Dynamic Cache Management in Multi-Core Architectures through Run-time Adaptation

Fazal Hameed, Lars Bauer, and Jörg Henkel
Karlsruhe Institute of Technology, Chair for Embedded Systems, Karlsruhe, Germany
{hameed, lars.bauer, henkel} @ kit.edu

***Abstract*–Non-Uniform Cache Access (NUCA) architectures provide a potential solution to reduce the average latency for the last-level-cache (LLC), where the cache is organized into per-core *local* and *remote partitions*. Recent research has demonstrated the benefits of cooperative cache sharing among local and remote partitions. However, ignoring cache access patterns of concurrently executing applications sharing the local and remote partitions can cause inter-partition contention that reduces the overall instruction throughput. We propose a dynamic cache management scheme for LLC in NUCA-based architectures, which reduces inter-partition contention. Our proposed scheme provides efficient cache sharing by adapting *migration, insertion*, and *promotion* policies in response to the dynamic requirements of the individual applications with different cache access behaviors. Our adaptive cache management scheme allows individual cores to steal cache capacity from remote partitions to achieve better resource utilization. On average, our proposed scheme increases the performance (instructions per cycle) by 28% (minimum 8.4%, maximum 75%) compared to a private LLC organization.**

1. Introduction and Related Work

Multi-core systems are expected to increase the last-level-cache (LLC) capacity to accommodate the working sets for memory intensive applications. However, increasing the LLC capacity increases the on-chip interconnect delay. As compared to the transistor delay, interconnect delay does not scale at the same rate with each technology node [1-4]. Increasing interconnect delay has made Non-Uniform Cache Access (NUCA) architectures a promising solution compared to traditional Uniform Cache Access (UCA) architectures. The concept of NUCA architectures is based on the non-uniformity of access time, where the access time depends upon the physical location of the line relative to the core. LLC in NUCA-based multi-core systems is organized into per-core *local partitions* and *remote partitions*, each of which has different access latencies. A significant body of work studied LLC management to provide better throughput and fairness [5-12].

Previous studies [5-7] have proposed several NUCA schemes by combining the strengths of private LLC organizations (i.e. local partition is only accessible to the requesting core) and shared LLC organizations (i.e. all partitions are shared by all cores providing equal capacity sharing). In such a hybrid scheme, each core is provided with a local partition, which can be shared with other cores. On a miss in the local partition, all of the remote partitions are searched until the request is satisfied or an LLC miss is detected. In case of an LLC miss, the main memory is accessed and the data is brought to the local partition of the requesting core. Cooperation among different local partitions was earlier proposed [5], where cache lines that are evicted from local partitions (*victim lines*) are stored in remote partitions (called *spilling*). This uncontrolled spilling of victim lines to remote partitions causes cache pollution. The research work in [5] focused on controlling the degree of cooperation between different partitions by dynamically tuning the partition size of each local partition, where a part of the local partition is only accessible to the local core (private portion), while the remaining part is accessible to the remote cores (shared portion). The sizes of the private and shared portions of the local partitions are controlled dynamically on a per-core basis.

Previously proposed NUCA-based approaches [5-7] do not consider application access patterns of the competing applications sharing local and remote partitions, which can cause contention leading to ineffective utilization of cache resources. As future multi-core architectures are expected to have a large number of cores, cache contention caused by concurrently running application will increase as well. Research work in mitigating cache contention has been carried out recently for a shared LLC in UCA-based architectures [12]. That work uses a Utility Monitoring Circuit (UMON) [13] for adapting the cache replacement policy in the shared LLC by tracking run-time miss rate information of the individual applications. However, the use of UMON comes at the cost of additional hardware overhead for maintaining shadow tags (a shadow tag is similar to a regular cache structure except that it only contains the tag array). In this paper, we focus on the cache management in NUCA-based multi-core architectures by allocating cache resources to competing applications in response to the diverse application requirements. There has been a considerable amount of work on managing NUCA caches [5-7] and we compare our results with the most recently proposed *Dynamic spill-receive* (*DSR*) architecture [6] as it provides a low-overhead cooperative caching between different partitions.

