A Dynamic Link-latency Aware Cache Replacement Policy (DLRP)

Yen-Hao Chen, Allen C.-H. Wu, TingTing Hwang
yhchen@cs.nthu.edu.tw,allenwuuw@gmail.com,tingting@cs.nthu.edu.tw
Department of Computer Science, National Tsing Hua University, Taiwan

ABSTRACT

Multiprocessor system-on-chips (MPSoCs) in modern devices have mostly adopted the non-uniform cache architecture (NUCA) [1], which features varied physical distance from cores to data locations and, as a result, varied access latency. In the past, researchers focused on minimizing the average access latency of the NUCA. We found that dynamic latency is also a critical index of the performance. A cache access pattern with long dynamic latency will result in a significant cache performance degradation without considering dynamic latency. We have also observed that a set of commonly used neural network application kernels, including the neural network fully-connected and convolutional layers, contains substantial accessing patterns with long dynamic latency. This paper proposes a hardware-friendly dynamic latency identification mechanism to detect such patterns and a dynamic link-latency aware replacement policy (DLRP) to improve cache performance based on the NUCA.

The proposed DLRP, on average, outperforms the least recently used (LRU) policy by 53% with little hardware overhead. Moreover, on average, our method achieves 45% and 24% more performance improvement than the not recently used (NRU) policy and the static re-reference interval prediction (SRRIP) policy normalized to LRU.

ACM Reference Format:
Yen-Hao Chen, Allen C.-H. Wu, TingTing Hwang. 2021. A Dynamic Link-latency Aware Cache Replacement Policy (DLRP). In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431420

1 INTRODUCTION

Nowadays, many processor devices, including servers, personal computers, and embedded systems, have used the multiprocessor system-on-chips (MPSoCs). In MPSoCs, since the working set size has grown larger and larger, a shared cache has been adopted to utilize the cache capacity fully. A shared cache architecture partitions a large cache into several banks to avoid fixed worst-case access latency, i.e., non-uniform cache architecture (NUCA) [1]. Figure 1 shows a 4×4 NUCA system with mesh topology in which each circle represents a network-on-chip (NoC) router connecting the network to a core, private cache, and shared cache bank. Take $core_3$ in Figure 1 as an example; it has a short physical distance to cache $bank_3$, which means that it has a short link latency to access data in cache $bank_3$, i.e., 0 link-latency. On the other hand, $core_3$ has a long physical distance to cache $bank_{12}$, which means that it has an extended link latency to access data in cache $bank_{12}$, i.e., six link-latency. The physical distance between core and data location will be a parameter of the hit rate in a NUCA cache system.

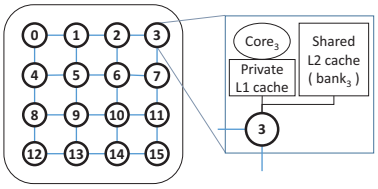

Figure 1: A 4×4 non-uniform cache architecture (NUCA) system.

C. Kim et al. first proposed the concept of NUCA [1]. Several chip implications on NUCA also has been developed by both academic communities and industrial companies, e.g., L-NUCA [2] and IBM Power9. C. Kim et al. categorized the NUCA into static NUCA (S-NUCA) and dynamic NUCA (D-NUCA) depending on if the data mapping scheme to the cache banks is fixed or not [1]. The S-NUCA has a fixed mapping scheme of the address space. The goal of mapping is to have a short average access latency overall. The simplicity of S-NUCA makes it easy for detailed analyses [3] and, thus, commercial multiprocessors often adopt the S-NUCA [4]. The D-NUCA avoids long link distances by adopting complex data migration, partition, or replication schemes [4–6]. Those D-NUCA techniques can reduce the average cache access latency but come with the price of high design complexity. In this paper, we will focus on S-NUCA.

Many modern processors, such as Intel i5-750, utilize a large cache to avoid misses. On the other hand, some embedded systems adopt a smaller cache size with dedicated caching policies to avoid cache misses [7]. In this paper, we will focus on the latter cache architecture and study the cache replacement policy for its shared L2 cache.

Many cache replacement policies have been proposed to improve cache performance. Most of them are heuristic-driven policies, e.g., least recently used (LRU) policy and static re-reference interval prediction (SRRIP) policy [8], developed by observations of program execution behavior. Some schemes even combine multiple policies to achieve better performances, e.g., dynamic re-reference interval prediction (DRRIP) policy [8]. More recently, learning-based policies have been proposed [9], which can achieve higher hit rates but come with more hardware area. In this paper, we will focus on heuristic policies.

