Searching Mechanisms for Dynamic NUCA in Chip Multiprocessors

Kartheek Vanapalli

Indian Institute of Technology - Guwahati, India.

Dr. Hemangee K Kapoor

Indian Institute of Technology - Guwahati, India.

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 not keep track of data blocks. In D-NUCA mapping and migration schemes, increasing data proximity which reduces access latencies, but if we do not 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 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 [1] by introducing 0.8% space overhead.

Keywords: Searching Mechanisms, Dynamic NUCA, Chip Multiprocessors

1. Introduction

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) [2]. CMPs consists of multiple processors integrated on a chip.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.

For any multiprocessor 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 [3]. Future applications like big data processing will require more number of cores, 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 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 [4] [5].

1.1. 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 do not 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 la-

tency 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 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 [4] upto now "efficient search for D-NUCA is an Open Problem". Hence if we do not 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.

In our algorithm we have considered baseline architecture of HK-NUCA algorithm where it will divide the cache into clusters and banksets, which contains cache banks. And data blocks logically associated to the cache banks and each cache bank will maintain the information about the cache blocks which belongs to it. In this paper we are introducing state for each cache block based on the movement of the cache block. By using state bit we are reducing multicast searches (dynamic energy consumption) to cache banks and by changing initial placement we are improving hit rate.

As we said earlier none of the searching techniques including recent innovations 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 multicast searches respectively. By exploiting private and shared cache lines we can reduce dynamic energy consumption. The main focus of this paper is making efficient search mechanism by fulfilling required constraints.

2. Baseline Architecture

We assumed same architecture used in HK-NUCA searching algorithm [1] with some additions.

2.1. HK-NUCA Baseline Architecture

HK-NUCA uses centralized shared CMP, where 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 set. A bank-set contains all the banks that are holding same way, and these bank-sets are organized into bank-clusters. 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. To configure this functionality to home, every bank has a set of HK-NUCA pointers (HK-PTRs).

2.2. Additions to HK-NUCA for our algorithm

In our algorithm 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 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.

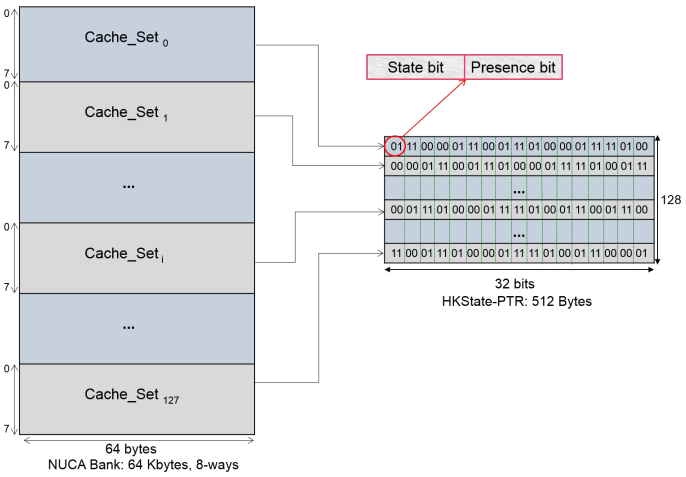

Figure 1: HKState-NUCA Pointer list (HKState-PTR) for a 64KB cache bank.

In this paper 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 movement happens. Second state is moved state, if the cache block is moved from local cache bank to central cache bank (demoted) or vice versa (promoted) its state becomes moved.

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.

As shown in Figure 3 if pair of bits in HKState-PTR is 00 then the corresponding cache bank does not contains a single block of the home bank. 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).

3. Access Policy

In this section we will describe the proposed searching algorithm. In HKState-NUCA access policy consists of four stages.

Stage 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 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.

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 2 core 2 will find hs bits of the requested address and forward the request to home cache

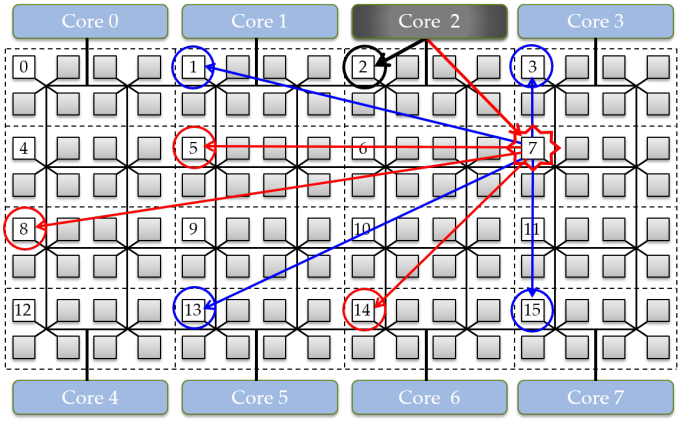

Figure 2: HKState-NUCA Access stages.