LLC size	art	bzip	mcf	milc	lbm	gos	povray
0.5 MB	79.8	6.86	53.34	18.65	26.15	0.694	0.0214
1 MB	56.14	4.35	37.68	18.65	26.14	0.520	0.0196
2 MB	0.04	1.47	23.80	18.64	26.13	0.350	0.0193
4 MB	0.001	0.69	16.86	18.61	26.13	0.213	0.0193

Table 1: Case study: LLC misses per thousand instructions (MPKI)

The LLC in our baseline architecture is realized as four equally sized Local Partitions (LPs, one per core) and assumes an LP size of 1 MB per core. Applications vary widely in terms of their cache requirements. Table 1 illustrates this observation showing LLC misses per thousand instructions (MPKI) for different LLC sizes and different applications. Increasing the LLC size of some applications (e.g. *art*, *bzip*, and *mcf*) incurs significant reduction in MPKI. These applications benefit from an increased cache capacity and are classified as *Taker* applications. The applications whose LLC requirement is far less than the cache space available in the LP are classified as *Giver* applications (e.g. *gos* and *povray* already have an MPKI < 1 for 0.5 MB LLC).

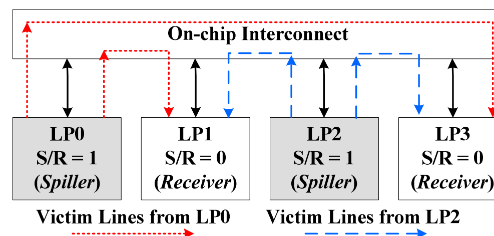

Figure 1: Overview of the DSR architecture [6]

978-3-9810801-8-6/DATE12/©2012 EDAA

Figure 1 shows an overview of the DSR architecture [6] for a 4-core system. The DSR architecture allows stealing cache resources from remote partitions in response to the cache demands of individual cores. The LP of the core executing a Taker application (at most one application executes per core) acts as a *spiller partition* to spill evicted victim lines to the remote partitions for later reuse. The LP of the core executing a Giver application acts as a *receiver partition* to provide some of its capacity to the Taker applications. In the DSR architecture, a victim line from a spiller partition is stored in a randomly chosen receiver partition, whereas a victim line from a receiver partition is spilled to the main memory. Each LP is provided with one bit named as *S/R* that decides whether that LP will act as a spiller partition (S/R=1) or a receiver partition (S/R=0). In Figure 1, the S/R bit of partitions LP0 (local partition associated with Core 0) and LP2 are set to 1 indicating that these partitions act as spiller partitions. Therefore, victim lines from LP0 and LP2 are spilled randomly to any of the remote receiver partitions, i.e. LP1 or LP3. Evicted victim lines from LP1 and LP3 are spilled to main memory. Each local partition learns the spiller-receiver decision using dynamic set sampling (details can be found in [6]).

Comparing across the applications, we found that the cache requirements do not correlate linearly with the increase in the cache demand. For example, *lbm* and *milc* incur a high miss rate and a high access rate (see Table 1). These applications have large working sets with very poor locality and do not get any benefit from increasing the cache resources. Instead, that would lead to inter-partition contention in the DSR architecture (details in Section 2). The main drawback of the DSR architecture is that it statically determines the line migration, insertion, and promotion policies (explained below) and suffers from inter-partition contention. If an application requires a different policy to improve the instruction throughput then this cannot be achieved when using a fixed policy. The main objective of this paper is to realize a low-overhead mechanism to reduce inter-partition contention with dynamic migration, insertion, and promotion policies. The differences in comparison to the DSR architecture and our novel contributions are:

1. We propose a *dynamic migration policy*, that decides at runtime, whether a line that receives a hit in the remote partition shall be migrated to the local partition of the requesting core or not. Our dynamic migration policy detects and avoids unnecessary migrations between local and remote partitions, whereas the DSR architecture always migrates the remote cache-hit lines to the local partition.
2. We propose a *dynamic insertion policy* (decides the insertion position for the incoming cache line in case of a cache miss) and a *dynamic promotion policy* (decides how a cache line moves towards the most recently used (MRU) position on receiving a cache hit) for the local and remote requests in response to the dynamic requirements of the applications. The DSR architecture uses the traditional least recently used (LRU) policy for line insertion and promotion.
3. Our proposed approach attempts to minimize the cache resources allocated to applications with streaming and thrashing behavior, since they get little benefit from increasing the cache resources. Cache thrashing occurs for applications that have a working set size that is significantly larger than the available cache resources [12, 14].

2. Dynamic Cache Management

Our proposed cache management scheme is based on the concept of NUCA architectures, where hits in the local partition are faster than hits in the remote partitions. On a miss in the local partition, all of the remote partitions are searched until the request is satisfied or an LLC miss is detected. An LLC miss will require an off-chip access, where the main memory is accessed and the cache line is brought to the local partition of the requesting core. For cooperative sharing among different partitions, we employ the recently proposed *set dueling* technique to learn whether a partition will act as a spiller partition or a receiver partition [6, 14].

2.1. Dynamic migration policy

In the previous NUCA schemes [5-7], on a miss in the local partition, all remote partitions are queried by en-queuing the request for determining a hit or a miss. In case of a remote hit in the DSR architecture [6], the cache line is invalidated in the remote partition, and migrated to the local partition of the requesting core (*migrate policy*) exploiting the temporal locality that the referenced line will be accessed in the near future. For example, Figure 1 illustrates two spiller and two receiver partitions. If a line for core 0 (LP0 is a spiller partition) hits in LP1 (receiver partition with core 1), then the DSR architecture migrates the hit line to LP0. The line evicted from the LP0 (as a result of the migration) is transferred to the receiver partition LP1. The migrate policy exploits the temporal locality to improve the local partition hit ratio. A migrated cache line that is not referenced between its insertion into the local partition and spilling it again is called *zero reuse migrated* line. It is not beneficial to migrate such a line into the local partition if its *reuse distance* (the number of insertions made before the line is reused) is greater than the local partition associativity as it will be spilled before it is reused. For applications with a rather large number of zero reuse migrated lines (e.g. *art* exhibits this behavior), the majorities of migrated lines reside in the local partition without contributing to cache hits and finally get evicted/spilled to remote partitions, resulting in frequent migrations between local partition and remote partition affecting the cache efficiency.

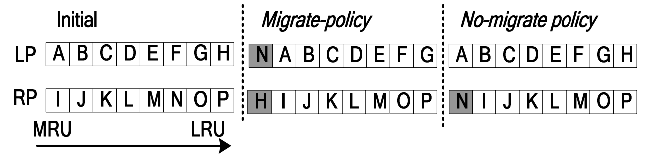

Figure 2: Example illustrating the migrate- and no-migrate policy for a hit to Line N in the remote partition for the LRU replacement policy LP: local partition, RP: remote partition

In this paper, we introduce the *no-migrate policy* and dynamically switch between migrate and no-migrate policy at run time. In the no-migrate policy, a cache line that receives a hit in a remote partition is transferred to the requesting core to handle its request without being installed in the local partition of the requesting core. Figure 2 shows an example illustrating the migrate- and no-migrate policy on receiving a cache hit in a remote partition for line N. In the migrate policy, line N is migrated to the local partition of the requesting core by placing it in its MRU position. This migration will cause eviction of line H in the local partition, which is then installed in the MRU position of the remote partition. In case of the no-migrate policy, line N is not installed in the local partition but promoted to MRU position in the remote partition.