In the past, researchers focused on minimizing the average access latency of the NUCA. We found that dynamic latency has a more critical impact on the interference than the overall lumped average

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431420

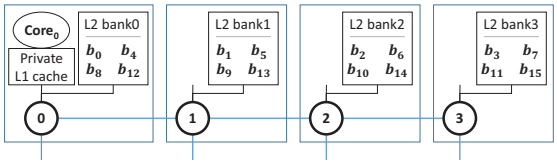

Figure 2: Allocation of 16 data blocks $b_1, b_2, ..., b_{15}$.

latency in multi-application environments. A cache access pattern with a large variety of dynamic latency will result in significant cache performance degradation without considering the dynamic latency. We have also observed that a set of commonly used neural network application kernels, including the neural network fully-connected and convolutional layers, contains substantial accessing patterns with varying dynamic latency. In this paper, we propose a hardware-friendly dynamic latency identification mechanism to detect such patterns and a dynamic link-latency aware replacement policy (DLRP) to improve cache performance based on the S-NUCA.

The paper is organized as follows. Section 2 gives motivation. Section 3 discusses the dynamic latency. Section 4 and Section 5 presents our proposed method and experimental results. The final section gives the concluding remarks.

2 MOTIVATION

The non-uniform cache architecture (NUCA) has variable access latency related to the physical distance of the core and data location [1]. A data block with long access latency may have interference from other cores. Interference will cause the block being evicted by other concurrently running applications before accessed. Ideally, allocating all data near the processor could avoid this problem. However, it is impossible to do so due to cache capacity. In the past, most researchers focused on reducing the average network access latency for the NUCA [1]. We found that the dynamic latency has a more critical impact on the interference than the overall lumped average latency in multi-application environments. To demonstrate the observation, we use an example on a NUCA architecture with two access patterns. We found that even with the same average access latency, the cache hit rate may be different under multi-application environments.

Assume that $core_0$ issues cache requests for 16 data blocks $b_0, b_1, ..., b_{15}$. Figure 2 shows location of the $core_0$ and data blocks. In the figure, data blocks b_0, b_4, b_8, and b_{12} are in bank 0 with link latency 0, data blocks b_1, b_5, b_9, and b_{13} in bank 1 with link latency 1, data blocks b_2, b_6, b_{10}, and b_{14} in bank 2 with link latency 2, data blocks b_3, b_7, b_{11}, and b_{15} in bank 3 with link latency 3. The first pattern, $pattern1$, is

$$(b_0\ b_1\ b_2\ b_3)^{k_0}(b_4\ b_5\ b_6\ b_7)^{k_1}(b_8\ b_9\ b_{10}\ b_{11})^{k_2}(b_{12}\ b_{13}\ b_{14}\ b_{15})^{k_3},$$

where k_0 means the sub-pattern $(b_0\ b_1\ b_2\ b_3)$ is repeatedly accessed k_0 times in $period0$, and, similarly, k_1, k_2, and k_3 mean sub-patterns $(b_4\ b_5\ b_6\ b_7)$, $(b_8\ b_9\ b_{10}\ b_{11})$, and $(b_{12}\ b_{13}\ b_{14}\ b_{15})$ are repeatedly accessed k_1, k_2, and k_3 times in $period1$, $period2$, and $period3$, respectively. It is a bank-interleaved accessing pattern. The second accessing pattern, $pattern2$, is with the same 16 data blocks but a different order. The accessing pattern is

$$(b_0\ b_4\ b_8\ b_{12})^{k_0}(b_1\ b_5\ b_9\ b_{13})^{k_1}(b_2\ b_6\ b_{10}\ b_{14})^{k_2}(b_3\ b_7\ b_{11}\ b_{15})^{k_3},$$

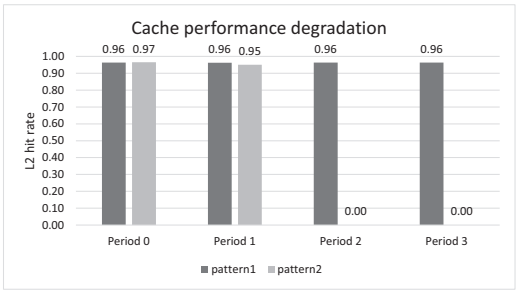

Figure 3: Distributed L2 cache performance degradation on accessing patterns of $pattern1$ and $pattern2$ under a multi-application environment.

where sub-patterns $(b_0\ b_4\ b_8\ b_{12})$, $(b_1\ b_5\ b_9\ b_{13})$, $(b_2\ b_6\ b_{10}\ b_{14})$, and $(b_3\ b_7\ b_{11}\ b_{15})$ are repeatedly accessed k_0, k_1, k_2, and k_3 times in $period0$, $period1$, $period2$, and $period3$, respectively. In this pattern, blocks in the same bank are accessed before moving to the next bank.

