The CSI Journal on
Computer Science and Engineering
Vol. 17, No. 1, 2019
Pages 1-8
Regular Paper

Adaptive Cache Clustering in Multi-Core Architecture

Masoud Dehyadegari

K.N. Toosi University of Technology

Abstract

Last level cache (LLC) management can considerably improve the performance of Chip Multi Processors (CMPs) by reducing the miss rate and decreasing the average L1 miss latency. LLC management approaches try to either map data to caches which are closer to cores, or exploit caches of neighboring cores to increase performance and reduce off-chip misses. To reduce the miss rate, we need to use cache slices of the other cores in NUCA architectures, but it increases the average L1 miss latency since the data to be accessed by cores are located in remote cache slices. As a result, designing an intelligent management approach to improve the system performance on a variety of workloads and systems is necessary. In this paper, an adaptive cache clustering called ACC is presented. ACC augments L2 cache size by using the L2 cache slices of neighboring cores adaptively. ACC selects cache slices with minimum distance to the target core while evaluating the amount of L2 cache slices being used on the surrounding tiles. To validate our approach, we apply several applications from SPEC CPU2006 and PARSEC benchmark suites. The results show the effectiveness of our approach, as we get speedup by up to 18%.
Keywords: Chip Multiprocessor, Caches, parallel architectures.

1. Introduction

With continuing technology scaling, the number of cores in a chip is increasing constantly and will be up to thousands in the near future [1][2][3]. Recently, many-core architectures include tens to hundreds of cores, such as Intel's 80-core Terascale chip [4], NVIDIA 128-core Quadro GPU [2], and Tilera with 64-core [3]. Moreover, large scales Chip Multiprocessors (CMPs) generate more memory requests because of increasing concurrent executing applications or threads. Therefore, management of the last level cache can reduce off-chip misses and lead to obtain higher performance. Shared and private LLC are two approaches for designing LLC. However, shared LLC is more common because it increases cache space utilization. Moreover, it provides inter-core communications. Also, there is no need to transfer data among caches and to keep consistent data. Shared LLC may lead to inter-core misses in a way a core could evict the LLC content of another core. In this case, off-chip misses are increased. Although private caches should not be interfered by a running thread on another core, shared data need to be replicated in each private cache. Besides cores in large scale CMPs, shared last level on-chip caches are interconnected via Network on chip (NoC). This leads to variable access latency which is referred to as non-uniform cache architecture (NUCA) [5]. NUCA provides a scalable solution for managing the design complexity and effectively using the resources available in digital systems. In other words, LLCs are physically distributed while they can be assumed as logically shared or private. Although NUCA provides scalability, it leads to an increase in miss latency and needs a higher bandwidth network. Designers would like to achieve a low number of misses and less average LLC access latency. Shared caches lead to a smaller number of misses and higher LLC average latency while private caches cause higher LLC misses.

Several techniques are proposed in the literature to find a compromise between shared and private caches. Some NUCA architecture with private partitioning tries to dynamically allocate a part of the private cache to the required cores [6]. To reduce LLC access latency, several data management techniques are proposed. Hammoud et al. [7] present an architecture called Dynamic Cache Clustering (DCC) which benefits from shared and private L2 cache slices dynamically, and depends on the status and the working set of the program. In this approach, the whole or part of L2 cache slices is viewed as a single entity or cluster. However, DCC suffers from the

lack of awareness about the status of neighboring L2 slices. When DCC decides to expand the L2 cache of the target core, it does not know if any LLC sensitive thread is running on neighboring cores. In this case, it may apply more pressure on the L2 cache candidate. Also, the DCC expands or contracts by the power of two, which is sometimes overestimated. In other words, the sharing degree (the number of cores that share their whole L2 cache slices) does not increase or decrease gradually.

In this paper, we propose an adaptive cache clustering (ACC), where we monitor thread behaviors to see whether more cache space from surrounding neighbors could be allocated. Baseline last level cache (LLC) is private for each core and this cache space could be allocated to the neighbor core as a shared cache. In each epoch (four million clock cycles), we identify memory-intensive threads and assign extra cache spaces to reduce off-chip misses. We select L2 cache slices whose sharing degree is less than a threshold and are closer to the target core. We assess the effectiveness of our approach through several benchmarks from SPEC CPU2006 and PARSEC benchmark suits. We compare ACC with DCC, shared and private L2 caches. The applications are executed on a 4X4 2D NoC. Our evaluation across the mentioned benchmark suits shows that we can get up to 15% L1 latency speedup while ACC keeps the number of off-chip misses somehow close to that of shared L2 caches.