Figure 3 shows the IPC of four different applications running on a quad-core system, when using the migrate- or no-migrate policy for the *art* application and the no-migrate policy for the other applications (these applications benefit from the no-migrate policy; the details of the experimental setup are presented in Section 4). The *art* benchmark shows a significant improvement in the IPC (58%) for the no-migrate policy compared to the migrate policy. While applying the migrate

policy for *art*, we found that about 82% of the lines that are migrated to the local partition of the core that executes the *art* benchmark are not re-used before spilling the lines again to a remote partition. Figure 4 shows the LLC MPKI for *art* as a function of number of ways allocated to it. The *art* benchmark obtains significant benefits from the extra cache capacity beyond its local partition size as its MPKI reduces significantly when extra cache resource are allocated to it. The *art* benchmark is determined as a Taker application by DSR, but has a much lower reuse frequency in the local partition and gets significant benefits with the no-migrate policy compared to migrate-policy.

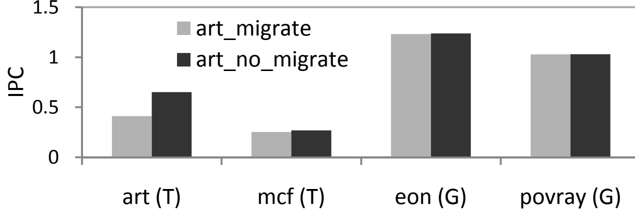

Figure 3: Individual benchmark instruction per cycle (IPC)
T: Taker application, G: Giver application

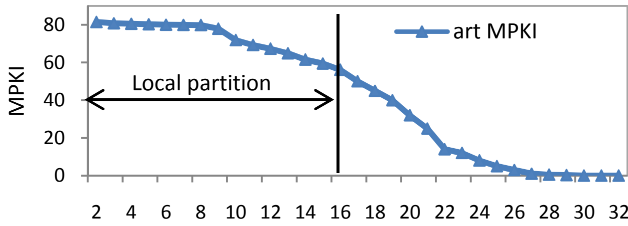

Figure 4: LLC misses per thousand instructions (MPKI) for art as a function of number of ways

We propose a dynamic migration policy that decides at runtime whether a line that hits in a remote partition shall be migrated or not. Algorithm 1 shows our proposed dynamic migration policy. If an application remote hit ratio exceeds the local partition hit ratio, then the requesting core chooses the migrate policy, otherwise the requesting core selects the no-migrate policy. The intuition behind our dynamic migration policy is that a high local hit ratio favors the migration policy (for later reuse of the cache line), while a high remote hit ratio favors the no-migrate policy (to mitigate the negative impact of zero reuse migrated lines). When deciding whether the migration or the no-migration policy shall be used it may happen that the core switches through different policy changes before converging on a good one. To address this issue, we invoke our dynamic migration policy after 4096 misses (interval period T_i in Algorithm 1) in the LP to prevent rapid policy changes. Once a policy change is made, it remains until 4096 misses occur in the LP.

N_{AL} Number of accesses to the *local partition*
N_{HL} Number of hits in the *local partition*
N_{AR} Number of accesses to the *remote partitions*
N_{HR} Number of hits in the *remote partitions*
1 At the end of an interval period $\boldsymbol{T_i}$
2 **for all** *Taker partitions* **do**
3 **if** $(N_{HL} / (N_{AL} + N_{AR}) < N_{HR} / N_{AR})$ **then**
4 choose *migrate policy* for the next time interval $\boldsymbol{T_{i+1}}$
5 **else**
6 choose *no-migrate policy* for the next time interval $\boldsymbol{T_{i+1}}$
7 **end if**
8 **end for**
9 Reset N_{AL}, N_{HL}, N_{AR}, and N_{HR} monitoring

