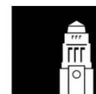

This is a repository copy of *Partially shared cache and adaptive replacement algorithm for NoC-based many-core systems*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/145787/

Version: Accepted Version

Article:

Yang, P, Wang, Q, Ye, H et al. (1 more author) (2019) Partially shared cache and adaptive replacement algorithm for NoC-based many-core systems. Journal of Systems Architecture, 98. pp. 424-433. ISSN 1383-7621

https://doi.org/10.1016/j.sysarc.2019.05.002

© 2019 Elsevier B.V. All rights reserved. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0/.

Reuse

This article is distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) licence. This licence only allows you to download this work and share it with others as long as you credit the authors, but you can't change the article in any way or use it commercially. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Partially Shared Cache and Adaptive Replacement Algorithm for NoC-based Many-core Systems

Pengfei Yang, Quan Wang, Hongwei Ye, Zhiqiang Zhang

Abstract—. The Network-on-Chip(NoC) is a promising alternative to traditional bus-based architectures that has been widely applied to interconnect multi/many-core systems due to its scalable and modular design. Undoubtedly, the memory wall problem is one of the most important challenges; however, this problem can now be somewhat be alleviated by cache subsystems. In this paper, to overcome the high resource consumption and low data-sharing rate problems of the private cache scheme, we propose a partially shared cache structure and a corresponding replacement algorithm based on a mesh NoC. In this scheme, the L2 cache is shared by each group of four cores that connected as a cluster to a given node by the local bus. To maximize the performance of this partially shared cache structure, we propose a core-aware re-reference interval prediction (CA-RRIP) replacement algorithm. The algorithm performs dynamic virtual partitioning on the partially shared cache; the core that initiated the cache access request will be given top priority when a cache area needs to be replaced or inserted. This approach guarantees cache exclusivity and can mitigate interactions among cores using different access patterns. We implement the traditional private, the proposed partially shared and the row-shared cache subsystems in our experiments. The comparisons indicate that the overall system resource occupation can be reduced by 20% with the same number of cores, and the instructions per cycle(IPC) of the system could increase by up to 49.2%. Moreover, the system throughput(STP) increased by an average of 5.89%. Our experimental results showed that the proposed CA-RRIP algorithm also reduces the average cache miss rate of the system under various cache access patterns.

Index Terms—Many-core System, NoC, Cache Structure, Replacemant Algorithm.

1 INTRODUCTION

The last decade has witnessed significant improvement in embedded system performance, because many cores can now be integrated into a single chip package. As the number of cores continues to increase, the current bus architecture is no longer suitable. Traditionally, cores are usually interconnected by single or multiple layers of shared buses, which provide benefits such as low system resource cost, simple topology and extensibility. However, such structures have some obvious disadvantages, including complicated design process, high power consumption, unpredictable delays and non-scalability. Therefore, the Network-on-Chip(NoC), which integrates a large number of processing elements(PEs), memory elements and a communication network connecting them, has been developed to replace traditional bus structures to improve the communication efficiency among different cores [1] and the overall system performance [2]. This new trend brings about a set of challenges, one of which is cache distribution among cores.

The gap between the speed of computing resources and storage resources is growing rapidly. The memory hierarchy, which consists of processor registers, different levels of caches and main memory, has an inevitable impact on all the parameters of a system, including area requirements, power consumption and performance. On-chip caches play important roles in improving processor performance. However, the current cache structure is another bottleneck that limits the system performance. Thus far, three levels of cache structure have been widely applied in modern multi/many-core systems [3] [4]. As shown in Figure 1, the L1 and L2 cache are private, while the L3 cache is shared by all the cores. Although this structure provides each core with fast access to resources, it requires excessive system resources to construct such private cache structures for each core. The structure may also cause destructive interference between threads, leading to thrashing, unfairness and poor Quality-of-Service(QoS) [4] [5]. Thus, to reduce the cache miss rate and access data from other cores' on-chip cache memory, a better cache replacement algorithm is required. Unfortunately, the existing algorithms are always access pattern and application specific, and no single replacement algorithm is applicable to all the various access patterns. For example, the Least Recently Used(LRU) is the most effective algorithm because it is easy to implement. It has high performance for the recency-friendly access pattern which according to that more recently used blocks are likely to be referenced again, it performs unsatisfactorily for the trashing access pattern or the streaming access pattern. One more thing, it does need a number of bits to maintain a record for each block, which contains details such that when a block is accessed before. Multiple-tasks with different access patterns may run simultaneously within a many-core system, the overall system performance may be severely

- *Pengfei Yang, Quan Wang and Hongwei Ye are with the School of Computer Science and Technology, Xidian University. E-mail: pfyang@xidian.edu.cn*
- *Zhiqiang Zhang is with the School of Electronic and Electrical Engineering, University of Leeds, UK.*

The paper is supported by the Natural Science Foundation of China(NSFC: 61702395,61572385,61711530248), the Fundamental Research Funds for the Central Universities(No. JBF180301)

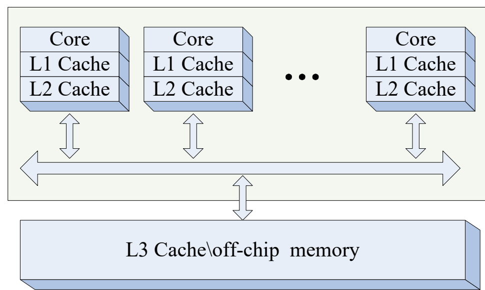

Fig. 1. Typical storage architecture of many-core system

affected by adopting any of the existing cache replacement algorithms; therefore, a new cache replacement schemes should be developed.

To increase the cache performance of NoC-based many-core systems, this paper constructs a partially shared cache structure and proposes the core-aware re-reference interval prediction(CA-RRIP) algorithm based on a cluster-shared cache. The proposed cache structure not only reduces system size but also improves system throughput(STP); in addition, the replacement algorithm reduces the average cache miss rate and improves system performance. The main contributions of this paper are as follows:

1) A new cache structure is constructed by integrating the advantages of bus-on-chip with NoC. The proposed L2 cache is shared by four cores that connected to the given node by the local bus. This optimized structure reduces the overall resource consumption of the system and improves the efficiency of L2 cache data sharing among cores.
2) The CA-RRIP cache replacement algorithm is proposed based on the designed cache structure. Unlike a fixed size L2 cache in the traditional cache structure, the algorithm performs dynamic virtual partitioning on the partially shared cache, which mitigates the interactions among cores using different access patterns. When a cache area needs to be replaced or inserted, the core that initiated the cache access request is given top priority, thereby guaranteeing cache exclusivity. The CA-RRIP algorithm reduces the cache miss rate under various cache access patterns and raises the overall performance of many-core systems.