The rest of the paper is organized as follows: Section II reviews the related work for improving performance by optimizing the last level cache (LLC). In Section III we describe our proposed architecture which is called adaptive cache clustering. Section IV presents the results of the experiments performed to evaluate ACC. Section V concludes the paper and gives directions for future work.

2. Related Work

In this section, we survey the related work on various approaches to optimize the last level cache (LLC). In CMPs several cache organization approaches have been proposed to optimize cache allocation and reduce off-chip misses. We classify previous works on hardware and software approaches.

2.1. Hardware approaches

Kim et al. [5] present a non-uniform cache architecture (NUCA) and its performance benefits. Based on the mapping policy, NUCA architecture is classified into static-NUCA (S-NUCA) and dynamic NUCA (D-NUCA). In S-NUCA there is a unique mapping of data to blocks while in D-NUCA data can be moved among blocks. Therefore, to locate a block of data in several blocks a search mechanism is required [5]. Chishti et al. propose CMP-NuRapid [8] as an alternative for NUCA by decoupling data and tags. Also, tags are closer to the cores to reduce the search time. NuRAPID searches tags sequentially while NUCA searches them in parallel. Although NuRAPID is suitable, it is not scalable. On the other hand, LLC can be organized as private or shared among all cores or subset of them.

Private caches improve cache access latency but reduce useful cache capacity and increase off-chip misses. However, shared caches increase the cache access latency while they benefit the cache capacity. Researchers have proposed various approaches to benefit from both architectures [6] [9] [10]. Zhang et al. present victim cache in shared architecture to save evicted cache lines from the L1 cache into the L2 cache (LLC) [11]. Since victim caches reduce useful L2 caches, it may lead to an increase of L2 misses. Beckman et al. [12] propose an approach to monitor the number of hits to decide if replication is useful or not.

Chang et al. [9] propose cooperative caching (CC) by using centralized coherence engine to find L2 misses of local cores in the other caches to reduce off-chip misses. In other words, cooperation among private caches forms an aggregate shared cache. However, this approach is not scalable because of the centralized engine. Besides, by increasing the number of cores many comparisons are needed which leads to increased power consumption. Herrero et al. [10] present a distributed approach to solve the scaling limitation of CC. They propose a distributed coherence engine (DCE). Several DCEs are exploited to avoid bottleneck and reduce tag comparison. Each DCE is responsible for the management of some parts of the address space. In the Adaptive Shared/Private NUCA (ASP-NUCA) [6], each local private cache is split into private and shared sections. The repartitioning unit is responsible for extending the shared portion based on the number of misses and providing a uniform shared cache. In this case, private caches are not beneficially utilized when single-threaded applications are running on all cores. Several partitioning schemes of a shared cache among multiple co-running cores are presented to allocate cache slices based on a core's demands [13][14]. Utility-based cache partitioning monitors running applications by keeping tags of evicted blocks to divide cache among applications based on utility or benefit of cache for each co-running application [13].

Data search scheme is a key constraint in D-NUCA because of placement and migration. Several works propose various techniques to search data efficiently in D-NUCA banks by exploiting the prediction of the next bank access, and the parallelization [15][16]. These techniques need to access all banks to ensure that the block is in NUCA, while Lira et al. [17] propose HK-NUCA architecture to only access a subset of banks which potentially includes data. Sharifi et al. design an end to end QoS approach called METE based on control theory to manage cores, cache ways and bandwidths [18]. It uses an autoregressive moving average (AMR) model to get the application's performance.

In previous work, researchers have provided shared and private regions by dividing slices of LLCs. On the other hand, the others have presented a sharing degree in NUCA architecture, which shows the number of L2 slices that are shared among all cores. Hammoud et al. [7] present the dynamic cache clustering (DCC) and use the average memory access latency (AMAL) to dynamically resize the cache. Therefore, they vary the sharing degree of each core depending on the application phases. However, AMAL cannot estimate the cache requirement of an application when most of the accesses are hit to remote cores. Because for assessing AMAL when most remote L2 accesses are hit, the time for traversing the NoC is higher than the time to access the off-chip memory. Besides, the DCC extends the LLC of each core without evaluating the status of neighboring cores.