We execute those two patterns with regular streaming applications as a multi-application environment. (The detail experimental settings will be given in Section 5.) Let $k_0 = k_1 = k_2 = k_3$ for both patterns. We collect and analyze L2 cache accesses only from $pattern1$ and $pattern2$ to understand the individual cache performance. From our experiment, those two patterns have the same average access latency, i.e., $\frac{1}{4}(0+1+2+3) = 1.5$. However, $pattern1$ has a 96.30% hit rate while $pattern2$ has only a 55.84% hit rate. Thus, even though two patterns have the same average latency, their cache hit rates are very different.

The difference in cache hit is explained as follows. Because all 16 data blocks are already in different cache banks and sets, the interference is the only possible reason to cause the hit rate difference. $pattern1$ sequentially accesses each bank repeatedly. The average link latency in all periods is the same. Consequently, the interferences to all periods are similar and result in similar cache hit rate in all periods. Figure 3 shows the cache hit rate of $pattern1$ in different periods, in which $pattern1$ (dark bars) shows a similar cache hit rate regardless of the periods. On the other hand, $pattern2$ (light bars) keep access to the same bank in a period. The average link latency of different periods is different. In some periods of $pattern2$, all accesses travel in a lengthy distance, e.g., $period3$. In those periods, the average link latency of the period is longer than the others. Consequently, those periods, e.g., $period3$, will suffer more interferences and have degraded cache performance. Figure 3 shows that cache accesses of $pattern2$ in $periods0$ and $periods1$ have over 95% hit rate, while those in $period2$ and $period3$ have a 0% hit rate. Notice that $pattern1$ and $pattern2$ have the same overall average latency. However, in terms of latency of periods, $pattern1$ has the same latency in all periods, while $pattern2$ has very long latency in some periods. For those periods, more interference from other core occurs. Thus, the cache hit rate of $pattern2$ degrades a lot under the multi-application environment.

Recently, the artificial neural network (ANN) has shown excellent results in various areas, and many real-world systems adopt the ANN technique. We found that the commonly used kernels of ANN, including 2D convolution and matrix multiplication [10], features long dynamic latency, which will be discussed in detail later.

3 DYNAMIC LATENCY

As shown in the previous section, patterns with long dynamic latency have hit rate degradation. This section will give a formal definition of dynamic latency and a hardware identification mechanism for implementation.

3.1 Long Dynamic Latency

The overall access latency (or static latency) is the average latency of all accesses. The latency in a period is defined as dynamic latency and formally defined as follows. Consider a sequence of cache access, i.e., an access pattern. Consider a constant length of time t. Let period P_i starts from time t_{i-1} to t_i of length t, where $i \in N^+$. Also, let n_i be the number of accesses in the period P_i, and accesses, a_j, with latency $l_j^{P_i}$ for j from 1 to n_i. Assuming $n_i > 0, \forall i$, the dynamic latency of period P_i is $dl_i = \frac{1}{n_i} \sum_{j=1}^{n_i} l_j^{P_i}$. If the time length t covers the whole access pattern, the dynamic latency equals the static average latency. The dynamic latency of the periods forms a sequence of averages, i.e., $dl_1, dl_2, dl_3, ...$ If all periods have the same dynamic latency, the interference as well as the hit rate will be the same. On the other hand, periods with long dynamic latency may have more interference and lower hit rate than others. Thus, we need dedicated mechanisms to solve the long dynamic latency problem.

The next question is how to define a period P_i with long dynamic latency. We say that a dynamic latency is long if it is larger than a threshold, i.e., $P_i : dl_i > threshold$. "Long or not too long" is a relative concept. Thus, there are many ways to define the threshold depending on the applications. The threshold can be the average latency of all latency, i.e., static latency, or the average latency from a core to all cache banks for a given architecture or any other definition derived from the target architecture. For example, let the threshold be the static latency. The static latency of *pattern*1 and *pattern*2 in Figure 3 are both 1.5. The dynamic latency of all periods in *pattern*1 is stable and equal to the static latency 1.5. On the other hand, the dynamic latency of *period*3 in *pattern*2 is up to 3 and more significant than the static latency. Thus, *period*3 has a long dynamic latency in *pattern*2.