We implement the traditional private, the proposed partially shared and the row-shared cache subsystems in our experiments. A comparison revealed that the size of the system can reduced by 20% with the same number of cores; the IPC of the system can increase by up to 49.2%, and the system throughput achieved an average boost of 5.89%. Our experimental results showed that the proposed CA-RRIP algorithm can also reduce the average cache miss rate with a varity of cache access patterns.

The remainder of this paper is organized as follows. Section II introduces related works on cache structures and replacement algorithms. Section III proposes the optimized cache structure and provides the details of the CA-RRIP algorithm. The experimental results are presented in Section IV. Finally, a summary and future work are given in Section V.

2 Related Work

There are two types of approaches to increase the cache performance of many-core systems: modifying the cache structure and designing a new cache replacement algorithm. We elaborate these approaches in the following sub-sections.

2.1 Cache structure

Various cache architectures have been proposed since different parameters have direct impacts on the efficiency of the memory design. Zhao et al. [6] proposed a directory-based collaborative cache scheme for chip-level multiprocessing(CMP) systems. In this scheme, a distributed L2 cache is shared by different cores through the directory. This scheme meets the requirements of different task loads, and the data could be easily transferred to other L2 caches when they had to be replaced. This scheme also reduced the number of off-chip memory accesses by reducing the frequency data exchanges between the cache and the off-chip memory. However, as the number of cores increased, the cache maintenance costs also increased, and the system's scalability decreased due to the shared bus-based storage structure. Feng et al. [7] proposed a non-inclusive cache that fully utilized only the last-level cache space for the CMP. However, this solution optimized the last level cache only; it did not reduce the average access latency of the NoC. Zhang et al. [8] proposed a victim cache scheme based on the NoC interconnected architecture that combined the advantages of the shared and private L2 caches and reduced the access latency of the L2 cache by saving the to-be-released data into the L2 cache. However, this solution required saving a copy of the L2 cache, which reduced the L2 cache utilization ratio and reduced global sharing of the L2-cache among all the cores. Therefore, the maintenance cost increased rapidly when the number of cores increased. To achieve energy-efficient optimization, Sampaio et al. [9] proposed an approximation-aware multi-level cell STT-RAM cache architecture. The technique attempeted to maximize the application performance while minimizing the energy consumption. Bengueddach et al. [10] proposed an optimal two-level cache that employed the dynamic reconfiguration technology in the multiprocessors and reduced the overall energy consumption of the hardware/software architecture. Mallya et al. [11] proposed a way halted prediction cache structure that attempted to save system energy by reducing the number of active ways to one when the prediction was hit. Naderializadeh et al. [12] proposed an achievable scheme and a particular cache placement pattern to maximize the overall efficiency of cache-aided transmits. Jadidi et al. [13] proposed a criticality-aware compressed LLC that favours lower latency over higher capacity based on the criticality of the data blocks; however, in this scheme, the last-level cache (LLC) is logically shared but physically distributed among cores. Merino et al. [14] apportioned the L2 cache among the private caches and it cache dynamically according to different cache access patterns. This approach

reduced the average access latency by adjusting the cache algorithm to save the private data to the private cache of the corresponding core. However, the dynamic partitioning process required too many system resources and did not provide a better solution for the problem of many-core systems based on a NoC.

2.2 Cache replacement algorithm

In the cache memory, when all the blocks in a set of cache become full and a new block from main-memory needs to be placed in cache, then the cache controller has to be discarded a line from cache set and replace it with the new block from main-memory. All cache replacement algorithms try to reduce the cache miss rate for one or more cache access patterns, because the cost of a cache miss is the largest performance penalty for a cache replacement algorithm. The Least Recently Used(LRU) [15], the Least Frequently Used(LFU) [16], Not Recently Used(NRU) [17], the Dynamic Aware Insertion Policy(DIP) [18], and related improved algorithms are all in common use. The LRU provides good performance for workloads with high-level data locality, but limits the system performance when predicted near-reference intervals are incorrect.

The LFU and LRU policies are consistent, except that the LFU used the most recently uses frequency for the insert and replace operations. Many previous researchers studied and analyzed the performance of cache replacement algorithms with several applications. The results showed that the LRU has the better performance than other algorithms just like Random, FIFO and LFU [19]. Many improved replacement algorithms based on these basic policies exist well. For example, Alghazo et al. [20] proposed a second chance-frequency - least recently used(SF-LRU) replacement algorithm that combined the LRU and LFU and considered not only the number of cache blocks used, but also the re-reference interval when the cache block was popped up.

However, this algorithm fit only for caches with recency-friendly workloads. Qureshi et al. [18] proposed a DIP cache algorithm that contained two insertion strategies: an LRU Insertion Policy(LIP) and a Binominal Insertion Policy(BIP). The LIP inserted a new cache block in place of the most recently used (MRU), while the BIP inserted a new cache block in place of the least recently used or the most recently used with a small probability. The algorithm set the sample cache group and determined the insertion policy based on the sample cache miss rate. This scheme achieved better cache performance for recency-friendly and trashing access patterns. Jaleel et al. [21] proposed a thread-aware dynamic insertion policy(TADIP) based on the DIP. TADIP considered the memory requirements of the concurrently executing applications, and chose the LIP or BIP according to the core's state. They also proposed re-reference interval prediction(RRIP) [22] by setting a correlation counter for each cache block that represented the re-reference interval value(RRIV) of the block. TADIP and RRIP enhanced the system caching performance, but did not solve the interference problem caused by other cores sharing the cache. Qureshi et al. [23] also proposed utility cache partitioning(UCP), which had the same insertion and promotional policies as LRU. It determined the size of the cache based on the value of the utility monitor. The algorithm efficiently used the cache space to improve the cache performance. Beckmann et al. [24] proposed an adaptive selective replication(ASR) using the shared cache and private cache architecture. The algorithm dynamically monitored program behaviour and conducted data replication only if the cost of cache replication exceeded the cost of cache miss. Dong et al. [25] proposed a new dynamic shared-cache management scheme called co-optimizing locality and utility(CLU) in thread-aware capacity management based on locality and utility values. The algorithm utilized the same apportioning method as the UCP; the only difference was that the CLU algorithm used the LRU and BIP values to determine the utility value.