Our proposed approach is much closer to DCC. We believe our approach is more effective because it monitors the application's behavior precisely. ACC monitors the number of misses in the last thousand instructions of the surrounding L2 cache slices. Besides, ACC evaluates the sharing degree of surrounding cores when the current core needs more L2 cache slices to reduce off-chip misses. Besides, ACC overcomes the main limitation of DCC for selecting sharing degrees, as DCC considers some constant sharing degrees. However, ACC provides a flexible approach to select sharing degrees.

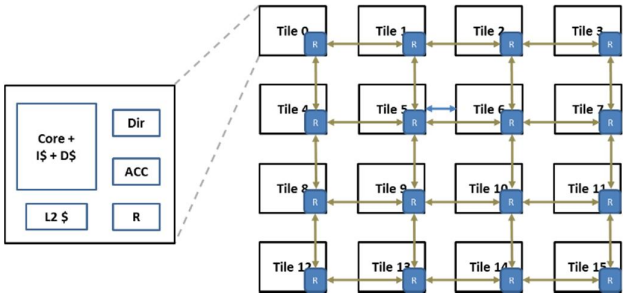


Fig. 1: adaptive cache clustering (ACC).

2.2. Software approaches

Software-based approaches provide libraries to enable cache management for programmers or operating systems (OS). These approaches are less flexible and lead to higher overhead than hardware approaches. Tam et al. [19] developed a software mechanism in OS by knowing the physical address to reduce contention in the shared L2 cache. Ding et al. designed libraries based on page coloring to identify high and low quality data [20].

Kandemir et al. [21] present a compiler-directed scheme to enhance the locality of data in shared cache. This approach targets allocation and scheduling in a way that different cores work on shared data and at the same time improving thus cache usage. Garcia et al. introduce an approach for mapping memory addresses to the last level cache by considering the distance between the L2 bank and a core [22]. Jin et al. proposed a mapping of page-level data to the cache line. In this approach, the operating system (OS) is responsible for deciding about the cache line placement [23]. Some researchers proposed a new cache organization [24] to utilize the last level cache efficiently by fast access to cache banks and taking advantage of the vicinity of cache banks to cores.

3. Proposed Approach

In this section, we describe the adaptive cache clustering and then consider an example to clarify it. Adaptive cache clustering tries to dynamically expand or shrink the size of the last level cache to reduce the average memory access time.

Memory intensive or non-intensive applications: The number of misses is used to decide how to classify applications into high, medium, and low memory intensive. We consider a threshold based on [25]. When it is greater than five, the application is memory intensive and when it is less than two, it is considered as a low memory intensive; otherwise, it is medium memory intensive [25]. In LLC cache-intensive applications, we try to find neighboring tiles with smaller sharing degree and smaller MPKIs. Applications that can get about 20% IPC gain compared to the ideal L2 cache are categorized as memory-intensive benchmarks [25][26][27].

3.1. Adaptive cache clustering

Our main objective is to expand or shrink the L2 cache of each core according to the need of the running application. In Adaptive Cache Clustering (ACC), each core locally changes the L2 cache size with knowing the behavior of the program or thread running on the neighboring cores. In ACC, a core is not allowed to use the neighboring L2 cache as a shared cache when it may heavily be used by running a memory intensive thread. It is designed to utilize the L2 cache efficiently. Each ACC block communicates with the surrounding tiles to get information about the utilization of their L2 cache.

In order to reduce the latency for moving ACC among routers, express virtual channels are exploited [28]. They ignore router pipelines by bypassing intermediate routers [28]. The ACC counts the number of misses per 1000 instructions (MPKI) in a period of four million cycles [29]. Therefore, each core is able to know how much L2 cache can be shared in run time. We assume a 2D 4x4 mesh network as a multi-core architecture shown in Figure 1. Each tile in this platform is composed of a core, L1 and L2 caches (it is the last level cache (LLC)), a router, and an ACC (Adaptive Cache Clustering) block.

