.

Sharing Pattern for LLC Lines

Author simulated applications from various benchmarks to understand usage characteristics of a LLC. If a cache line is accessed by only one processor in its lifetime we call it as private cache line, otherwise it is shared cache line. Based on the simulation results author made some conclusions those are:

1. On average, 87.2% of the L2 lines are private.
2. Shared lines are used most frequently than the private lines. Almost 65.1% of the L2 accesses are to the shared cache lines.
3. First requester is the most frequent requester, 50% of the time.

Based on the above results we can made some conclusions. However the private cache lines dominate shared lines, most of the line accesses to shared ones. So shared lines plays important role in performance metrics. If we can find a best suitable position for a shared line we can reduce number of migrations. And for newly fetched lines (initial placement) place it in local cache of the requester because, most of the cache lines are private and first requester is the frequent one, so that we can reduce migrations for private cache lines.

Chapter 4

HKState-NUCA: Home knows where to find data and its state within the NUCA Cache

D-NUCA employs dynamic data movement, which is useful in data proximity to requester core. In D-NUCA data proximity will decrease the distance and improve access latency, but due to movement of data blocks, we cant keep track of them. If we dont have any mechanism for searching a block, we strain ourselves in searching whole cache in case of a miss. Hence searching data blocks will increase access latencies drastically, In order to get significan't performance benefits from D-NUCA we need to improve searching mechanism to find data block in cache.

As we said earlier none of the searching techniques including recent innovations discussed in this paper are efficient because of the poor data proximity, space overheads, power limitations and wire-delays on-chip. In all the above search mechanisms to confirm a miss in NUCA cache we must access all banks of a bank-set to ensure that the requested data block is not present. If we can assign data blocks logically to the banks, then the banks can maintain the information about the blocks which belongs to it. HK-NUCA paper is based on the same methodology, but it is not efficient because of unreduced access latencies and dynamic energy consumption, due to initial placement and large number of searches (hops) respectively. By exploiting private and shared cache lines we can reduce dynamic energy consumption.

In this section we will describe the proposed searching algorithm.

4.1 Baseline Architecture

We assumed similar architecture used in HK-NUCA searching algorithm [13] which is centralized shared CMP. In HKState-NUCA address space spreads across the cache banks, which are connected via a 2D mesh interconnection network and surrounded by the cores. Here shared LLC is partitioned into small banks, and one bank holds one way of the bank-set. A bank-cluster consists of one way (one bank) of every bank-set, so that we can place a data block in any of the bank-cluster.

Here the term home bank is predetermined based on the lower bits of the data block's

address which are denoted as home select bits. A data block can stay in any of the banks that forms a bank-set, but the initial position of the block is determined based on the requested core local cluster and the block will be placed in local bank.

The basic function of home is to know which other NUCA banks have at least one of the data blocks and their states that it manages. To configure this functionality to home, every bank has a set of HK-State pointers (HKState-PTRs). HKState-PTR has the bits that are equal to double the number of banks of a bank-set, which means it will assign two bits to each bank of the bank-set. In HKState-PTR shown in Figure 4.1 first bit represents the state of the cache block that belongs to home bank set, second bit represents the presence of the cache block.

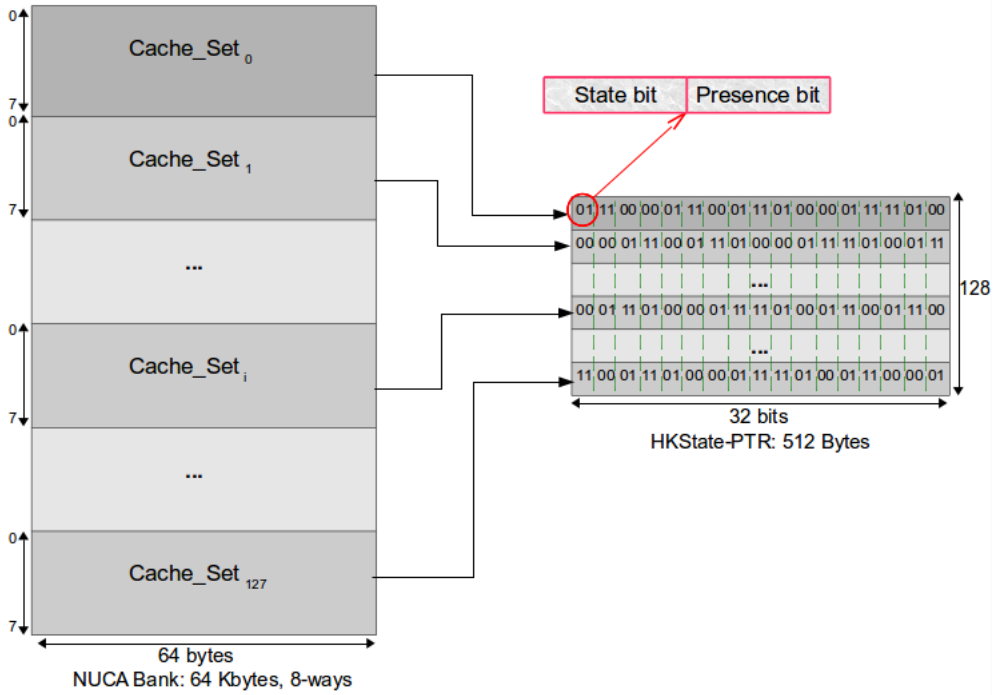


Fig. 4.1 HKState-NUCA Pointer list (HKState-PTR) for a 64KB cache bank.

Here we are introducing states for each cache block, state represents the movement of the cache block. Initial state of the cache block is none, which represents upto now there is no promotion or demotion happens. And the remaining two states represents moved category (may be shared cache blocks), those are promoted and demoted. If the cache block is moved from local cache bank to central cache bank then its state becomes demoted, if it is moved from central cache bank to local cache bank then its state becomes promoted.

Here we are assuming each bank can accommodate n ways corresponding to its home bank, so if we have only one bit in presence section we have to check complete set for every updation of HKState-PTR. In our implementation we made the presence bits and state bits based on the associativity of the cache bank, even for the HK-NUCA also we considered same to compare perfectly with our algorithm. Because in HK-NUCA author has not discussed how to manage more than one cache block with single bit in HK-PTR while updation. We are adding extra bits (n bits) to presence section and state section of HKState-PTR for maintaining how many

ways corresponding to home bank are present and their state, so that we can manage updations easily. Which will increase minor space complexity, But it will increase overall performance. If we are not considering extra bits for presence section, we need to check complete set for every updation, which will increase time complexity. If we consider one bit (way) in presence section we are naming it as HKSptr-1-way structure, otherwise we are naming it as HKSptr-n-way structure.