Most of the current replacement algorithms are based on typical three-level storage architecture; they ignore the interactions among cores with different access patterns and are unsuitable for the partially shared cache structures. These algorithms always result in poor cache performance for the mixed access patterns of many-core systems. To resolve this problem, this paper proposes the CA-RRIP algorithm, which is a core-aware re-reference interval prediction algorithm. The algorithm performs dynamic virtual partitioning on the partially shared cache which mitigates the interactions among cores using different access patterns. When a cache area needs to be replaced or inserted, the core that initiated the cache access request is given top priority which guarantees cache exclusivity when it is being used. The CA-RRIP algorithm reduces the cache miss rate under a varity of cache access patterns and improves the overall performance of many-core systems.

3 Cache Structure and Replacement Algorithm Optimization

3.1 Motivation

To verify the advantages of NoC, we built a typical NoC system with 16 nodes on the SoCKit development board as shown in Figure 2. We designed the system hardware and software based on theoretical NoC research, including fault-tolerant topology [26], the adaptive routing algorithm [27], heuristic task scheduling and the mapping algorithm [28]. Regarding the memory structure, the typical three-level cache structure and LRU replacement algorithm put us at a disadvantage. The effects of different cache parameters and configurations of the platform are investigated. One reason is that the L1 and L2 cache are private, it takes up too much system resources to guarantee enough cache space for each processing element. One more thing, the fixed size L2 cache makes it difficult to fit all the cache access patterns. Another reason is that the hit/miss ratio of a cache subsystem is an application/access pattern specific parameter, which makes finding a low complexity and energy efficient replacement algorithm actually a difficulty.

There are four types of cache access patterns: recency-friendly, thrashing, streaming and mixed access patterns [29]. To better represent the differences between these patterns, we provide the following descriptions [29]: a_i represents the address of a cache line, $(a_0a_1a_2...a_{k-1})$ represents a temporal sequence of k_address references, $(a_0a_1a_2...a_{k-1})^N$ denotes that the sequence repeats N times

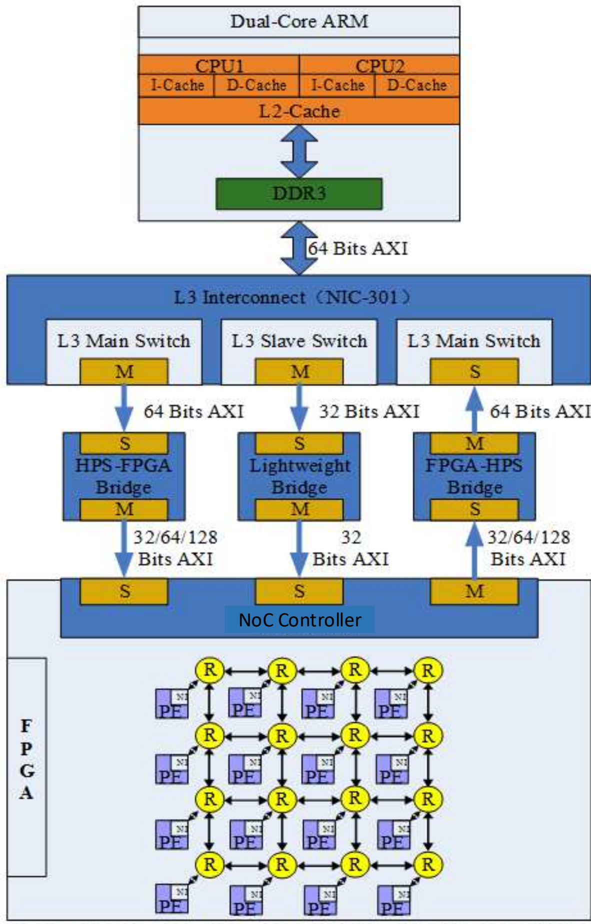

Fig. 2. NoC Implementation based on SoCKit

and $P_i(a_0a_1a_2...a_{k-1})^N$ is the probability that the temporal sequence will occurs.

$$(a_0, a_1, a_2, ..., a_{k-1}, a_{k-1}, ..., a_2, a_1)^N \tag{1}$$

$$(a_0a_1a_2...a_{k-1})^N(K > cachesize) \tag{2}$$

$$(a_0a_1a_2...a_{k-1})(K = \infty) \tag{3}$$

$$\begin{gathered}[(a_0, a_1...a_{k-1}, a_{k-1}...a_1)^A P_i(a_0a_1...a_{k-1}...a_m)]^N \\ [(a_0, a_1, a_2...a_{k-1})^A P_i(b_0b_1b_2...b_m)]^N\end{gathered} \tag{4}$$

In the recency-friendly access pattern, k is equal to or less than the total number of cache blocks, and $N > 1$(Formula 2). This access pattern benefits from the LRU replacement algorithm; replacement algorithms other than LRU can degrade the system performance. The trashing access pattern(Formula 3) is similar to the recency-friendly access pattern, but in each access cycle, k is larger than the number of cache blocks. If the LRU replacement algorithm is adopted in the trashing access pattern, the algorithm will pop up the data that will be accessed the next time during current operation, causing frequent cache replacement. The LRU provides no cache hits unless the cache size is sufficiently large to hold all k entries of the access pattern. For the streaming access pattern(Formula 4), k tends towards infinity, thus, the data to be accessed do not have local correlation features, or long time intervals occur between accesses of the same data. Consequently, streaming access patterns receive no cache hits when using any replacement algorithm. The mixed pattern is a combination of the different patterns mentioned above.

Real-world many-core system applications required a variety of cache access patterns. In addition, the cache access pattern of a core changes with program requirements change. Thus, a cache replacement algorithm must be designed to achieve good performance across the different access patterns. We employ the LRU replacement algorithm and the system experiences significant performance fluctuations for various tasks and occupies many system resources. Then we implement and compare the optimal cache structure and the CA-RRIP replacement algorithm.

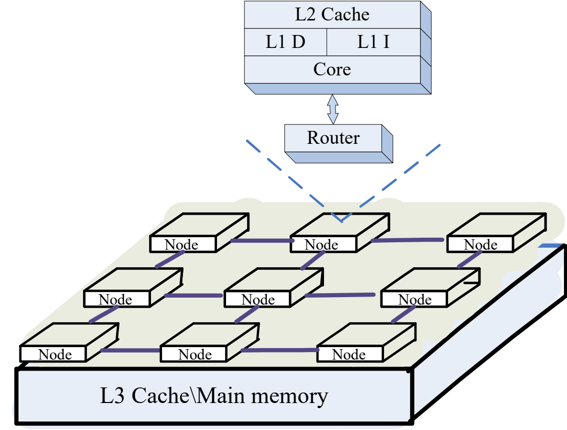

Fig. 3. Typical cache structure of NoC

3.2 Cache Structure Optimization

The typical cache architecture of NoC is shown in Figure 3, the nodes are connected by a high-performance network in a silicon chip. Every node consists of two parts, one is the router, which connects to its neighbour nodes and the local PE. The other part is the PE, it is made up of a core and the cache. The core executes the task, and the cache is supposed to knock down the growing disparity between CPU clock and off-chip memory.