3.2. Expansion of the L2 cache

Figure 2 shows Algorithm 1 to expand the L2 cache. First, for each tile, the number of reachable tiles based on their hops from the current tile is put in a list (by using Get_All_tiles_hop()). Then the number of misses in the previous epoch is used for estimation (Line 5). We get the number of tiles based on the ascending number of hops from the current tile (Get_tiles_hop (L,i))and then sort them according to their sharing degree (Sort(L1)). Then we evaluate the current list of possible candidates to find a tile whose MPKI was less than 2 in the previous epoch (Lines 11-18). The tiles whose sharing degrees are less than 2 are candidates for expanding the L2 cache. In this case, those tiles are used as a shared L2 cache.

```
1  /*Algorithm 1 to expand L2 cache*/
2
3  List L = Get_All_tiles_hop();
4  int i = 0;
5  if( MPIK >= 5)
6  {
7   while(L is not empty){
8    List L1 = Get_tiles_hop(L,i);
9    Queue sharing_degree = Sort(L1);
10
11   for ( int j =0;
12     j < sharing_degree.Length; j++) {
13        if(L1[j].MIPK < 2)
14        {
15                L1[j].SD++;
16                return  L1[j].ID;
17        }
18   }
19   i++;
20  }
21 }
```

Fig. 2: Algorithm 1 to expand the L2 cache.

3.3. Contract the L2 cache

ACC executes Algorithm 2 which is shown in Figure 3 to shrink the L2 cache size. For this purpose, we evaluate MPKI and SD (sharing degree) in every epoch. In this case, if MPKI is less than 2 and SD is greater than 2. We reduce SD by one. ACC sub-module in each tile includes negligible hardware overhead. The area overhead of ACC is about 1.6% compared to the router area. It exploits local caches and intelligently expands and shrinks the L2 cache size of each core.

```
1 /*Algorithm 2 to contract L2 cache*/
2
3 if( MPIK <= 2 && SD >= 2)
4 {
5  SD--;
6  Delete last entry
7   from list
8 }
```

Fig. 3: Algorithm 2 to shrink the L2 cache.

3.4. Case study

For illustrative purposes, we present an example to compare expanding L2 cache size in ACC and DCC approaches when both cores are running L2 cache intensive applications. In this case, the ACC approach does not expand the L2 cache size, because each core knows the status of the surrounding L2 cache slices. While, in the other approaches such as DCC, they are unaware of sur-rounding L2 cache pressures. Therefore, they expand their L2 cache size as a shared one which leads to more misses and inter-core misses, because they are racing to use the other cache. Although, L2 shared cache is increased but its effective size is slightly augmented.

The second limitation of DCC is that the cluster dimension is increased or decreased by a factor of two which implies some limitations on possible candidates. Figure 4 shows the difference between DCC and ACC approaches. DCC presents five distinct sharing degrees (sharing degrees: 1,2,4,8,16) for each core to expand L2 cache slices. The dotted rectangles present two of the sharing degrees (1,2) in Figure 4-a. For example, core 6 has a sharing degree of 2 with core 7 in the green rectangle and a sharing degree of 4 with cores 2, 3, and 7. Although the DCC approach provides higher performance due to its simple sharing degree configuration, it causes some constraints. For instance, in Figure 4-a cores 5 and 6 cannot mutually share their L2 cache slices, based on the dotted rectangles. ACC provides a more flexible clustering than DCC because it allows a core to form a cluster with its surrounding cores, presented in Figure 4-b. For example, core 6 can share L2 cache slices with cores 2, 5, 7, and 10.

4. Experimental Results

4.1. Target architecture

In this section, we present the evaluation of ACC with different benchmarks. These benchmarks are categorized into multiprogramming and multi-threaded ones. Several applications are carefully selected from PARSEC, and SPEC CPU2006 benchmarks to cover various categories such as high-performance computing, financial domain, signal processing, and media processing. Table 1 presents the selected benchmarks. First, these benchmarks are classified in memory intensive and non-intensive in order to comprehensively evaluate the proposed approach. For this purpose, we have followed the approach presented in [25][26][27].

4.1.1. Multithreaded benchmarks

In this section, we consider some applications from the PARSEC benchmark [30] including multithreaded programs to evaluate the ACC approach precisely. These programs are spawned on a 4x4 2D mesh NoC architecture. Several applications are chosen from these suites to cover programs from different domains such as data mining, financial analysis, and media processing.