If pair of bits in HKState-PTR is 00 then the corresponding cache bank doesn't contains a single block of the home bank (assumption : presence section having one bit is same as having more than one bit). If it is 01 then the corresponding cache bank contains atleast one block of the current home bank and theirs state is none. If it is 11 then the corresponding cache bank contains atleast one block of the current home bank, and theirs state is moved (promoted or demoted).

4.2 Access Policy

In HKState-NUCA bank access policy consists of four stages.

1. **Fast Access :** Because of the initial placement and migration policy adopted by D-NUCA frequently accessed data moved closer to the requested core, so most of the times the block can be found in Local bank-cluster (nearer bank-cluster to a core). Hence first access the corresponding cache bank in the local bank-cluster. If there is a hit then the block is accessed with a minimum access latency, otherwise the request will be forwarded to home bank. In the example shown in the Figure 4.2 core 2 requested for an address which contains required data, first accesses its local cache bank. If there is a miss at local cache bank request is forwarded to home bank by entering into next stage.

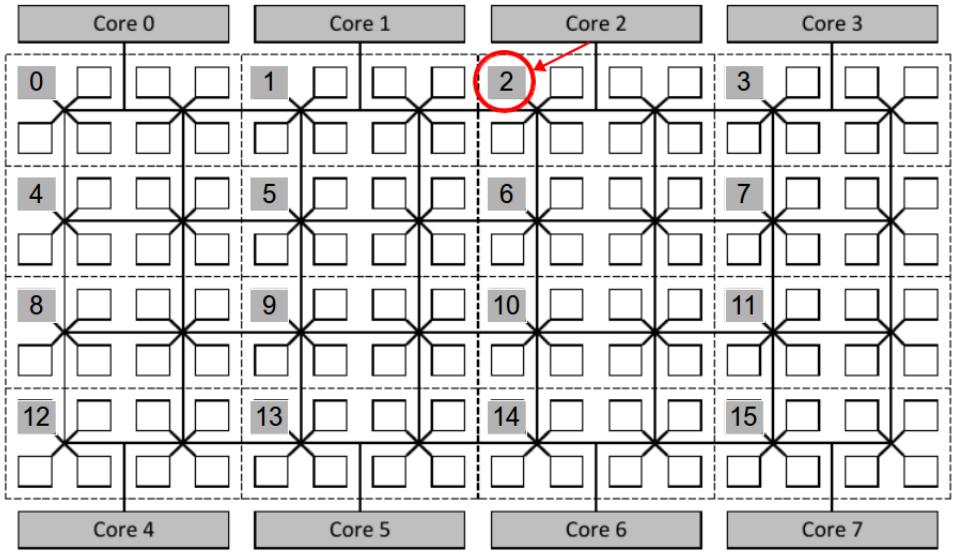


Fig. 4.2 HKState-NUCA Fast Access stage.

2. **Call Home :** In this stage the request is forwarded to the home bank by using home select bits. In home bank data is searched if it is found the data is send back to the requested core, otherwise from home bank core will get the corresponding HKState-PTR to find out which cache banks can have the requested data. Then request will be forwarded to the banks which are set in HKState-PTR in parallel by entering into next stage. In the example shown in the Figure 4.3 core 2 will find hs bits of the requested address and forward the request to home cache bank present in cluster 7, it will access the home bank. If a miss happens at home bank core will access HKState-PTR as shown in Figure 4.5 corresponding to the required cache block and enters into next stage.

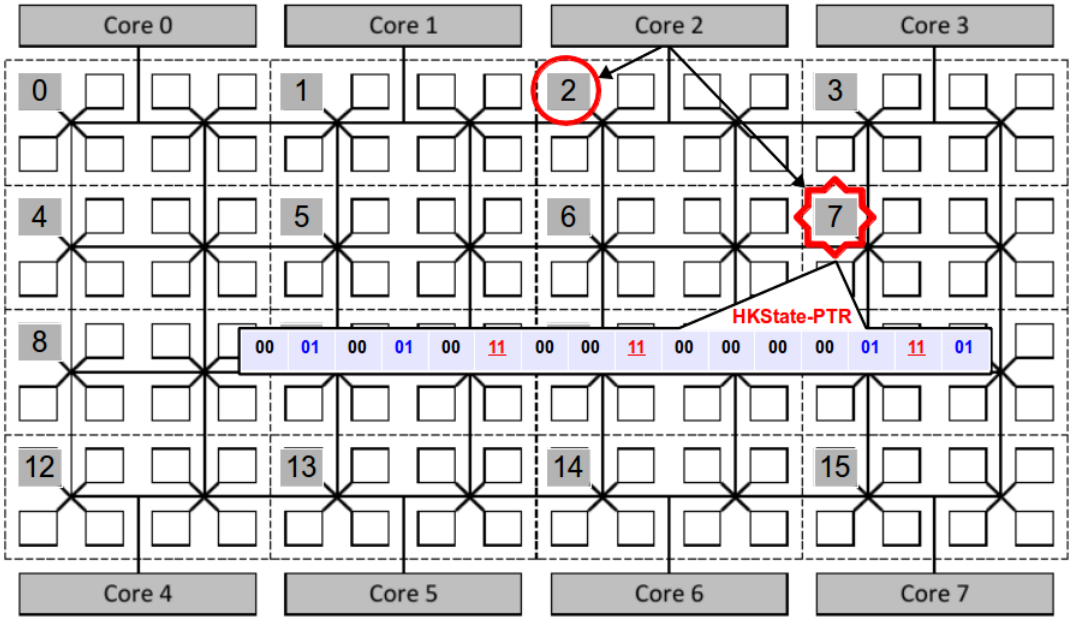


Fig. 4.3 HKState-NUCA Call Home stage.

3. **Parallel Access to moved state :** In this stage request is forwarded in parallel to all the banks that are set in presence section as well as moved state bit in HKState-PTR, here only few banks (which contains promoted and demoted cache blocks) are accessed because of the home knowledge and state knowledge, which will reduce the network traffic and power consumption drastically. In the example shown in the Figure 4.4 core 2 found cache banks of cluster 5, cluster 8 and cluster 14 contains at least one cache block with moved state. Core will forward the request in parallel to those clusters, if a miss happens core will send request to remaining presence clusters by entering into next stage

4. **Parallel Access to none state :** This is the final stage request is forwarded in parallel to remaining banks that are set in presence bit in HKState-PTR. If the block is not found in any of the requested banks the request is finally sent to the off-chip memory. In the example shown in the Figure 4.6 core 2 found cache banks of cluster 1, cluster 4, cluster 11 and cluster 13 contains at least one cache block with none state. Core will forward the request in parallel to those clusters, if a miss happens core will send request to the off-chip memory.