This structure solves communication problem between the cores in many-core sysems satisfactorily, but it also introduces new problems, such as the overhead occupation of system resources and the access latency. The private L1 and L2 caches consume a large amount of system storage resources, and the data-sharing latency among cores is large because low-radix networks always introduce high network latency caused by long diameters. In addition, many tasks require multiple cores to work in parallel, which involves massive and frequent data sharing among the cores. However, this separate private cache structure increases system power consumption and causes performance fluctuations across different tasks.

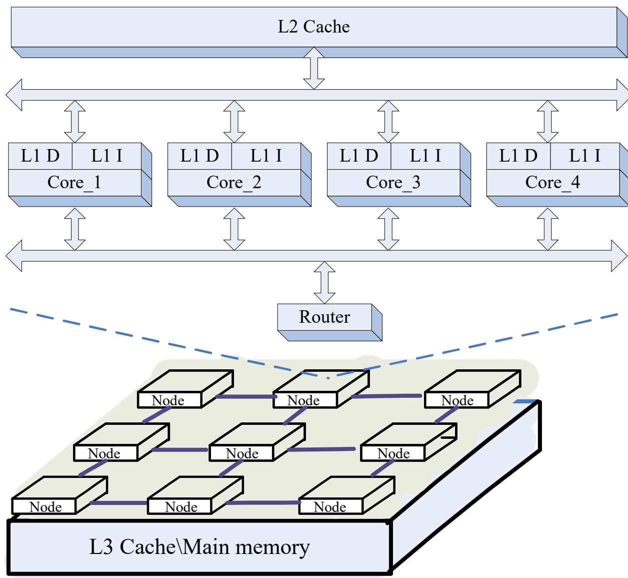

Fig. 4. Optimized cache structure

In this study, a partially shared cache architecture is constructed as shown in Figure 4. The basic communication network on the chip remains constant, and every node still contains a router to transmit data between its neighbour nodes and the PE. The difference is that each node no longer contains only one processing core but a cluster, which is composed of four cores connected with each other by a local on-chip bus. Each core has its own private L1 cache, while its L2 cache is shared by all four cores together. Compared with the bus, the transmission rate of the NoC is extremely fast, thus, the structure with four cores concentrated in one node does not have a significant impact on the overall system transmission. The optimized architecture combines the advantages of private and shared caches. The usable L2 cache of each core becomes larger than before, nevertheless, the overall system resources occupation does not increase.

All the cache structures and replacement algorithms aim to reduce the cache miss rate, the cost of misses includes bandwidth and power consumption and miss penalty. The cache miss rate is usually used as the cache performance evaluation metric [30]. For a many-core system based on NoC, assume that the cache miss rate is H, the time to access data on the cache is T_1, the Manhattan distance between the core and the cache block to be accessed is L, the average data transmission time is T_2, and the time to access data in off-chip memory is T_3, the access time for system T is shown in Formula 5.

$$T = (1 - H) \times T_1 + H \times (L \times T_2 + T_3) \tag{5}$$

Usually, $T_2 \gg T_1$ and $T_3 \gg T_1$; therefore, a higher cache miss rate means longer access time T.

The partially shared cache structure increases the available cache space of every core. The cache size is one of most effect parameters in memory design. By increasing the cache size, the probability of better cache hit rate is increased. One more thing, bigger cache size means more memory area occupation and more static power dissipation of cache structure. The optimised architecture combines the advantages of private and shared caches. For each core, its usable L2 cache becomes larger than before, but the overall memory area occupation is reduced. The optimized cache structure also reduces the number of read-write and inter-chip transmissions because data are shared among different cores, which also helps to reduce system power consumption and improve system scalability.

3.3 CA-RRIP Replacement Algorithm

The choice of a cache replacement algorithm in hybrid associative cache subsystem has a considerable and direct effect on the overall system performance. Based on the partially shared cache structure, the CA-RRIP replacement algorithm is proposed. The algorithm quantifies the reuse of a cache block according to the re-reference interval. It sets a counter (RRIV) for every cache block that represents the reuse prediction of the cache block. As the value of the counter increases, the correlation interval becomes larger, and the probability that the cache block will be reused decreases. In general, the counter is set to N bits. Therefore, its maximum value is $2^N - 1$. Here $2^N - 1$ is called the distant interval value(DIV), while $2^N - 2$ is defined as the long interval value(LIV). Different from the RRIP algorithm, the proposed algorithm records the ID of the core that initiated access and searches for blocks based on the core's ID when the data in the cache need to be replaced. The algorithm performs dynamic virtual partitioning on the partially shared cache, which mitigates the interactions among cores using different access patterns. When a cache area needs to be replaced, the core that initiated the cache access request will be given top priority; this scheme guarantees cache exclusivity when it is used.

All cache replacement algorithms can be divided along three management strategies: a replacement policy, an insertion policy and a promotional policy. The replacement policy determines which cache block to pop up when a new cache block is inserted. The insertion policy is used to determine the state in which the cache block is inserted into the cache. The promotional policy modifies the cache state when a cache line is hit. We use these three strategies to illustrate our cache replacement algorithm.

Insertion policy: When inserting the cache block, the value of RRIV is set to $2^N - 2$(LIV).

Promotion policy: When the cache is hit, the RRIV value of the cache block is decreased by one.