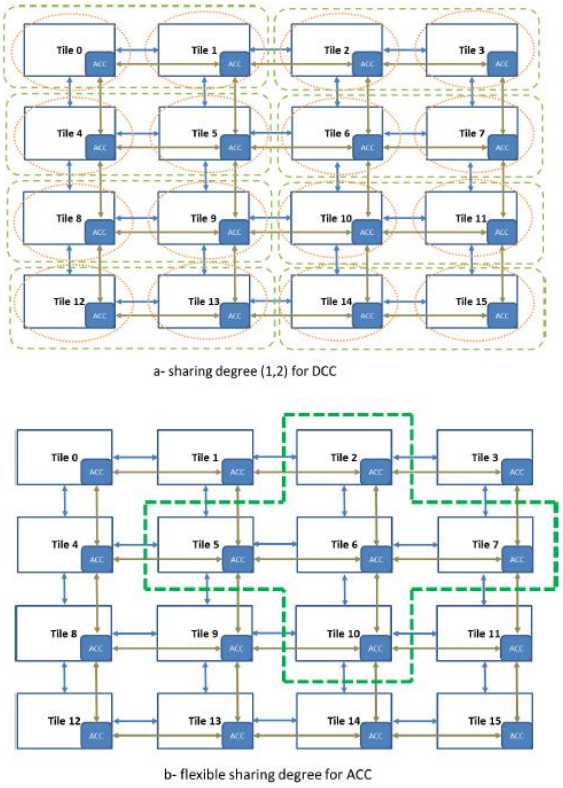


Fig. 4. Different clustering for DCC and ACC approaches.

PARSEC benchmarks include three phases called Initialization, Region of Interest (ROI), and cleanup. The master thread initializes the data before spawning worker threads in the Initialization phase. ROI includes the parallelized code section. It exhibits parallelism at the level of a macro-block and ROI is thus distributed among cores. Both Initialization and cleanup phases are executed in a single core. All the programs use standard Pthread APIs to provide synchronizations by using locks and barriers. An exception is Freqmine, which uses OpenMP for parallelization. Results are gathered from the simlarge dataset inputs as a working set size. We consider six applications from this suite presented in Table 1. Also, it shows parallelization approach for the aforementioned programs. Blackscholes, Swaptions, X264 are

data-level parallel programs while Fluidanimate and Streamcluster both have a streaming approach.

Applications like streamcluster and canneal are memory intensive because they have a large number of off-chip misses per 1000 instructions. On the other hand, Fluidanimate, Blackscholes, Swaptions, and X264 are memory non-intensive because of a low number of misses.

4.1.2. Multiprogramming benchmarks

We also evaluate multiprogramming workloads with SPEC CPU 2006 workloads by choosing eight Multiprogramming workloads which are shown in Table 2. These benchmarks cover a diverse area of applications such as artificial intelligence, video compression, etc. gombk, hmmer, h264ref, and sjeng use integer instruction while lbm, gromacs, soplex, and dealII are floating-point benchmarks. In addition, those workloads are classified into two LLC intensive and non-intensive benchmarks. Benchmarks such as gobmk, lbm, soplex, and sjeng are memory intensive applications while dealII, hmmer, gromacs, h264ref are memory non-intensive.

We selected six groups of applications based on their LLC misses to be run in parallel. These groups shown in Table 2 are chosen to cover workload variability in running multiple applications. In each group, four different applications are running on 16-core processors. Mix01 includes four LLC intensive applications, while Mix02 contains three LLC intensive applications and one LLC non-intensive application. Mix03 and Mix04 include two LLC intensive applications while Mix04 and Mix05 contain three LLC non-intensive applications.

4.2. Simulation methodology

The experiments were performed using Gem5 simulator assuming the Alpha ISA [31]. The simulations were performed for a 16-node CMP arranged as a 4x4 mesh. In our experiment, we use 16 threads to run each application and employ the thread per core assignment approach. Each processor node has a private L1 cache of 32 KB and 1 MB L2 cache (16 MB shared distributed L2 for the entire system). There are 4 memory controllers at the corners. The major parameters of processors and architectures are presented in Table 3.

4.3. Evaluation metrics

We use weighted speedup to quantify the performance of the system while multiple threads or applications are running concurrently. Weighted speedup presents a reduction in execution time. Therefore, for a system with N running threads, it is given by Eq. (1):

$$Weighted\ speedup = \sum \frac{IPC_i}{IPC_{alone}} \quad (1)$$

IPC_i is the IPC of an application when it is running with other applications, $IPC_{alone}$is the IPC of the same application when it is running in isolation.