   As we got less hitrate in stage 2 we can combine stage 1 and stage 2 by parallelly accessing local cache bank and home bank, so that we can reduce number of stages and access time.

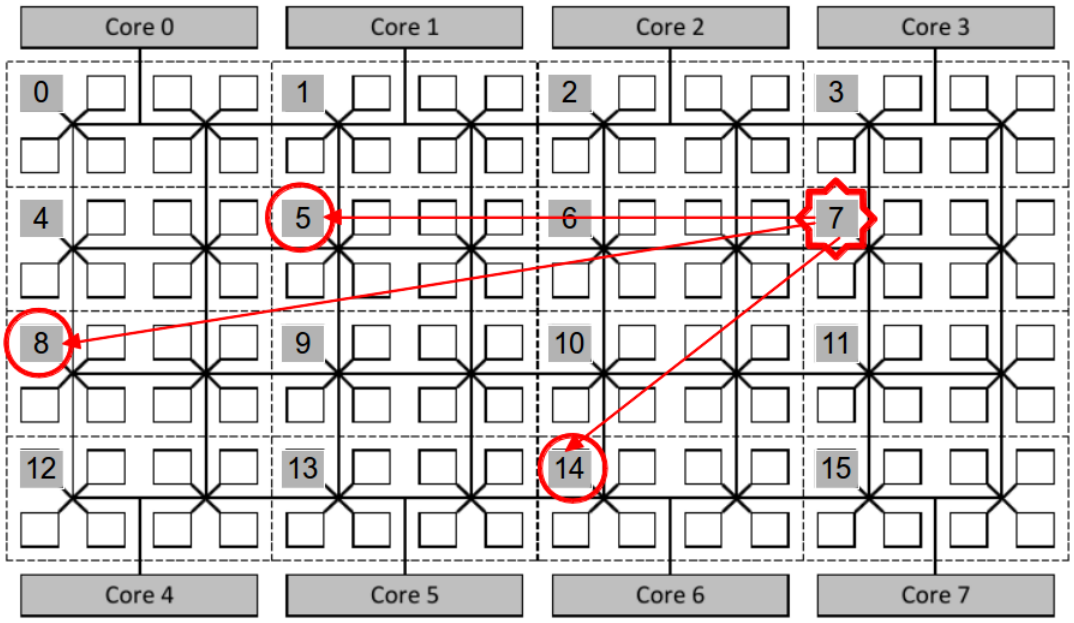


Fig. 4.4 HKState-NUCA Parallel Access to moved states stage.

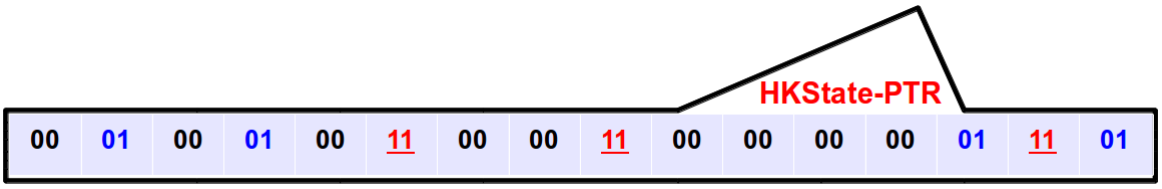


Fig. 4.5 HKState-PTR for the example.

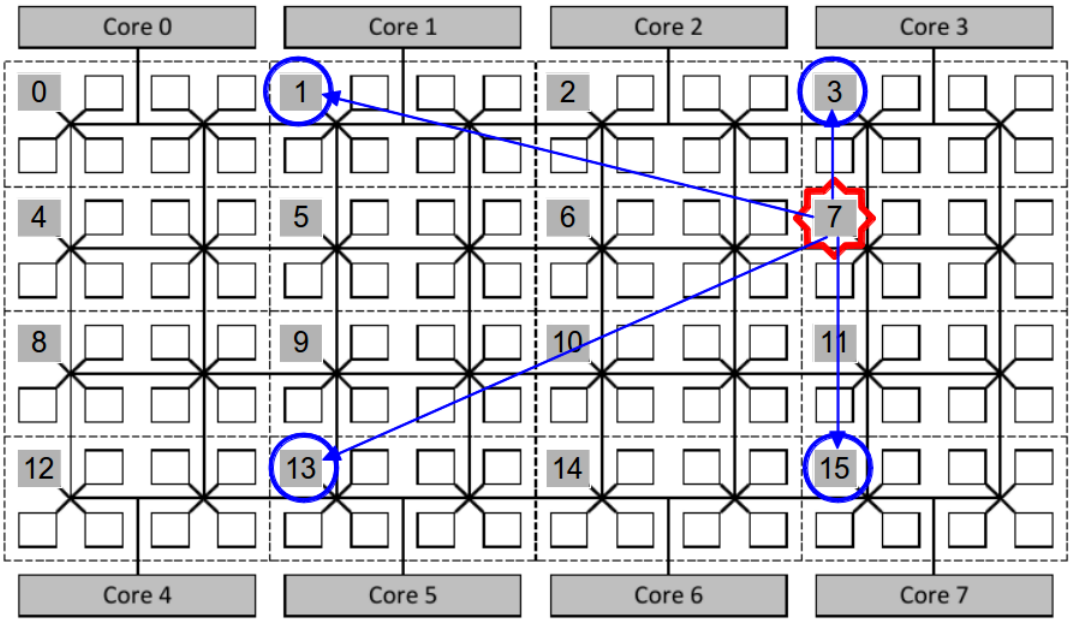


Fig. 4.6 HKState-NUCA Parallel Access to none states stage.

4.3 Managing Home State Knowledge

Keeping updated HKState-PTRs is very critical for ensuring accuracy. There are three actions that may arise update in HKState-PTR. here we will explain updation for HKSptr-1-way structure. For HKSptr-n-way structure algorithm 1 will explain update for all the three actions.

1. **An insertion into a NUCA bank :** Similar to HK-NUCA for initial placement of cache block we will update corresponding HKState-PTR presence bit to 1, if presence bit is 0 initially, otherwise we will keep the presence bit intact. In both the cases we will keep the state bit intact, Because if state bit is set means other cache blocks corresponding to home in that cache bank are in moved state so no need to update state bit to 1 again, if state bit is 0 means other cache blocks corresponding to home are present then they are in none state so no need to update state bit to 0 again.