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 3 corresponding to the required cache block and enters into next stage.

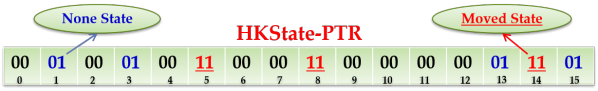

Figure 3: HKState-PTR for the example.

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 2 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

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 2 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.

4. 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 do not 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
```

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 simulator called CmpSim in C, which is similar to some prior simulators like SimpleSim [7], Multi2Sim [8] etc. But CmpSim does not 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. Each cache bank contains its properties and sets, each set contains its ways, each way contains a cache block and its properties.

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 [9], the trace file contains core number and access to an addresses. Based on the core number in the trace file we have given these addresses specific to core. Here for every address at every core (thread) 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 do not have cache coherence protocols, so to deal with coherence issue we are maintaining a hash table which will keep track of present accessing addresses. If an address is present in hast table we will not allow another core to access the same address, we will put another core in waiting state until it is deleted. 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.

5.2. Experimental Results and Analysis

This section discuss results and analysis of performance comparision with respect to different parameters. The configuration parameters used in my work are 16 cores, 8 MB cache, 64 KB cache lines and 8 way set associativity. But we explained with 8 core examples in all the sections for simplification.

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 [1] is better compared to many of existing algorithms like linear search. So we mainly focused comparing our algorithm with HK-NUCA,

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 1: HK-NUCA and HKState-NUCA results for PARSEC Benchmarks

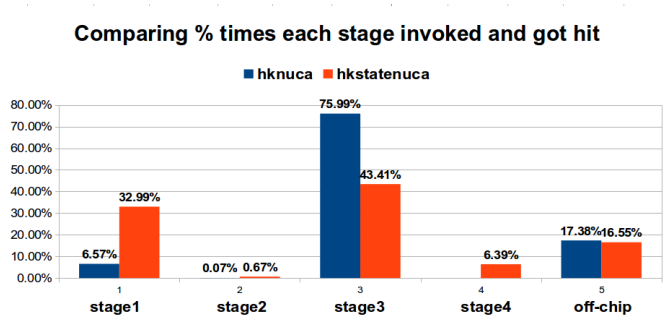

Figure 4: Hitrate Comparision for each stage.

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 [10] for our experimentation and successfully executed 6 full benchmarks and got results. Table 1 shows the 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.

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 and Figure 6 shows the hit ratio comparision and improvement for different benchmarks.

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 7 and Figure 8 shows the hop count comparision and reduction for different benchmarks.

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 would not 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. 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.