3.2 Identification Mechanism

In the real implementation, it is hard to cut the time for periods during runtime. Instead of calculating the average latency of periods, we propose to use the exponentially weighted moving average (EWMA) to represent the dynamic latency due to hardware efficiency. The exponentially weighted moving average (EWMA) is updated when cache access is issued, as shown below.

$$EWMA_{i+1} = l_{i+1} \times p + EWMA_i \times (1 - p), 0 < p < 1,$$

where l_{i+1} is the latency of current cache access. The parameter p controls the averaging weights of latency. A smaller p gives a higher weight to historical values. A detector should give a higher weight to historical values when more banks (or more possible latencies) are in the cache architecture. In this paper, we set p to the reciprocal of the number of cache banks. For example, consider a 4×4 MPSoC with the mesh topology. The value p is $1/16$.

The next question is to define the threshold. If the threshold is too small (or too close to the average latency), the detector may over-report the long dynamic latency. However, if the threshold is too large, it may miss real long ones. In this paper, we will set the threshold using the target architecture. Let L be the average latency from a core to all cache banks and D the maximum latency. We define the threshold as the mean of the average latency L and the maximum latency D derived from our experiment, i.e., $threshold = \frac{L+D}{2}$. The reason behind this heuristic is that it maximizes the margin to both ends. Thus, a dynamic latency, $EWMA_i$, is a long dynamic latency if it is larger than the threshold, i.e., $EWMA_i > \frac{L+D}{2}$. For example, consider a 4×4 MPSoC with the mesh topology. The average latency, L, is three link-latency, and the maximum latency, D, is six link-latency. Thus, the period has long dynamic latency if there exists EWMA larger than 4.5 link-latency.

A period with long dynamic latency suffers more interferences than others. So, accesses in the period with long dynamic latency may have a lower hit rate. A dedicated mechanism is needed to solve this long dynamic latency problem.

4 CACHE REPLACEMENT POLICY

In this section, we will introduce a dynamic link-latency aware replacement policy (DLRP) considering the dynamic latency based on the widely used Static Re-Reference Interval Prediction (SRRIP) replacement policy [8].

4.1 Review of the Replacement Policies

A. Jaleel et al. proposed the static re-reference interval prediction (SRRIP) replacement policy [8]. The SRRIP intelligently tries to avoid the scanning and thrashing access patterns by separating the accesses into two categories, namely accesses with the *near-immediate re-reference interval* and the *long re-reference interval*. The SRRIP assigns them to different priority correspondingly. To do so, it uses multiple replacement policy bits (RPB) to indicate each block's priority. Also, the RPB value will be updated when the cache is in use. Consider an SRRIP policy with m-bit RPB, i.e., an m-bit SRRIP policy. On a cache hit, the SRRIP will predict this cache block to have a *near-immediate re-reference interval* and set it to the highest priority, i.e., 0. On a cache miss, a cache block is fetched and stored. The SRRIP will predict this cache block to have a *long re-reference interval* and set the cache block to a low priority, i.e., $2^m - 2$. On eviction, the SRRIP will search for a cache block with the lowest RPB value, i.e., $2^m - 1$, as the victim. If there is no cache block with the lowest RPB value, the SRRIP will update RPBs of all cache blocks by adding one and then search for a victim again. The SRRIP policy repeats the above steps until it finds a victim cache block. Take $m = 3$ (3-bit SRRIP policy) as an example. There are eight priority levels $\in \{0...7\}$. Priority 0 denotes the highest priority indicating the most recently used blocks, and priority 7 denotes the lowest priority indicating the least recently used block. On a cache hit, SRRIP will set the cache block to priority 0. On the other hand, on a cache miss, SRRIP will set the cache block to priority 6. On an eviction, SRRIP always searches for the cache block with priority 7. If there is no cache block with priority 7, SRRIP will decrease the priorities of all blocks by adding one and then search for a victim again. The SRRIP does not consider the latency, which may result in performance degradation.

Table 1: Replacement policy bit (RPB) priority.

	NRU (1-bit)	SRRIP (m-bit)	DLRP (m-bit)
Hit	0	0	0
Miss	0	$2^m - 2$	$2^m - 2 - RRI\_lat$

4.2 Link-latency Aware Replacement Policy

A new link-latency aware replacement policy is invoked when a long dynamic latency is detected using the method proposed in Section 3.2. The new link-latency aware replacement policy based on SRRIP is designed as follows.