2. **An eviction from a NUCA bank :** While evicting a cache block from the cache, if that cache bank does not contain atleast one cache block corresponding to home we will update corresponding HKState-PTR to 00. Otherwise, If the cache block state is moved and other cache blocks corresponding to home are in none state then we will update corresponding HKState-PTR to 01. If any of the other cache block corresponding to home is in moved state then we don't update HKState-PTR, we will just evict the cache block.

3. **A migration movement :** While migration we will swap the cache blocks from one cache bank to another cache bank if we do not have space for migrating cache block. Here we will swap the HKState-PTR information corresponding to swapping cache blocks with respect to source and destination cache banks. If we move cache block from source cache bank to destination cache bank, we will update HKState-PTR at destination cache bank like in insertion but its state is updated instead of none and we will update HKState-PTR at source cache bank like in Eviction.

Algorithm 1 Updation of HKState-PTR for HKSptr-n-way structure

Begin:
if *Update for insertion* **then**
  increment Pbits of HKSptr
else if *Update for eviction* **then**
  if *evicting cache block state is none* **then**
    decrement Pbits of HKSptr
  else
    decrement Pbits and Sbits of HKSptr
  end if
else if *Update for migrating movement* **then**
  if *moving cache block state is none* **then**
    decrement Pbits of HKSptr corresponding to SCbank
    increment Pbits and Sbits of HKSptr corresponding to DCbank
  else
    decrement Pbits and Sbits of HKSptr corresponding to SCbank
    increment Pbits and Sbits of HKSptr corresponding to DCbank
  end if
  if *Core finds no space in destination cache bank for moving cache block* **then**
    we need to update HKSptr corresponding to victim cache block
    increment Pbits and Sbits of HKSptr corresponding to SCbank
    decrement Pbits and Sbits of HKSptr corresponding to DCbank
  end if
end if
End:
Shortcuts: *HKState-PTR = HKSptr*, *presence bits = Pbits*, *state bits = Sbits*, *source cache bank = SCbank*, *destination cache bank = DCbank*

Chapter 5

Experimental Evaluation

5.1 Simulation Setup

In this section we will explain how we simulated our algorithm and existing searching algorithms. We created a virtual standalone simulator called CmpSim in C, which is similar to some prior simulators like SimpleSim [2], Multi2Sim [19], GEMS [14], CACTI [16] etc. But CmpSim doesn't simulate data and instructions, it will take addresses specific to each core from different benchmarks and produces hits, misses, replacements, hops count and block movements specific to cache banks and cores.

CmpSim initially creates a cache grid of clusters, each cluster contains a cache bank of all the banksets. In each cache bank contains its properties and sets, each set contains its ways, each way contains a cache block and its properties. Here we have created Grid structures Figure 5.1 contains 2D Clusters, which contains pointers to Cluster structure. Cluster structure contains 2D Cache banks, which contains pointers to Cache bank structure.

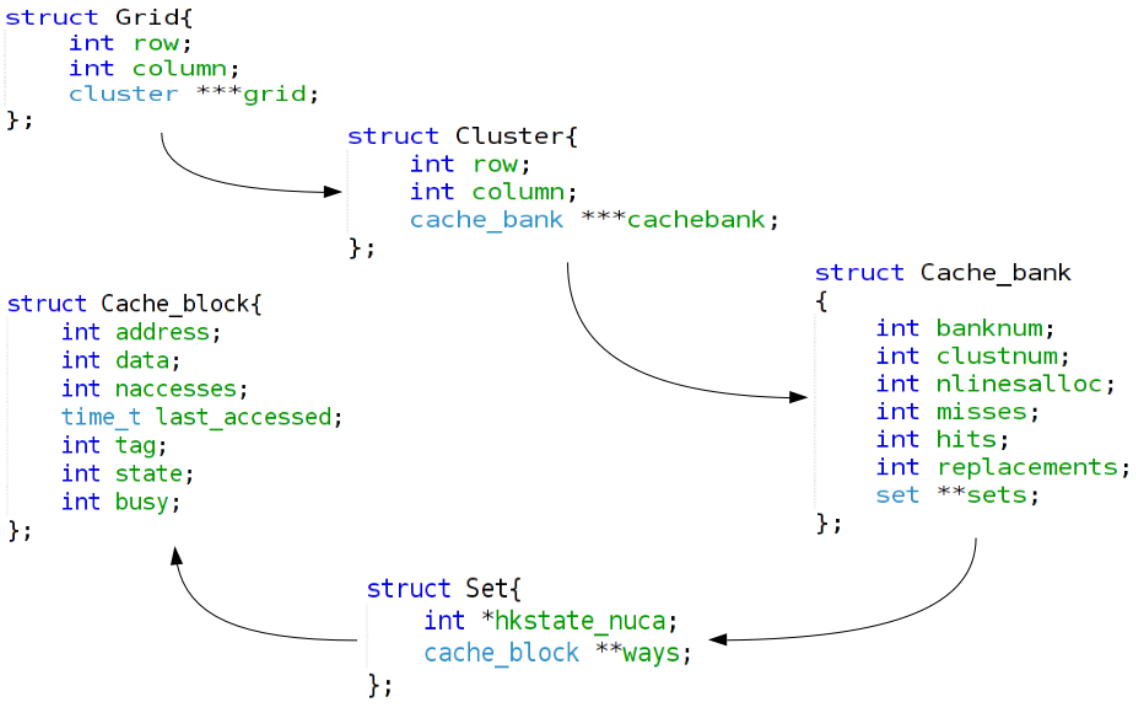


Fig. 5.1 Cache internal structure

Cache bank structure contains 1D Sets, which contains pointers to Set structure. Set structure contains 1D Ways, which contains pointers to Cache block structure. Here in every level each structure has its own properties, thats the reason to create structures internally.In our configuration we will compute bank set associativity based on the number of cores, so each grid having n clusters, where n is double the number of cores (bankset associativity) and each cluster having 8 cache banks.

In CmpSim we have created number of threads that are equal to number of cores by using pthread library, here each thread will act as a core. We have generated the trace file for 6 Parsec benchmarks using GEMS simulator [14], the trace file contains core number and access to an addresses. We have separated these addresses according to core number and we put them in a file with core number, so that each core can access individually its own set of addresses from its own file. In our experiment we have considered 16 cores, so we will run 16 threads initially. Each thread will take an address from its own file and we will search for that address in the cache. If the address is absent in the cache we will insert it into the cache, if it requires any migrations or replacements we will do it. Here for every address at every thread (core) we will run Linear Search, HK-NUCA search and HKState-NUCA for synchronous results. In all these algorithms we are using different cache grids for each algorithm, So that we can get our results for three algorithms in a single run.