Table 1: List of benchmarks

SPEC CPU 2006		
gobmk	Integer	Yes
lbm	Floating point	Yes
hmmer	Integer	No
gromacs	Floating point	No
h264ref	Integer	No
soplex	Floating point	Yes
sjeng	Integer	Yes
dealII	Floating point	No
PARSEC		
Blackscholes	Financial analytics	No
Swaptions	Financial analytics	No
X264	Media processing	No
canneal	CAD tool	Yes
Fluidanimate	Animation	No
Streamcluster	Data mining	Yes

TABLE 2: Multi-program workloads

Workloads	Descriptions	Memory intensive
Mix01	gobmk+lbm+ soplex+ sjeng	high
Mix02	dealII+lbm +soplex + sjeng	high
Mix03	dealII+hmmer+ soplex + sjeng	medium
Mix04	gobmk+lbm+gromacs+h264ref	medium
Mix04	gobmk + h264ref + gromacs + sjeng	low
Mix05	gobmk+h264ref+ dealII + hmmer	low

TABEL 3: System configuration

Multi-core system configuration	
Number of cores	16 cores
L1 cache	32KB, 4-way
Cache line size	64 Bytes
L2 cache	16MB (1MB per core)
Memory access time	400 cycles
Memory controllers	4 Memory controllers at corners
NoC topology	4x4 2D Mesh, XY routing

Also, we consider three other metrics, namely the number of off-chip misses (L2 misses), and the average L1 miss latency. We consider the first two metrics to comprehensively evaluate

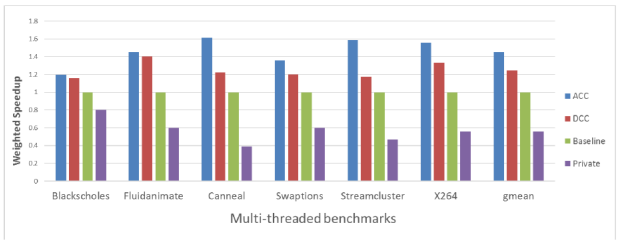
the proposed approach because shared caches reduce off-chip misses while private caches decrease the miss latency.

5. Experimental Results

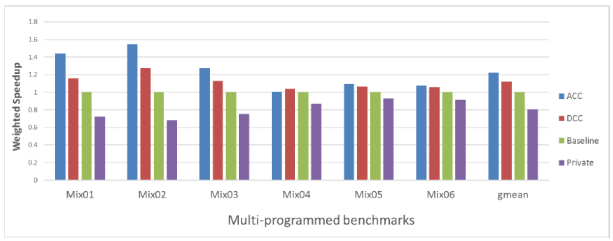
We compare our approach with the baseline where the L2 cache is shared and distributed among cores. We also compare the approach with dynamic cache clustering [7] and private L2 caches. The baseline, private L2 cache, and dynamic cache clustering are marked as Baseline, Private, and DCC. For all figures, the numbers are normalized to baseline ones.

5.1. Performance

Figure 5 shows the weighted speedup of ACC over the performance of the baseline, DCC, and Private. The gmean bar indicates the geometric mean of all 6 workloads. For memory sensitive applications, such as streamcluster and canneal we observe higher performance because the workload is sensitive to the amount of available cache. Figure 5-b presents the results for multi-program workloads. The results show higher performance for memory-intensive workloads (Mix01, and Mix02). On average, ACC improves performance by 18% over baseline for multi-threaded benchmarks and 9% for multi-programmed applications.



a) Multi-threaded benchmarks

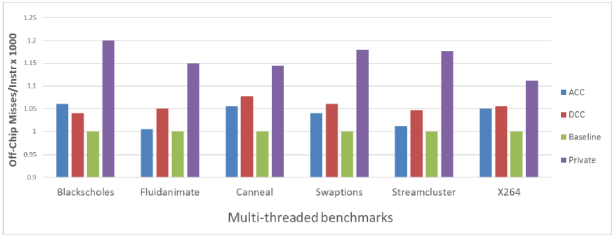


b) Multi-programmed benchmarks

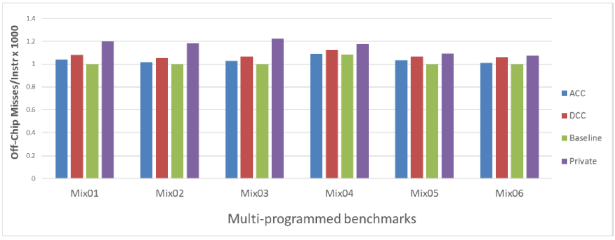
Fig. 5: Performance of workloads.

5.2. Off-chip misses

Figure 6 shows off-chip misses per thousand instructions. The numbers are normalized to the results obtained from the baseline architecture using the same configuration. The baseline approach leads to the minimum number of misses because it exploits larger L2 cache sizes. As we can observe, ACC and DCC exhibit the number of L2 misses close to that of the baseline architecture and ACC outperforms up to 6% on average. Figure 6) b shows that the number of off-chip misses in ACC is much closer to the baseline architecture and outperforms up to 4% on average.



a) Multi-threaded benchmarks



b) Multi-programmed benchmarks

Fig. 6: Off-chip misses per thousand instructions.