First, we define that the re-reference interval (RRI) of cache access to a data block b_i is the number of data blocks accessed before re-accessing the data block b_i in the same cache set. Table 1 summarizes the replacement policy bit (RPB) values for a currently accessed data block. The not-recently used (NRU) policy does not differentiate hit or miss access and gives the highest priority to all accesses. On the other hand, the SRRIP utilizes multiple bits for RPB and gives the missing access a relatively low priority. However, SRRIP policy does not consider latency, which may introduce additional re-reference interval (RRI). In this paper, we propose a link-latency aware replacement policy that considers interferences by giving a higher priority to the miss accesses using the *RRI_lat* parameter.

We define the parameter *RRI_lat* as the difference of RRI that is related to current latency. The following equation captures the *RRI_lat*, where the first term is the interference from other applications, and the second term estimates the portion of delay caused by the link latency.

$$RRI\_lat = inter\_RRI \times \frac{latency \times (inner\_RRI + 1) \times N\_sets}{reaccessing\_cycles} \quad (1)$$

The *latency* denotes the link latency of the current access. The *inter_RRI* is the re-reference intervals that are from other applications accessing the same cache bank, *reaccessing_cycles* the average elapsed cycles between two accesses to the same data block, *inner_RRI* the re-reference interval that is from the same application accessing the same cache bank, and *N_sets* the number of cache sets in a bank. Notice that the sum of *inter_RRI* and *inner_RRI* is the average reference interval, *RRI*, i.e., $inter\_RRI + inner\_RRI = RRI$. The *N_sets* is dependent upon the target memory architecture and, thus, a constant value. Furthermore, the *inter_RRI*, *reaccessing_cycles*, and *inner_RRI* are application dependent. Notice that the number of bits of the *RRI_lat* depends on the number of priority levels (Table 1). For example, a 4-bit *RRI_lat* is enough for a 16-way cache system, and 8-bit multiplication/division operations are enough to estimate the *RRI_lat* value, which is relatively small compared to the cache tag array.

A hardware monitor is designed to collect the application characteristics on-the-fly, including the *inter_RRI*, *reaccessing_cycles*, and *inner_RRI*. An inter_RRI counter records the interval from other applications between two accesses to the target set, and an inner_RRI counter records the interval between two accesses to the target set from the same application. Also, the *reaccessing_cycles* is the cycle time difference between two accesses of the target block.

Figure 4 shows additional hardware in red blocks on a 16-core mesh system, in which each core has a separate monitor in each cache bank. The hardware monitors sample addresses, i.e., the monitoring addresses, representing the accessing pattern of the core.

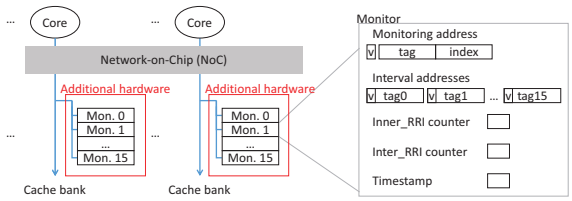

Figure 4: Proposed hardware monitors.

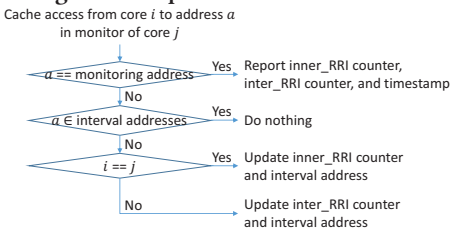

Figure 5: Control flow chart of hardware monitor.

The hardware also records the interval addresses for the monitoring address. The inner_RRI and the inter_RRI counters indicate current counting numbers of *inner_RRI* and *inter_RRI*. Notice that the sum of inner_RRI and inter_RRI counters equals the number of valid interval addresses. Lastly, the timestamp records the first accessing time of the monitoring address.

When a core i issues a cache accessing request, the network-on-chop (NoC) will also pass the request information to each monitor of core j. Figure 5 shows the control flow chart for a cache access from core i to address a in the monitor of core j. If a is equal to the monitoring address, it is a re-access to the monitoring address. Then, the monitor reports the inner_RRI counter, inter_RRI counter, and timestamp as the core characteristics. Otherwise, if the address, a, is in interval addresses, it does nothing. If not, it will check whether i is equal to j. if $i = j$, then it increases the inner_RRI counter by one and stores the address, a, into interval addresses. Otherwise, it increases the inter_RRI counter by one and stores the address, a, into interval addresses. Notice that the distribution of the requests will cause additional network loading.