Replacement policy: First, scan the cache blocks occupied by the current core and replace a cache block if its RRIV is $2^N - 1$(DIV). Otherwise, add one to the RRIV values of the cache blocks occupied by the core. Then, determine whether a cache block whose RRIV is $2^N - 1$ exist. When such a block exists, it is replaced. If all the conditions above fail, the RRIV values occupied by all the cores is increased by one. Then, check again whether any other cache blocks whose RRIV is $2^N - 1$ exist. This process will be carried out repeatedly until the replacement accomplished. In any case, it is necessary to preferentially detect the RRIV value of the cache occupied by the current core. If multiple cache blocks have the DIV value, the RRIV is selected according to the

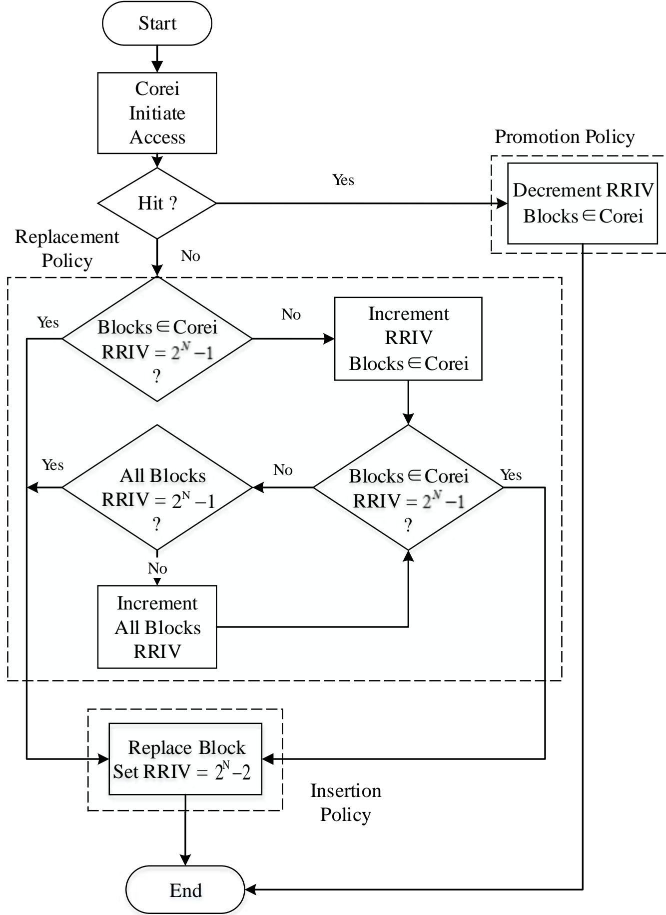

Fig. 5. Flowchart of CA-RRIP

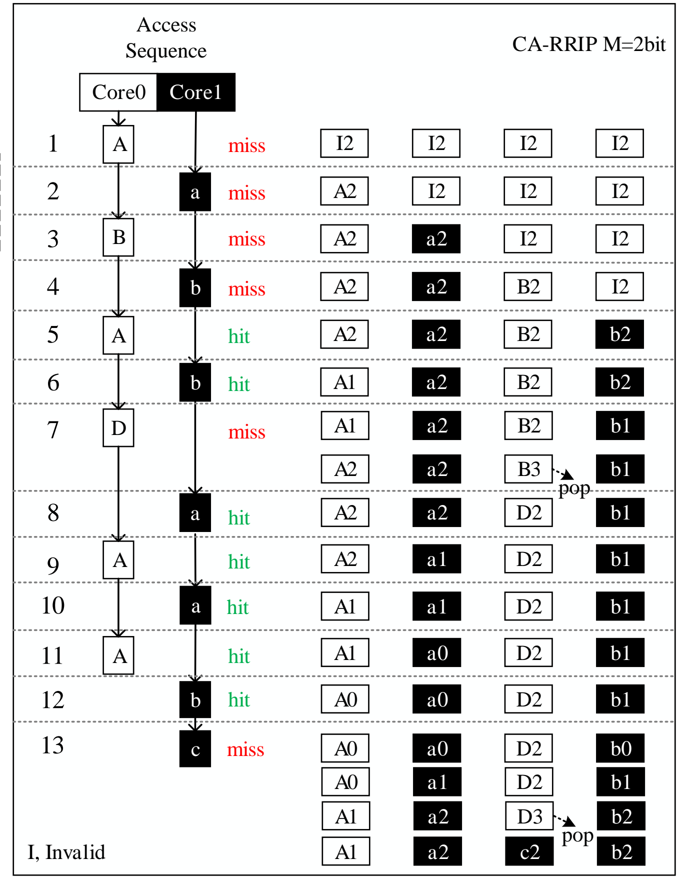

Fig. 6. Behavior example of CA-RRIP

index of the cache block. The algorithm flow is shown in Figure 5.

An example of the CA-RRIP replacement algorithm is shown in Figure 6. Assume that there are two cores sharing the cache, the ID value is set to 2 bits, and the RRIV value is also set to 2 bits. Initially, the shared cache is not partitioned, and each core's ID is recorded when the core initiates cache access. When a cache block is inserted, the insertion policy sets the value of the inserted cache block to 2. The promotional policy is shown in lines 5 and 6 of Figure 6. When the cache is missed, memory access is initiated by core 0, which first checks whether the RRIV of the cache block occupied by the core is 3 (RRIVA = 1, RRIVB = 2, and none = 3). The cache access algorithm increases the RRIV of the cache block occupied by the core so that RRIVA = 2 and RRIVB = 3. Then, cache block B is replaced by D. Similarly, for line 13, the RRIV values of a and b are increased by one firstly because they do not meet the replacement conditions. The RRIV values for all the caches of other cores are increased then, such that RRIVa= 2 and RRIVb = 2, which do not meet the conditions. At this time, when RRIVD = 3, cache block D is replaced by D, and core 1 occupies the cache space of core 0. The cache structure and replacement algorithm relieve the interaction among cores and make full use of the cache space. This approach achieves a better application performance.

TABLE 1
System parameters

Operation system	Ubuntu 12.04 LTS-64bit
Memory size	2 GB
Processor model	Intel Xeon E5620 quad-cores

4 Implementation and Comparison

In this section, the sniper multi-core simulator tool [31] [32] is used to evaluate the efficiency of our proposed design based on simulation of real world benchmark suits.

Based on its interval core model and graphite simulation infrastructure, the Sniper simulator allows a range of flexible simulation options when exploring different homogeneous and heterogeneous many-core architectures [33]. The simulator needs to modify the system architecture parameters only when setting up a comparison scenario. We run the Sniper on the VMware virtual machine with the system parameters set as shown in Table 1.