Here we don't have cache coherence protocols, so instead of that we are maintaining a hash table and we will insert every address with respect to each core. If a core is accessing a address, after taking the address from the core specific file we will insert it into hash table if it is already existing, another core is accessing the same address so we will move to busy waiting state (cpu polling). After this insertion no other core can access the address we are accessing, after Implementing searching algorithms we will delete the address from the hash table and take another address from to access, In this way we are dealing with coherence issue.

Because of threads' parallelism we got some memory problems for shared variables (critical section), we are using mutex locks to deal with critical section problems. Cache block is also a critical section to deal with cache blocks we are using a busy state for every cache block, when ever any core is accessing a cache block we will make the busy bit 1, so that no other core can access that block for any other purpose like replacement, migration etc.

5.2 Experimental Results

This section discuss results and analysis of performance comparision with respect to different parameters. The configuration parameters used in my work are shown in the Table 5.1. In our experiment we have considered 16 cores, But we explained with 8 core examples in all the chapters for simplification.

Number of Cores in the system	16
Cache size	8 MB
Cache line size	64 Bytes
Bank set associativity	32
Bank associativity	8

Table 5.1 Configuration Parameters

5.2.1 Overall Analysis

As we said earlier we simulated three searching algorithms, So we compared results of those three algorithms for different benchmarks. We already know the fact that HK-NUCA searching algorithm [13] is better compared to many of existing algorithms like linear search. So we mainly focused comparing our algorithm with HK-NUCA, if our algorithm is better than HK-NUCA which will imply our algorithm is better than remaining algorithms also. We have chosen PARSEC benchmark suite [5] [6] for our experimentation and successfully executed 6 full benchmarks and got results. Table 5.2 shows the complete results for PARSEC Benchmarks.

PARSEC Benchmarks	HK-NUCA Results			
	Hits	Misses	Replacements	Hops
Body	3367691	200736	399536	104476396
Ferret	13913820	1453506	2905089	467672995
Fluid	5005012	1943873	3885778	206191280
Freqmine	6074884	259239	516551	155030837
Vips	4065369	3161654	6321374	219428596
x64	12570149	1938590	3875199	438830885
	HKState-NUCA Results			
	Hits	Misses	Replacements	Hops
Body	3376743	191685	381666	52912370
Ferret	13991583	1375742	2749829	186902768
Fluid	5090518	1858367	3715002	133327349
Freqmine	6099328	234794	467991	22161036
Vips	4196391	3030633	6059596	144666678
x64	12685945	1822793	3643882	189566717

Table 5.2 HK-NUCA and HKState-NUCA results for PARSEC Benchmarks

First we will compare number of invocations and hit rate in each stage for HK-NUCA and HKState-NUCA. In the stage 1 of access policies for HK-NUCA and HKState-NUCA we got huge change in hit rate, for HK-NUCA we got 6.57% and for HKState-NUCA we got 33.99% on an average. In stage 2 home bank access also we got improvement in hit rate, with home bank search HK-NUCA got very less hit rate (0.07%) but we got good hit rate (0.67) compared to HK-NUCA for our algorithm By using state bit in HKState-NUCA, we are reducing many unnecessary accesses to none state banks in stage 3. The Table 5.3 shows the number of times each stage is invoked and got hit.

Number of times each Stage in access policy invoked and got hit

	HK-NUCA access policy stages				% times stages invoked and got hit				
	Stage 1	Stage 2	Stage 3	Misses	stage1	stage2	stage3	stage4	misses
Benchmarks									
Body	97528	6127	3207421	197361	2.78%	0.17%	91.42%		5.63%
Ferret	2113100	2874	11634251	1436415	13.91%	0.02%	76.61%		9.46%
Fluid	718948	4049	4188182	1907429	10.54%	0.06%	61.42%		27.97%
Freqmine	42902	2334	6025475	259060	0.68%	0.04%	95.19%		4.09%
Vips	184241	4088	3878031	3162423	2.55%	0.06%	53.65%		43.75%
x64	1297270	8002	11264877	1938590	8.94%	0.06%	77.64%		13.36%
Average					6.57%	0.07%	75.99%		17.38%
	HKState-NUCA access policy stages								
	Stage 1	**Stage 2**	**Stage 3**	**Stage 4**					
Body	982283	7494	1834726	495472	188462	28.00%	0.21%	52.29%	14.12%	5.37%
Ferret	2933465	50702	10158289	684618	1359566	19.32%	0.33%	66.89%	4.51%	8.95%
Fluid	706526	87957	3637272	563327	1823526	10.36%	1.29%	53.34%	8.26%	26.74%
Freqmine	5472711	19575	571258	31595	234632	86.46%	0.31%	9.02%	0.50%	3.71%
Vips	2042462	111106	1582604	461241	3031370	28.25%	1.54%	21.89%	6.38%	41.93%
x64	3704024	52612	8270062	659248	1822793	25.53%	0.36%	57.00%	4.54%	12.56%
Average						32.99%	0.67%	43.41%	6.39%	16.55%

Table 5.3 Number invocations for each stage.

The Table 5.2 Comparision between hit rate for each stage for HK-NUCA and HKState-NUCA.

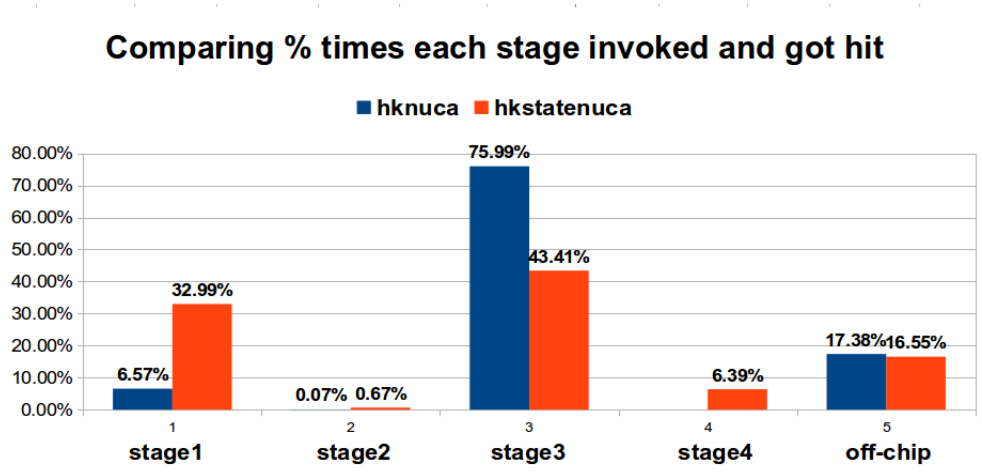


Fig. 5.2 Hitrate Comparision for each stage.