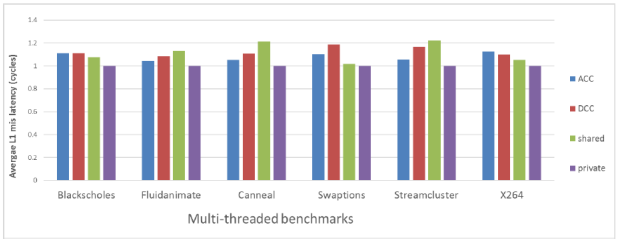
5.3. Average L1 miss latency

Since ACC intelligently allocates neighboring caches, it reduces L2 access latency efficiently and it outperforms the other configurations. ACC tries to identify neighboring tiles with less pressure on L2 cache slices so that it will cause a minimum average L2 miss latency. Therefore, it will reduce the average L1 miss latency, shown in Figure 7. We can get speedup up to 15% compared to DCC on average and up to 11% compared to baseline in multi-threaded benchmarks. Although private architectures impose little pressure on NoC because of non-sharing data among cores, it increases the number of off-chip misses, which implicitly adds more traffic on the interconnection network. The second graph of Figure 5 shows that we can get speedup up to 18% for Mix01 which is a memory-intensive application. The ACC approach tries to exploit caches that belong to cores with the least distance which leads to less average L1 miss latency than the DCC approach.

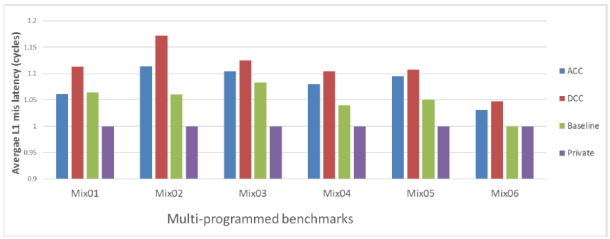
6. Conclusion

In this paper, an adaptive cache clustering algorithm called ACC was proposed. It provides efficient usage of the L2 cache by monitoring the behavior of running applications. We designed a distributed monitoring approach that identifies the extra amount of L2 cache that an application needs to provide higher performance. Moreover, the number of misses per thousand instructions was considered as a criterion to distinguish between memory intensive and memory non-intensive applications. Also, we allocated an L2 cache slice which needs fewer hops to be accessed by the target core.
Simulations were performed for ACC, DCC, shared, and private L2 caches under several multi-threaded and multi-programmed workloads. The results demonstrate the effectiveness of our approach. The main advantage of ACC is

that it leads to up to 15% average L1 miss latency while it exhibits off-chip misses comparable to shared L2 caches.



a) Multi-threaded benchmarks



b) Multi-programmed benchmarks

Fig. 7: Average L1 miss latency

References

[1] L. Pinho, E. Quinones, and A. Marongiu, High-Performance and Time-Predictable Embedded Computing, River Publishers, 2018.

[2] C. M. Wittenbrink, E. Kilgariff, A. Prabhu, "Fermi gf100 gpu architecture," IEEE Micro, vol. 2, no. 31, pp. 50-59, 2011.

[3] S. Bell, B. Edwards, J. Amann, et al., "Tile64-processor: A 64-core SoC with mesh interconnect," in IEEE International Solid-State Circuits Conference-Digest of Technical Papers, 2008.

[4] S. Vangal, J. Howard, G. Ruhl, S. Dighe et al., "An 80-tile 1.28 TFLOPS network-on-chip in 65 nm CMOS," in IEEE International Solid State Circuits Conference, 2007.

[5] C. Kim, D. Burger, S.W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in International Conference on Architectural Support for programming languages and operating systems, 2002.

[6] H. Dybdahl, P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in International Symposium on High Performance Computer Architecture, 2007.

[7] M. Hammoud, S. Cho, Sangyeun, R. Melhem, "Dynamic cache clustering for chip multiprocessors," in International Conference on Supercomputing, 2009.

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," in International Symposium on Computer Architecture, 2005.

[9] J. Chang and G.S. Sohi, "Cooperative caching for chip multiprocessors," ACM SIGARCH Computer Architecture News, vol. 2, no. 34, pp. 264--276, 2006.