4.1 Cache Structure Comparison

Instructions per cycle(IPC) is used to evaluate the performance of a single-core system. For a many-core system, the IPC_{sum} is the sum of each core's IPC. As shown in Formula 6, N indicates the number of cores, and i is the index of the ith core. The system throughput(STP) is used to measure the system performance which is calculated in Formula 7, where IPC_i^{MP} indicates the IPC of the ith

TABLE 2
System's configuration

	System Parameter
Processor	4Cores, 2.66GHZ, Nehalem
Private L1 Inst.	Private L1 Inst. 32KB, 64B block-size, 4-way, LRU
Private L1 Data	32KB, 64B block-size, 8-way, LRU
DRAM Controller Number	8(each controller 8Gb/s)
DRAM Bandwidth	64GB/s, 100ns access latency

program, and IPC_i^{SP} indicates the corresponding IPC of a single-core system. The performance improvement of the entire optimized system is calculated by Formula 8, where $IPCi_{cluster}$ is the IPC of the ith program based on the partially shared cache, and $IPCi_{private}$ is the same program's IPC based on the private cache.

$$IPC_{sum} = \sum_{i=1}^{N} IPC_i \tag{6}$$

$$STP = \sum_{i=1}^{N} \frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{7}$$

$$ImproRate = \sum_{i=1}^{N} \frac{IPCi_{cluster} - IPCi_{private}}{IPCi_{private}} \tag{8}$$

We build three separate systems that use the proposed partially shared L2 cache, the private L2 cache, and the row-shared L2 cache respectively. The size of the partially shared L2 cache is 1 MB, the size of the private L2 cache is 256 KB, and the size of the row-shared L2 cache is 2MB. All the systems contain 64 cores, and all of which are based on Intel's Nehalem architecture. The DRAM bandwidth is set to 64 Gb/s, and the access latency is 100 ns. The specific parameters of the system's configuration are shown in Table 2.

Then the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark is used to compare the performance of the systems based on different cache structures [34]. The PARSEC benchmark suite is composed of multithreaded programs. The suite focuses on emerging workloads and is designed to be representative of next-generation programs for chip-multiprocessors. PARSEC is integrated into the Sniper simulation tool, and it is easy to select the granularity, working set size, data sharing and exchange characteristics within the program. It includes test programs from many fields, including blackscholes, canneal, dedup, facesim, ferret, fluidanimate, raytrace, swaptions, vips, x264, and others. These applications are the most well established and common benchmarks which are considered in the current works on NoC domain for general purpose applications . As shown in Figure 7, the different test programs have their own features in the application domain, such as parallelization, working set and data usage.

In general, because of the multiple forwarding via nodes and the sharing competition, the performance of the row-shared cache structure is the worst. Compared with the

Algorithm 1 GetReplacementIndex(core_id_t, m_core_id, CacheCntlr, *cntlr)

1: **for** $i \leftarrow$ 0 to m_associativity-1 **do**
2: **if** m_cache_block_info_array[i] **then**
3: m_rrip_bits[i] $\leftarrow m_rrip_insert$
4: m_core_bits[i] $\leftarrow m_core_id$
5: $return\ i$
6: **end if**
7: **end for**
8: **for** $i \leftarrow$ 0 to m_rrip_max **do**
9: **for** $i \leftarrow$ 0 to m_associativity-1 **do**
10: **if** m_core_id == m_core_bits[i] && m_rrip_bits[i] $\geq$ m_rrip_max **then**
11: m_rrip_bits[i] $\leftarrow m_rrip_insert$
12: $return\ i$
13: **else**
14: m_rrip_bits[i] $++$
15: **end if**
16: **end for**
17: **end for**
18: **for** $i \leftarrow$ 0 to m_associativity-1 **do**
19: **if** m_core_id == m_core_bits[i] && m_rrip_bits[i] $\geq$ m_rrip_max **then**
20: m_rrip_bits[i] $\leftarrow m_rrip_insert$
21: $return\ i$
22: **end if**
23: **for** $i \leftarrow$ 0 to m_associativity-1 **do**
24: **if** m_rrip_bits[i] $\geq$ rrip_max **then**
25: m_rrip_bits[i] $\leftarrow m_rrip_max$
26: $return\ i$
27: **if** m_rrip_bits[i] $<$ m_rrip_insert **then**
28: m_rrip_bits[i] $\leftarrow m_rrip_max$
29: m_rrip_bits[i] $++$
30: **end if**
31: **end if**
32: **end for**
33: **end for**

private cache structure, the overall system resource occupation of the proposed partially shared cache system can be reduced by 20% with the same number of cores. The IPC of different test programs based on two cache structures are shown in Figure 8. While there are huge differences for different test programs due to their disparate features, the IPC of the system with the partially shared L2 cache is generally higher than any other system, especially for the canneal test program, where its IPC increased by up to 49.23%. The STP of the partially shared cache structure increased by 5.89% on average. The parallel processing capability is also enhanced.

4.2 Cache Replacement Algorithm Comparison

The Sniper simulation tool is also applied to validate the cache replacement algorithm. The related cache classes are shown in Figure 9. The classes include the methods for reading and writing the cache, and each method returns a result to show whether the cache was hit or not. The cache manager uses this return value to update the simulation performance parameters such as the number of hits, the

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Fig. 7. Test programs of PARESEC

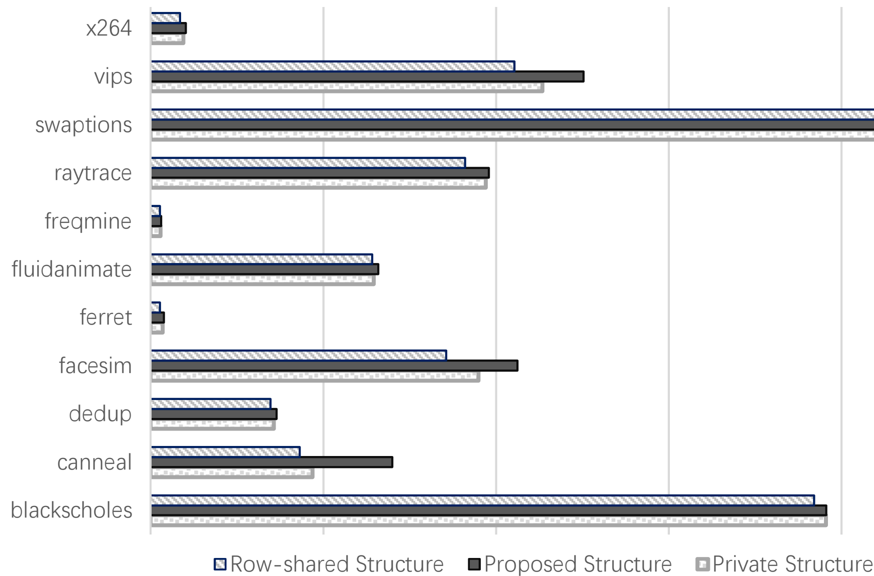

Fig. 8. Comparison of the IPC

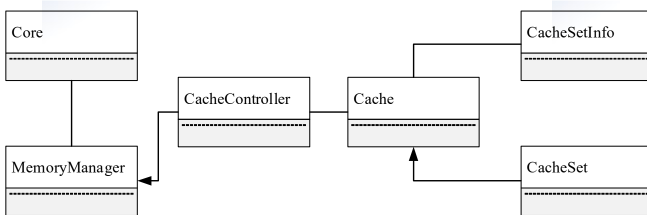

Fig. 9. The cache class

simulation time, and so on. The CacheSet class creates the cache and modifies the cache management algorithm. All the cache blocks are allowed to use the CacheSetInfo instance. CacheSetInfo stores information about the cache group, including the cache status cache size, and so on.