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, which is less than 0.8% of space overhead. And we are using 1 bit for representing state of each cache block which is negligible . Apart from space overhead our algorithm introduce some time complexity if we use HKSptr-1-way structure, to overcome these time complexities we are introducing HKSptr-n-way structure. But if we use HKSptr-n-way structure, it will increase space overheads by multiple of log(n). Although our algorithm made some small hardware overheads, it will give better performance than many of the existing algorithms even HK-NUCA in all aspects.

5.4. How Our Algorithm is getting better performance than HK-NUCA

1. Initial placement of the cache block in local cache bank : 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. Which proves local cache bank (cache bank nearer to requested core) is the best initial placement for the cache block.

2. Reducing number of searches(hops) by exploiting private and shared cache lines : In our algorithm, if a cache block state is none means that is accessed by only one core (at initial placement) else its state is moved means 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

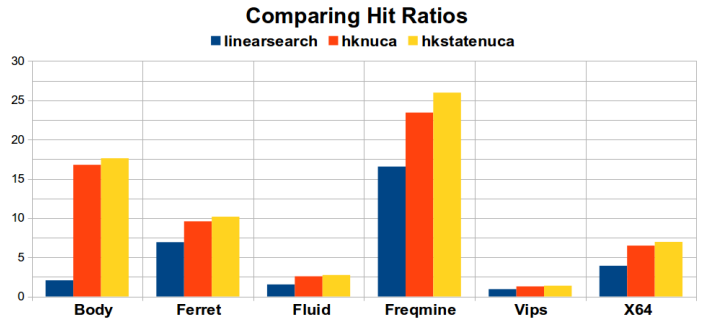

Figure 5: Hit Ratio Comparision

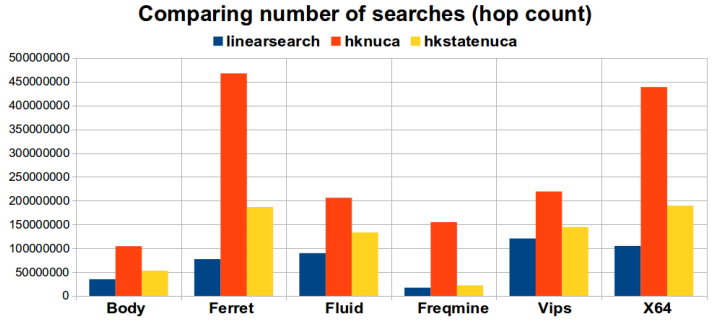

Figure 7: Hop Count Comparision

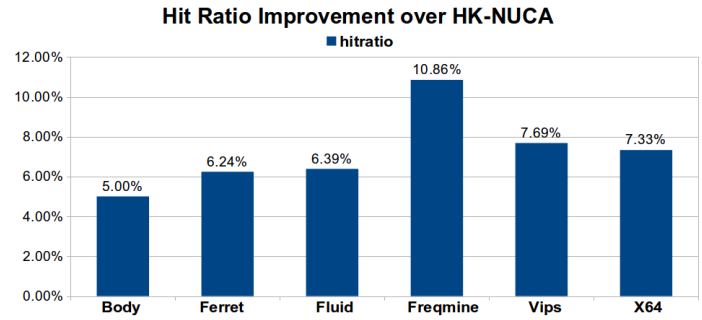

Figure 6: Hit Ratio improvement over HK-NUCA [1]

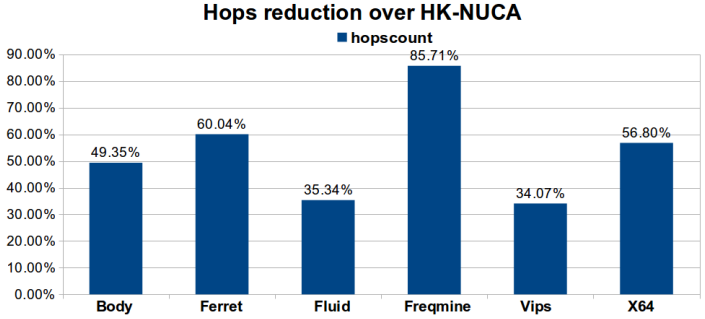

Figure 8: Hop Count reduction over HK-NUCA [1]

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 that cache block is private because it is ejected and requested by another core. Based on the conclusions of the paper [A Novel Migration Based NUCA Design [11]] *1) On average 87.2% of the cache lines are private. 2) 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 state blocks which contains shared cache blocks. In very few cases the cache block 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.

6. Conclusion

Our HKState-NUCA algorithm outperforms HK-NUCA in every aspect, our algorithm improves overall performance by 7.04% and reduces dynamic energy consumed by the memory system by 50.89%. But because of HKState-PTR it increases minor space overhead by 0.8%.

6.1. 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] J. Lira, C. Molina, 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.

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

[3] J. L. Rueda, Managing dynamic non-uniform cache architectures, Ph.D. thesis, Universitat Politècnica de Catalunya (2011).

[4] R. Balasubramonian, N. P. Jouppi, N. Muralimanohar, Multi-core cache hierarchies, Synthesis Lectures on Computer Architecture 6 (3) (2011) 1–153.

[5] C. Kim, D. Burger, S. W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, in: Acm Sigplan Notices, Vol. 37, ACM, 2002, pp. 211–222.

[6] T. M. Austin, Simplescalar tool set (2002).

[7] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, Multi2sim: A simulation framework for cpu-gpu computing, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, ACM, New York, NY, USA, 2012, pp. 335–344.

[8] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (gems) toolset, ACM SIGARCH Computer Architecture News

33 (4) (2005) 92–99.

[9] C. Bienia, 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.

[10] M. Kandemir, F. Li, M. J. Irwin, 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.