In our design, all of the monitors are set to invalid initially. Also, the monitor is set to invalid when context switching. It starts monitoring and reporting the characteristic when long dynamic latency is detected. When turning on, the monitor of core i will monitor the first accessing request from the core i and record the current time into the timestamp. Later on, after the second time access to the monitoring address, the inner_RRI and inter_RRI counters are reported as the *inner_RRI* and *inter_RRI* of the core. Also, the *reaccessing_cycles* of the core is the subtraction of the current cycle time and the timestamp. After that, the monitor invalidates the addresses and resets the counters to 0. Then, the monitor will wait for the next request from the core i to monitor.

Figure 6 shows a top-level replacement policy bit (RPB) selection of our proposed dynamic-link-latency-aware replacement policy (DLRP). If the access has no long dynamic latency, then the policy behaves the same as the SRRIP, which sets RPB value to 0 to the hit blocks and $2^m - 2$ to the miss blocks. In this case, the *RRI_lat* is a don't-care signal, and those related hardware monitors are power-gated to save energy. Otherwise, the access has a long dynamic latency. The multiplexer will select the $2^m - 2 - RRI\_lat$ as the RPB value to the miss blocks.

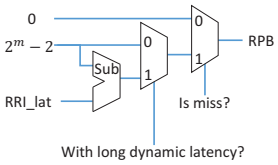

Figure 6: The selection of replacement policy bits (RPB).

4.3 Hardware Overheads

Our proposed replacement policy requires additional monitors to obtain the characteristics of applications. We calculate the number of storage buffers of monitors to estimate the hardware area overhead. Each monitor for a core in a cache bank has a monitoring address, several interval addresses, two positive integer counters, and a timestamp. The monitoring address is a full-sized cache address without offset bits. Also, the interval addresses only need to record the tags to present the addresses. Assume that the number of interval addresses is 16. In this case, 4-bit counters are adequate. We also assume that the timestamp uses a general 64-bit integer, and the system uses an address of 64-bit wide with 6-bit offset, 9-bit indexing, and 49-bit tag. It will be approximately 120 bytes for each monitor, including the valid bits. In our design, each cache bank has a separate monitor for each core. Thus, there are N^2 monitors for an N-core system. Assuming a system consists of 16-core and 320kB cache with 16 distributed cache banks. The overall additional hardware area will be $16 \times 16 \times 120B = 30kB$. By the cache simulator Cacti6.5, it is 4.10% additional overhead to the cache tag hardware area and 0.07% to the whole cache hardware area. Consider a system with 64-core and 2.5mB cache as another example. The simulation result from the Cacti6.5 shows 12.00% additional overhead to the cache tag hardware area and only 0.49% to the whole cache hardware area. Overall, we consider that the hardware area overhead of our design is relatively small.

One may suspect that it may require additional timing overheads due to the calculation of *RRI_lat*. The DLRP calculates the priority value only when there is a miss and performs in parallel with the data fetching from the next level of memory. As a result, the implementation of our proposed policy requires no additional delay overhead.

5 EXPERIMENTAL RESULTS

We use the Gem5 system simulator and the ruby system to evaluate our method on an in-ordered 16-core, 1Ghz system with x86 64 ISA. The system has a 4×4 mesh network-on-chip (NoC) using the MOESI coherence protocol with link-latency eight cycles. Furthermore, processors have private 4kB L1 caches with three cycles and a shared distributed 320kB L2 cache with 15 cycles. Furthermore, the DRAM latency is 500 cycles.

Our method is orthogonal to other heuristic policies and, thus, can be integrated with any other policies, e.g., DRRIP, to achieve better performance. In this experiment, we only select well-known and fundamental heuristic policies for comparisons, such as LRU, NRU, and SRRIP.

Ten benchmarking kernels selected from various areas, including the critical kernels of artificial neural networks (ANNs), are listed in Table 2 as well as their descriptions and L1 miss per kilo-instructions (MPKI), which indicates how often the kernel accesses the L2 cache.

Table 2: Benchmarking kernels.

Name	Description	L1 MPKI
strcpy	String copy in the standard library	3.47
random	Random memory access	55.56
mco	Matrix multiplication chain order	0.00
hwdec	Haar wavelet image decompression	18.46
rlchky	Right-looking cholesky factorization	36.39
2DConv	Two-dimensional discrete convolution	79.91
llchky	Left-looking cholesky factorization	3.17
hwcom	Haar wavelet image compression	18.46
multiply	Matrix multiplication	63.01
transpose	Matrix transposition	81.56

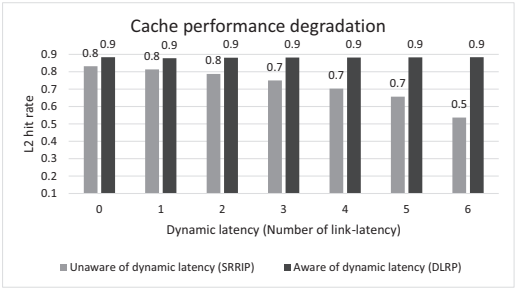

Figure 7: Cache performance degrades on kernels with long dynamic latency under a multi-application environment.