We can see that number of invocations and got hits for stage 2 are very less compared to other stages, which is even less than 1%. So we can combine stage 2 with stage 1 by searching local cache bank and home cache bank parallelly, which will improve access time.

- **Hit Ratio Comparision:** If we compare hit ratio of HKState-NUCA with HK-NUCA, we got 7.04% improvement on geo-mean of all the benchmarks we executed.
  Figure 5.3 shows the hit ratio comparision for different benchmarks.

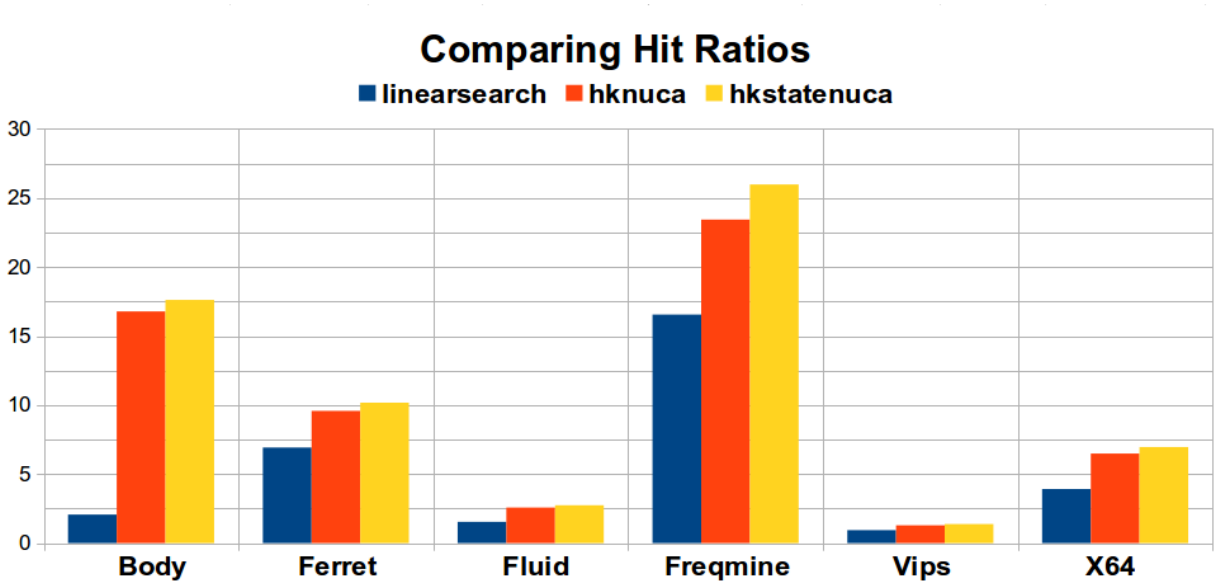


Fig. 5.3 Hit Ratio Comparision

Figure 5.4 shows hit ratio improvement of our algorithm over HK-NUCA.

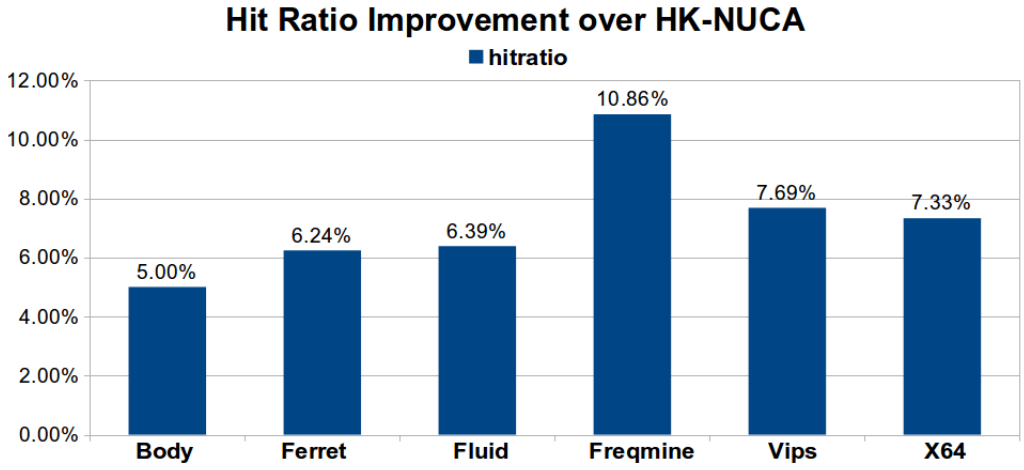


Fig. 5.4 Hit Ratio improvement over HK-NUCA [13]

- **Hop Count Comparision:** If we compare number of searches (hop count) of HKState-NUCA with HK-NUCA, we got 50.89% reduction in searches on geo-mean of all the benchmarks we executed.

  Figure 5.5 shows the hop count comparision for different benchmarks.

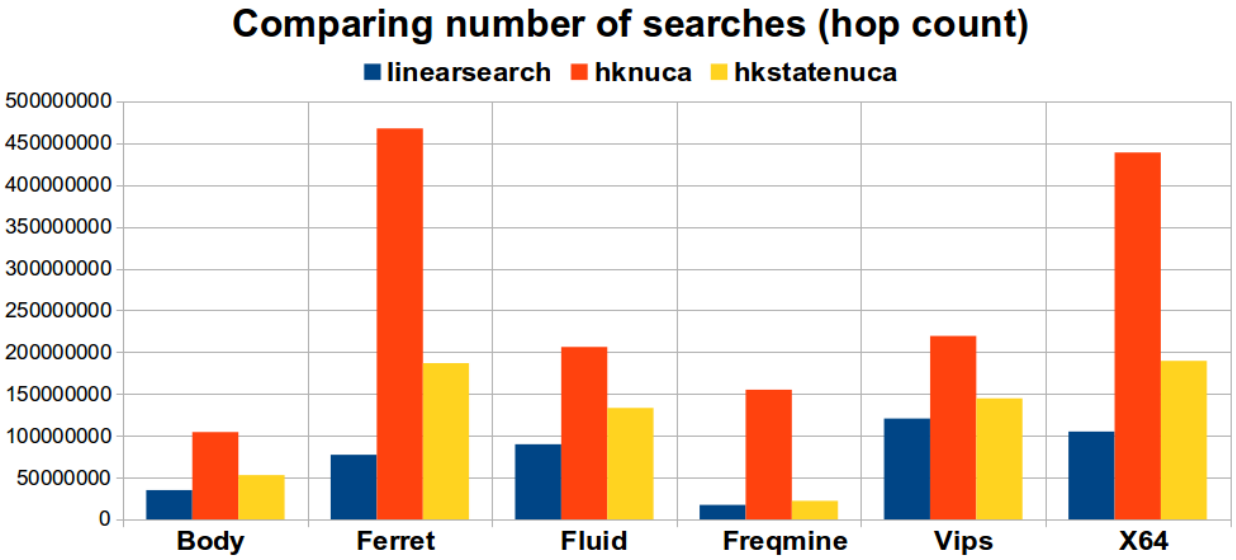


Fig. 5.5 Hop Count Comparision

If we compare HKState-NUCA with linear search we are getting almost 70% increase in number of searches, because every bank will be searched sequentially for linear search and when ever core finds a cache block in a cache bank it won't search remaining banks. Where as HK-NUCA and HKState-NUCA algorithms uses multicast search which will search all the cache banks specified by the algorithm parallelly, even if it finds a cache block in a cache bank.