In the algorithm implementation, the cache access function needs to be rewritten in the Core model file, and the functions of finding, inserting, and replacing information in the cache replacement algorithms must be rewritten as well. We added the m_core_id field to the function that initiates the cache application; this field is recorded as a function of the cache management algorithm. The CacheSet CA-RRIP inherits from the CacheSet function. When initializing the cache, it is necessary to record the data structure of the core ID and RRIV. The pseudocode for the key algorithm is shown in Algorithm 1.

Based on the partially shared cache system structures, we continue testing the cache replacement algorithm. The system configuration is the same as that shown in Table 2. The configuration parameters for the L1 cache are shown in Table 3. According to Intel's Nehalem architecture, the L1 data cache and instruction cache are both 32 KB, the data associativity is 8, and the instruction associativity is 4. Assuming that the system clock frequency is set as 1 GHz, the access latency is 3 ns.

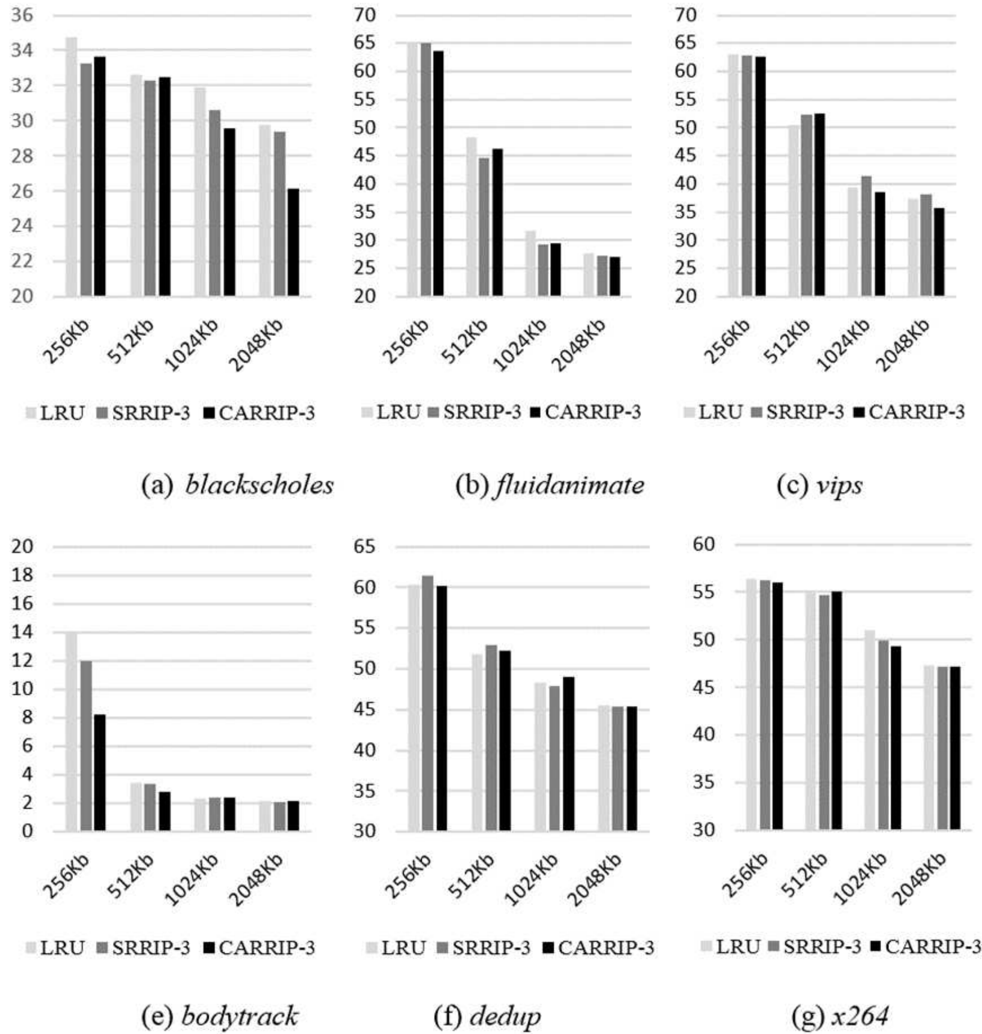

Fig. 10. The comparison results of cache miss rate

TABLE 3
The configuration of L1 cache

Cache	Data	Instruction
Cache Size	32KB	32KB
BlockSize	64B	64B
Assiociativity	8	4
AccessCycles(Tag)	1	1
AccessCycles(Data)	3	3
AccessTime(Tag)	1ns	1ns
AccessTime(Data)	3ns	3ns

TABLE 4
The configuration of L2 cache

L2 Cache Size	256KB	512KB	1024KB	2048KB
BlockSize	64B	64B	64B	64B
Assiociativity	16	16	16	16
AccessCycles(Tag)	3ns	3ns	3ns	3ns
AccessCycles(Data)	9ns	9ns	9ns	9ns

The L2 cache sizes are set as shown in Table 4. We compare the cache replacement algorithms with four different L2 cache sizes: 256 KB, 512 KB, 1,024 KB and 2,048 KB. Two types of test programs with various cache access patterns are used as comparisons: one type consists of programs with low data-sharing frequency among cores that include the blackscholes, fluidanimate, and vips programs. The other type consists of programs with high data-sharing frequency that include bodytrack, dedup, and x264. We implement three different cache replacement algorithms, LRU, SRRIP and CA-RRIP, and compare the cache miss rate among them. The results are shown in Figure 10.

As the experimental results show, for all the programs, the cache miss rate diminishes as the cache size increases, which is easy to understand because a larger cache size means more data buffering is available. For the programs with low data-sharing-frequency among cores, relatively independent cache spaces are required for every core, and the interactions among different cache access patterns should also be reduced. The CA-RRIP algorithm achieves better

performance in this case due to its pseudo-partitioning for the shared cache and because it grants the core that initiated the cache access request top priority, which increases the cache hit rate as well. In the best case, the CA-RRIP algorithm reduces the cache miss rate by 31% compared with the LRU algorithm, and it is 13% lower than that of the RRIP algorithm. For the programs with high data-sharing frequency, all the cores attempt to read/write the cache simultaneously. Thus, the advantage of the CA-RRIP algorithm is not obvious because the L2 cache is shared by four cores, which affects the average read-write speed of any individual core.