We conduct the first experiment on a controlled workload, where one core executes a benchmarking kernel, and other cores execute stream applications in parallel to emulate a multi-application environment. Table 3 shows the experimental results. Unsurprisingly, *strcpy* and *random* have no long dynamic latency and do not benefit from our proposed method (DLRP). Also, the *mco* is compute-intensive (L1 MPKI=0.00) and does not benefit from DLRP. In those three kernels, no large exponentially weighted moving average (EWMA) exists, indicating no long dynamic latency. The *hwdec* and *rlchky* also have no large EWMA, and the DLRP produces the same performance as the SRRIP. On the other hand, the *llchky* has large EWMA and 3% performance improvements compared to the LRU policy. Furthermore, the *hwcom*, *transpose*, and the ANN related kernels *2DCovn* and *multiply* show significant performance improvements, 38%, 78%, 73%, and 75%, respectively, due to more miss blocks being promoted by DLRP (>15%) as compared to *llchky* (2%). Overall, the DLRP, on average, outperforms LRU by 53% over the five kernels with long dynamic latency. Furthermore, DLRP, on average, achieves 45% and 24% more performance improvement compared to NRU and SRRIP in terms of instructions per cycle (IPC) normalized to LRU. Our method achieves 13%, 11%, and 7% higher L2 hit rates than LRU, NRU, and SRRIP over the five kernels with long dynamic latency. The performance of other cores running stream applications remains identical.

To understand if the increase of hit rate is indeed from the identification of long dynamic latency, we classify accesses by their link latency and their hit rate. In this experiment, we collect and analyze the cache accesses of *transpose*, as shown in Figure 7. Figure 7 shows the hit rates of SRRIP (without considering long dynamic latency) and DLRP (considering long dynamic latency). The x-axis is the accesses with different dynamic latency, and the y-axis the hit rate of the accesses. The figure shows that the performance of SRRIP is degraded for the accesses with long dynamic latency, while DLRP remains the same cache hit rate for all accesses.

We conduct the third experiment to evaluate a more realistic workload, consisting of kernels with/without long dynamic latency

Table 3: Experimental results.

Kernel	#large EWMA	With long dynamic latency?	IPC (normalized to LRU)			#promoted misses by DLRP (%)	L2 hit rate			
			NRU	SRRIP	DLRP		LRU	NRU	SRRIP	DLRP
strcpy	0	No	1.00	1.00	1.00	-	0.00	0.00	0.00	0.00
random	0	No	1.00	1.00	1.00	-	0.00	0.00	0.00	0.00
mco	0	No	1.00	1.00	1.00	-	0.00	0.00	0.00	0.00
hwdec	0	No	1.00	1.00	1.00	-	0.21	0.20	0.21	0.21
rlchky	0	No	0.98	0.99	0.99	-	0.12	0.12	0.12	0.12
Avg.	0	No	1.00 (-0.01)	1.00 (-0.00)	1.00 (-0.00)	-	0.07 (-0.00)	0.06 (-0.01)	0.07 (-0.00)	0.07 (-0.00)
llchky	7297	Yes	1.00	1.00	1.03	0.02	0.01	0.01	0.01	0.04
2DConv	55034	Yes	1.08	1.15	1.73	0.31	0.08	0.12	0.15	0.34
hwcom	62988	Yes	1.00	1.32	1.38	0.16	0.03	0.03	0.07	0.07
multiply	144544	Yes	1.17	1.48	1.75	0.17	0.04	0.08	0.15	0.21
transpose	151958	Yes	1.16	1.51	1.78	0.16	0.04	0.08	0.15	0.20
Avg.	>0	Yes	1.08 (-0.45)	1.29 (-0.24)	1.53 (-0.00)	0.16	0.04 (-0.13)	0.06 (-0.11)	0.11 (-0.07)	0.17 (-0.00)

Table 4: The average IPC of mixed kernels with long dynamic latency (DL), kernels without long dynamic latency (NDL), and compute-intensive (CI) kernels (normalized to LRU).