Figure 5.6 shows reduction in number of searches of our algorithm over HK-NUCA.

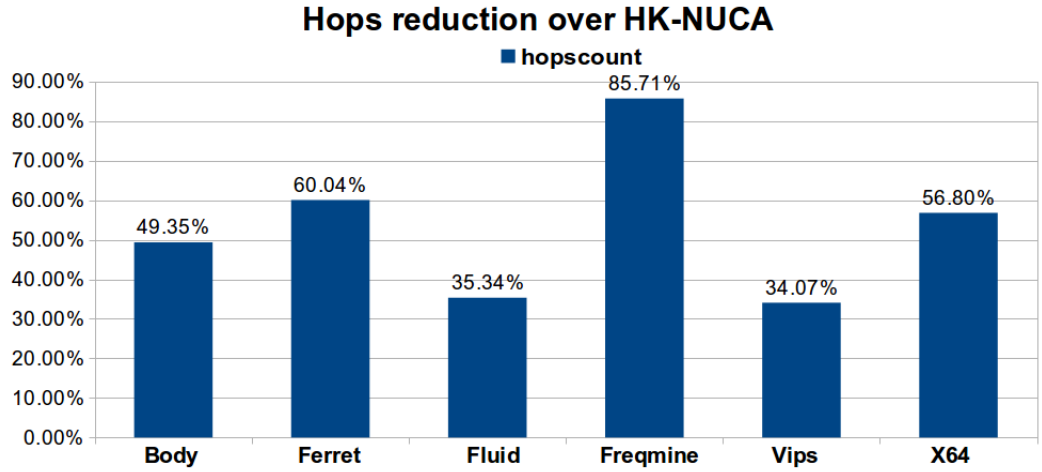


Fig. 5.6 Hop Count reduction over HK-NUCA [13]

But in Linear search even the hop count is less access time and hit rate will be very much more compared to HK-NUCA and HKState-NUCA, so overall performance will be very less for linear search. For HK-NUCA and HKState-NUCA we need more hardware support compared to Linear search.

- **Replacements Comparision:** If we compare number of replacements of HKState-NUCA with HK-NUCA, we got 5.39% reduction in replacements on geo-mean of all the benchmarks we executed.
  Figure 5.7 shows the replacements comparision for different benchmarks.

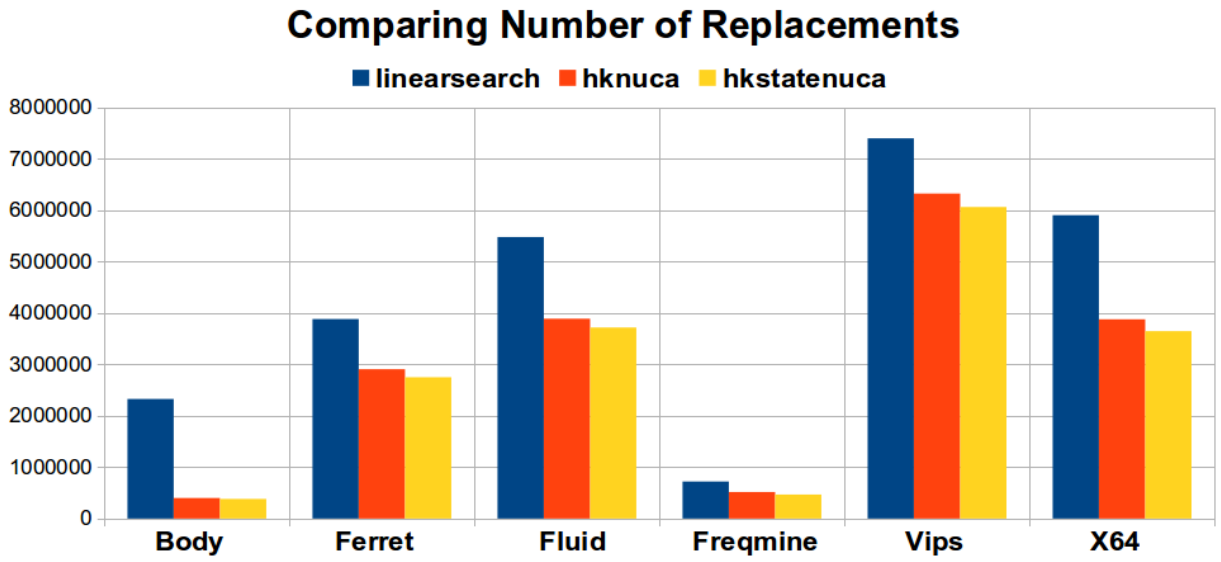


Fig. 5.7 Replacements Comparision

Figure 5.8 shows reduction in number of replacements of our algorithm over HK-NUCA.

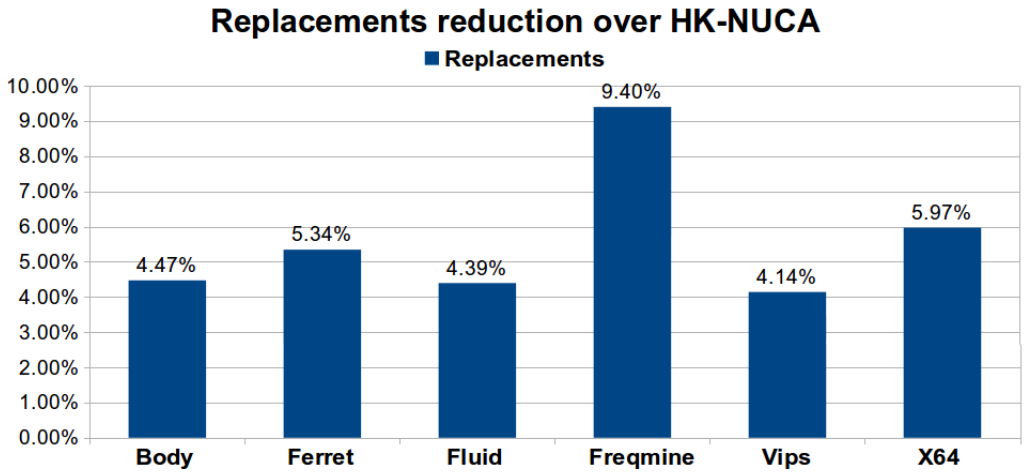


Fig. 5.8 Replacements reduction over HK-NUCA [13]