Generally, the CA-RRIP algorithm reduces the interactions among many-core architectures by increasing the number of bits in the RRIP algorithm to record the ID of every core and implement pseudo-partitioning for the shared cache. These operations not only have small system resource consumption but also reduce the cache miss rate and improve the system's performance effectively . One more thing, it is easy to implement and is more capable.

5 SUMMARY

The advanced semiconductor and System-on-Chip technologies allow dozens of or even more cores to be integrated into a single die. In particular, the NoC architecture separates task-computing from the communication structure and has become the inevitable choice for the next generation of sophisticated system architectures. However, the typical cache structure and replacement algorithms have difficulty meeting the requirements of the increasing number of cores of many-core systems based on NoC due to their high resource consumption and long access latencies.

This paper describes the construction of a partially shared cache structure and proposes a CA-RRIP cache replacement algorithm. Through the hybrid interconnect technology of bus-on-chip and NoC, this optimized cache structure reduces the overall system resource occupation and improves the data-sharing efficiency in the L2 cache among cores, increasing system throughput by 5.89% on average. The CA-RRIP cache replacement algorithm reduces the cache miss rate of programs with various cache access patterns and improves the overall performance of many-core systems. Based on these research results, applying the optimized cache structure and replacement algorithm to practical systems is part of our ongoing work.

REFERENCES

[1] H. Temuçin and K. M. İmre, "Scheduling computation and communication on a software-defined photonic network-on-chip architecture for high-performance real-time systems," *Journal of Systems Architecture*, vol. 90, pp. 54–71, 2018.

[2] M. O. Agyeman, A. Ahmadinia, and N. Bagherzadeh, "Energy and performance-aware application mapping for inhomogeneous 3d networks-on-chip," *Journal of Systems Architecture*, vol. 89, pp. 103–117, 2018.

[3] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, pp. 402–412, ACM, 2014.

[4] S. M. Shahtouri and R. T. Ma, "App: adaptively protective policy against cache thrashing and pollution," in *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, pp. 1–6, IEEE, 2015.

[5] C. Lin and J.-N. Chiou, "High-endurance hybrid cache design in cmp architecture with cache partitioning and access-aware policies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 10, pp. 2149–2161, 2015.

[6] Z. Xiaoyu and W. Junmin, "Collaborative directory-based cache design for cmp," *Computer Engineering*, vol. 32, no. 21, 2010.

[7] H. Feng and W. Chengyong, "Design and performance analysis of cmp architecture without caching," *Computer Engineering*, vol. 29, no. 7, 2008.

[8] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 336–345, IEEE, 2005.

[9] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel, "Approximation-aware multi-level cells stt-ram cache architecture," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on*, pp. 79–88, IEEE, 2015.

[10] A. Bengueddach, B. Senouci, S. Niar, and B. Beldjilali, "Energy consumption in reconfigurable mpsoc architecture: Two-level caches optimization oriented approach," in *Design and Test Symposium (IDT), 2013 8th International*, pp. 1–6, IEEE, 2013.

[11] B. Cilku, D. Prokesch, and P. Puschner, "A time-predictable instruction-cache architecture that uses prefetching and cache locking," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pp. 74–79, IEEE, 2015.

[12] N. Naderializadeh, M. A. Maddah-Ali, and A. S. Avestimehr, "Fundamental limits of cache-aided interference management," *IEEE Transactions on Information Theory*, vol. 63, no. 5, pp. 3092–3107, 2017.

[13] A. Jadidi, M. Arjomand, M. T. Kandemir, and C. R. Das, "Hybridcomp: A criticality-aware compressed last-level cache," in *Quality Electronic Design (ISQED), 2018 19th International Symposium on*, pp. 25–30, IEEE, 2018.

[14] J. Merino, V. Puente, P. Prieto, and J. Á. Gregorio, "Sp-nuca: a cost effective dynamic non-uniform cache architecture," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 2, pp. 64–71, 2008.

[15] Z. Ming, M. Xu, and D. Wang, "Age-based cooperative caching in information-centric networking," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pp. 1–8, IEEE, 2014.

[16] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[17] M. Kharbutli and R. Sheikh, "Lacs: A locality-aware cost-sensitive cache replacement algorithm," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1975–1987, 2014.

[18] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, ACM, 2007.

[19] S. Kumar and P. Singh, "An overview of modern cache memory and performance analysis of replacement policies," in *Engineering and Technology (ICETECH), 2016 IEEE International Conference on*, pp. 210–214, IEEE, 2016.

[20] J. Alghazo, A. Akaaboune, and N. Botros, "Sf-lru cache replacement algorithm," in *Memory Technology, Design and Testing, 2004. Records of the 2004 International Workshop on*, pp. 19–24, IEEE, 2004.

[21] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 208–219, ACM, 2008.

[22] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

[23] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 423–432, IEEE, 2006.

[24] B. M. Beckmann, M. R. Marty, and D. A. Wood, "Asr: Adaptive selective replication for cmp caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 443–454, IEEE, 2006.

[25] D. Zhan, H. Jiang, and S. C. Seth, “Clu: Co-optimizing locality and utility in thread-aware capacity management for shared last level caches,” *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1656–1667, 2014.
[26] P. Yang, Q. Wang, W. Li, Z. Yu, and H. Ye, “A fault tolerance noc topology and adaptive routing algorithm,” in *Embedded Software and Systems (ICESS), 2016 13th International Conference on*, pp. 42–47, IEEE, 2016.
[27] P. Yang and Q. Wang, “Heterogeneous honeycomb-like noc topology and routing based on communication division,” *International Journal of Future Generation Communication and Networking*, vol. 8, no. 1, pp. 19–26, 2015.
[28] P.-F. Yang and Q. Wang, “Effective task scheduling and ip mapping algorithm for heterogeneous noc-based mpsoc,” *Mathematical Problems in Engineering*, vol. 2014, 2014.
[29] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, “High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 343–353, IEEE, 2015.
[30] S. Wu, B. Mao, Y. Lin, and H. Jiang, “Improving performance for flash-based storage systems through gc-aware cache management,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2852–2865, 2017.
[31] T. E. Carlson, W. Heirmant, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12, IEEE, 2011.
[32] W. Heirman, T. Carlson, and L. Eeckhout, “Sniper: Scalable and accurate parallel multi-core simulation,” in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pp. 91–94, High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.
[33] T. Carlson and W. Heirman, “The sniper user manual,” 2013.
[34] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.