Workload	Memory-intensive		Compute-intensive (CI) (MPKI=0.00)
	With long dynamic latency (DL)	No long dynamic latency (NDL)	
1DL, 3NDL, 12CI	1.05	1.00	1.00
1DL, 4NDL, 11CI	1.14	1.00	1.00
1DL, 5NDL, 10CI	1.29	0.99	1.00
1DL, 6NDL, 9CI	1.37	0.99	1.00
1DL, 7NDL, 8CI	1.53	0.99	1.00
1DL, 8NDL, 7CI	1.54	0.99	1.00

and compute-intensive kernels. In this experiment, one workload consists of both memory-intensive and compute-intensive kernels. The memory-intensive kernels include kernels with/without long dynamic latency. In the experiment, the kernel with long dynamic latency (DL) is a single *2DConv* kernel and executed with various numbers of kernels without dynamic latency (NDL) and compute-intensive (CI) kernels. Table 4 shows the average IPC normalized to LRU. For example, the first workload consists of a kernel with long dynamic latency (*2DConv*) executed on one core, kernels without long dynamic latency (*random*, *strcpy*, and *rlchky*) executed on three cores, and compute-intensive kernels (*mco*) executed on the other 12 cores. From the table, the IPC of the kernel with long dynamic latency has 5% to 54% improvements compared to LRU. Meanwhile, the average IPC degradation on the kernels without long dynamic latency remains below 1%. Furthermore, all compute-intensive kernels remain the same IPC. Overall, our proposed DLRP can achieve better performance on the kernel with long dynamic latency with a relatively small performance impact on others in our experimental workloads.

In our proposed DLRP, each application only has a single monitor that monitors a small subset of random accesses to represent the application behavior. Adding more monitors may attain more application behavior. We conducted an experimental comparison on a single monitor and 1000 monitors. The results show that DLRP with more monitors will not further improve the performance compared to the one with a single monitor. Thus, a single monitor is adequate to capture the application behavior, which is very efficient in terms of hardware area and energy overheads.

6 CONCLUSIONS

In this paper, we have focused on the multiprocessor system-on-chips (MPSoCs) with a non-uniform shared cache (NUCA). We have discovered that dynamic latency is an essential index to cache hit rate. For the patterns with long dynamic latency may result in performance degradation under multi-application environments. We propose an efficient online hardware mechanism to identify such patterns. Also, hardware and energy-efficient cache replacement policy with an online hardware monitor obtaining application behaviors, the dynamic link-latency aware replacement policy (DLRP), has been developed to target the dynamic latency. The experimental results show that the DLRP, on average, outperforms LRU by 53% on a set of benchmarking kernels with long dynamic latency. Furthermore, our method, on average, achieves 45% and 24% more performance improvement compared to NRU and SRRIP replacement policy normalized to LRU. The results have demonstrated that our method is beneficial and effective for applications containing substantial kernels with long dynamic latency, including the kernels of fully-connected and convolutional layers, in the artificial neural network (ANN).

REFERENCES

[1] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002.* (K. Gharachorloo, ed.), pp. 211–222, ACM Press, 2002.

[2] D. S. Gracia, G. Dimitrakopoulos, T. M. Arnal, M. G. H. Katevenis, and V. V. Yufera, "Lp-nuca: Networks-in-cache for high-performance low-power embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, pp. 1510–1523, Aug 2012.

[3] Y. Ding and W. Zhang, "Wcet analysis of static nuca caches," in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pp. 1–6, Dec 2014.

[4] N. Beckmann, P. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 538–550, Feb 2015.

[5] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), p. 113–127, Association for Computing Machinery, 2016.

[6] Q. Wu and Z. Ji, "A reuse-degree based locality classifier for locality-aware data replication," *IEEE Access*, vol. PP, pp. 1–1, 12 2019.

[7] A. Boro, B. Thomas, S. R. Ahamed, and G. Trivedi, "Fpga implementation of a dedicated processor for temperature prediction," in *2016 International Conference on Accessibility to Digital World (ICADW)*, pp. 21–26, Dec 2016.

[8] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France* (A. Seznec, U. C. Weiser, and R. Ronen, eds.), pp. 60–71, ACM, 2010.

[9] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 413–425, Association for Computing Machinery, 2019.

[10] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "Rstensorflow: GPU enabled tensorflow for deep learning on commodity android devices," in *Proceedings of the 1st International Workshop on Embedded and Mobile Deep Learning (Deep Learning for Mobile Systems and Applications), EMDL@MobiSys 2017, Niagara Falls, NY, USA, June 23, 2017*, pp. 7–12, ACM, 2017.