5.3 Hardware Overhead

In our algorithm state section in the HKState-PTR structure plays key role in reducing number of searches (hops). As we said earlier by introducing state section for the HKState-PTR structure we are doubling the size compared to HK-PTR used in HK-NUCA algorithm, so it made double the overhead compared to HK-NUCA, which is less than 0.8% of hardware or space overhead. Apart from this we are using 1 bit for representing state of each cache block which is negligible compared to cache block size (64 Kbytes). Apart from hardware overhead due to HKState-PTRs, our algorithm introduce some time complexity if we use HKSptr-1-way structure for checking home cache blocks in the set while updating HKState-PTR. Which are same in HK-NUCA algorithm also, to overcome these time complexities we are introducing HKSptr-n-way structure, which can able to store state and presence for more than one cache block. But if we use HKSptr-n-way structure based on the n value, it will increase space overheads by multiple of log(n).

Although our HKState-NUCA algorithm made some small hardware overheads, it will give better performance than many of the existing algorithms even HK-NUCA in all aspects.

Chapter 6

Conclusion

6.1 Improvements of HKState-NUCA over HK-NUCA

1. Initial placement of the cache block nearer to requested core. Based on the conclusion of the paper [A Novel Migration Based NUCA Design] [11] first requester is the most frequent requester, that means more than 50 % of the time initial requester is accessing the cache block. So local bank (cache bank nearer to requested core) is the best initial placement for the cache block. Because of the better initial placement, HKState-NUCA algorithm got 33% hitrate in the first stage only, where as HK-NUCA got 6.57% hitrate in the first stage in our experiment.Which proves that the local cache bank is the best initial placement for cache blocks, it should be implemented in every D-NUCA searching algorithm to get good performance.

2. Reducing number of searches(hops) in case the cache block is found in third stage, where in third stage we are eliminating none state banks. If we get a miss in first and second stages, most of the times the cache block would be found in third state. In our algorithm, if a cache block state is none means that is accessed by only one core (at initial placement). If a cache block state is either promoted or demoted then after initial placement the cache block is moved, may be because it is requested by another core or because of migration movement policy adopted by D-NUCA. That means none state does not contain shared cache blocks, But few cache blocks of moved state may be private. So if the cache block is not found in local bank then in most of the cases that block may become shared or migrated, in very few cases(in the first access only) that cache block is private if it is present in the cache.
   Based on the conclusions of the paper [A Novel Migration Based NUCA Design] [11]

   - On average 87.2% of the cache lines are private.
   - Shared lines are used most frequently than the private lines.

   most of the time cache block found in third stage if it is not found in first and second stages. Because in third stage we are searching moved category which contains shared cache blocks. In very few cases(in the first access only) the cache block may found in fourth stage (none state). In every access on an average of 87.3% of the time we got hit in the stage 3 of HKState-NUCA access policy compared to stage 4 in our experiment. Which proves that the reduction in the number of hops due to exempting none state cache banks.

3. HKState-NUCA outperforms HK-NUCA in every aspect, our algorithm improves performance by 7.04% and reduces hop count which is dynamic energy consumed by the memory system by 50.89%. But because of HKState-PTR it increases minor space overhead (0.8%) and network traffic.

6.2 Possible Future Prospect

In our algorithm we are using moved state to represent movement, but we are not considering extra bits for separating promoted and demoted states. We can separate promoted and demoted states from moved state and find out which states contains more shared lines. So that we can add one more stage in between 3 and 4, or we can combine stage 4 with promoted or demoted based on the criteria that state containing less shared lines, which may decreases dynamic energy consumed by the memory system. But this type of assumptions will be specific to applications.

References

[1] S. Akioka, F. Li, K. Malkowski, P. Raghavan, M. T. Kandemir, and M. J. Irwin, "Ring data location prediction scheme for non-uniform cache architectures," in *ICCD*, 2008, pp. 693–698. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2008.4751936

[2] T. M. Austin, "Simplescalar tool set," 2002. [Online]. Available: http://www.simplescalar.com

[3] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-core cache hierarchies," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–153, 2011. [Online]. Available: http://dx.doi.org/10.2200/S00365ED1V01Y201105CAC017

[4] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *MICRO*, 2004, pp. 319–330. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MICRO.2004.21

[5] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009. [Online]. Available: http://www.bibsonomy.org/bibtex/2fe3313e649ce40ff06b50e3960069fa5/ytyoun

[6] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Princeton, NJ, USA, January 2011. [Online]. Available: ftp://ftp.cs.princeton.edu/reports/2010/890.pdf

[7] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[8] P. Dubey, "A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera," in *in Intel White Paper, Intel Corporation*, 2005. [Online]. Available: http://heim.ifi.uio.no/~inf3410/docs/Intel_Corp_Platform_2015.pdf

[9] M. Hammoud, S. Cho, and R. Melhem, "C-amte: A location mechanism for flexible cache management in chip multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, pp. 889–896, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2010.11.009

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088154

[11] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son, "A novel migration-based nuca design for chip multiprocessors," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/SC.2008.5216918

[12] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Acm Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 211–222. [Online]. Available: http://dx.doi.org/10.1145/605397.605420

[13] J. Lira, C. Molina, and A. González, "Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 419–430. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.48

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005. [Online]. Available: http://dx.doi.org/10.1145/1105734.1105747

[15] D. Matzke, "Will physical scalability sabotage performance gains?" *Computer*, vol. 30, no. 9, pp. 37–39, Sep. 1997. [Online]. Available: http://dx.doi.org/10.1109/2.612245

[16] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to model large caches," 2009.

[17] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian, "Leveraging bloom filters for smart search within nuca caches," in *7th Workshop on Complexity-Effective Design (WCED)*, 2006. [Online]. Available: http://www-new.cs.utah.edu/~rajeev/pubs/wced06.pdf

[18] J. L. Rueda, "Managing dynamic non-uniform cache architectures," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2011. [Online]. Available: http://arco.e.ac.upc.edu/wiki/images/3/32/Lira_phd.pdf

[19] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[20] Users-Anonymous, "Multi-core processor wikipedia." [Online]. Available: http://en.wikipedia.org/wiki/Multi-core_processor