[10] E. Herrero, J. Gonz ález, Jos´e, R. Canal, "Distributed cooperative caching: An energy efficient memory scheme for chip multiprocessors," IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 23, pp. 853--861, 2011.

[11] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in International Symposium on Computer Architecture, 2005.

[12] B. Beckmann, M. Marty, D.A. Wood, "ASR: Adaptive selective replication for CMP caches," in International Symposium on Microarchitecture, 2006 .

[13] M. Qureshi, Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in International Symposium on Microarchitecture, 2006.

[14] J. Chang, G. Sohi, "Cooperative cache partitioning for chip multiprocessors," in International Conference on Supercomputing, 2007.

[15] S. Akioka, F. Li, K. Malkowski, P. Raghavan, M. Kandemir, M.J. Irwin, "Ring data location prediction scheme for non-uniform cache architectures," in International Conference on Computer Design, 2008.

[16] R. Ricci, S. Barrus, D. Gebhardt, R. Balasubramonian, "Leveraging bloom filters for smart search within NUCA caches," in Workshop on Complexity-Effective Design, 2006.

[17] J. Lira, C. Molina, A. Gonz ález, "Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors," in IEEE International Parallel \& Distributed Processing Symposium, 2011 .

[18] A. Sharifi, S. Srikantaiah, A. Mishra, M. Kandemir, "METE: meeting end-to-end QoS in multicores through system-wide resource management," in International Conference on Measurement and modeling of computer systems, 2011.

[19] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in Workshop on the Interaction between Operating Systems and Computer Architecture, 2007.

[20] X. Ding, K. Wang, and X. Zhang, "ULCC: a user-level facility for optimizing shared cache performance on multicores," in ACM Symposium on Principles and practice of parallel programming, 2011.

[21] M. Kandemir, S. Muralidhara, S. Narayanan, Y. Zhang, O. Ozturk, "Optimizing shared cache behavior of chip multiprocessors," in International Symposium on Microarchitecture, 2009.

[22] A. Garcia-Guirado, R. Fern´andez-Pascual, A. Ros, and Jos´e M. Garc´ıa, "DAPSCO: Distance-aware partially shared cache organization," ACM Transactions on Architecture and Code Optimization, vol. 8, no. 4, pp. 1-19, 2012 .

[23] L. Jin, H. Lee, and S. Cho, "A flexible data to L2 cache mapping approach for future multicore processors," in Workshop on Memory system performance and correctness, 2006.

[24] Z. Guz, I. Keidar, A. Kolodny, U. Weiser, "Utilizing shared data in chip multiprocessors with the Nahalal architecture," in Annual Symposium on Parallelism in algorithms and architectures, 2008.

[25] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P., "Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs," in International Symposium on High-Performance Comp Architecture, 2012.

[26] F. Guo, Y. Solihin, L. Zhao, R. Iyer, "A framework for providing quality of service in chip multi-processors," in International Symposium on Microarchitecture, 2007.
[27] Y. Xie, G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 174--183, 2009.
[28] A. Kumar, L.-S. Peh, P. Kundu, N. K. Jha, "Express virtual channels: Towards the ideal interconnection fabric," ACM SIGARCH Computer Architecture News, vol. 35, no. 2, pp. 150--161, 2007.
[29] B. Lin, M.B Healy, R. Miftakhutdinov, P.G Emma, Y., "Duplicon cache: mitigating off-chip memory bank and bank group conflicts via data duplication," in International Symposium on Microarchitecture, 2018.
[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in International Conference on Parallel architectures and compilation techniques, 2008.
[31] N. Binkert, B. Beckmann et al., "The gem5 simulator," ACM SIGARCH computer architecture news, vol. 39, no. 2, pp. 1-7, 2011.

Masoud Dehyadegari received his Ph.D. degree from the University of Tehran, Tehran, IRAN, in 2013 in computer engineering. He is currently an Assistant Professor of the school of computer engineering with the K. N. Toosi University of Technology, Tehran, IRAN. From September 2011 until December 2012, he was a visiting scholar at the University of Bologna, Italy. His research interests include Low-power system design, Network-on-chips, and Multi-Processor System-on-chip.
Email: dehyadegari@kntu.ac.ir

Paper Handling Data:

Submitted: 11.05.2018
Received in revised form: 03.01.2020
Accepted: 04.02.2020
Corresponding author: Dr. Masoud Dehyadegari
Affiliation of the corresponding School of computer engineering with the K. N. Toosi University of Technology, Tehran, IRAN