Algorithm 1: Our Dynamic Migration Policy

2.2. Dynamic insertion and promotion policy

Cache replacement policies can be categorized into two parts: *insertion policy* and *promotion policy* [14, 12]. The priority position of a cache line determines the eviction policy on a cache miss. Figure 5-a shows a logical organization (lines are shown from left to right in priority order, the physical order in the set may differ) of a cache set with priority values assigned to each line that decide the priority position of the line in the set. The cache line with the least priority (Line H with a priority value of 1) is the candidate for eviction to make room for the incoming line on a cache miss. On receiving a cache miss, the *insertion policy* decides the *insertion position* for the incoming line in the priority list [14] which requires modification to conventional least recently used (LRU) policy. In Figure 5-a, the LRU replacement policy evicts Line H from the lowest priority position 1 to make room for the new incoming Line I that is inserted into the highest priority position 8. The *promotion policy* decides how a cache line moves towards the highest priority position on receiving a cache hit. In an LRU based cache, a hit causes the line to move towards the highest priority position. In the example shown in Figure 5-b, Line E is promoted to the highest priority position 8 on receiving a cache hit.

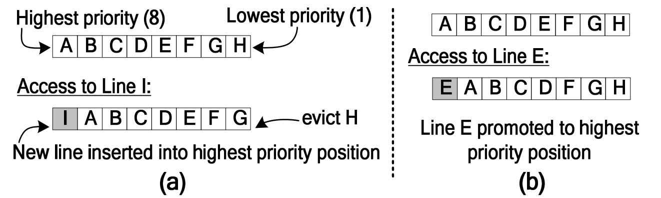

Figure 5: Example illustrating (a) LRU Insertion policy and (b) LRU Promotion policy

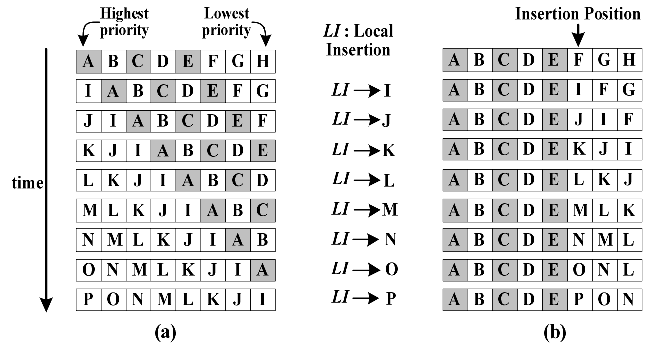

Figure 6: Example illustrating (a) Inter-partition contention (b) Performance Isolation between local and remote partition (remote lines shown in grey boxes)

The LRU replacement policy (as for instance used by the DSR architecture [6]) does not work well with applications that have streaming or thrashing behavior. These applications have a high cache access rate with a working set size greater than the available cache size and thus get negligible benefits from extra cache capacity. Applications that exhibit streaming or thrashing behavior are for instance *lbm*, *libquantum*, *mesa*, and *milc*. The local partition of a core running a streaming application is not efficiently utilized due to many rarely reused lines. Streaming applications are typically classified as Giver applications because they do not benefit noticeably from larger cache capacity and thus allow Taker applications to spill their lines into the local partition of the streaming application. However, they have a high access rate to their local partition relative to the access rate of Taker applications to the remote lines in this partition. This means, they insert a large number of lines in the local partition, and as a result, they quickly

evict useful remote cache lines. The eviction of useful remote cache lines increases the contention between local and remote partitions. Figure 6-a illustrates a receiver partition running a streaming application with LRU-based replacement policy, which initially contains some useful lines from the remote partitions (shown in grey boxes). As the local core inserts more lines into its local partition, the useful remote lines are evicted. Subsequent accesses to these useful remote lines will result in cache misses, hereby affecting performance. Thus, by inserting the rarely used local lines in the highest priority position, the LRU replacement policy increases inter-partition contention. The occupancy time of the rarely used lines can be reduced, if they are inserted into low priority positions. Figure 6-b shows how a local cache line for streaming applications is inserted into a low priority position (i.e. *insertion position* of 3), hence providing inter-partition performance isolation/protection. Since local lines exhibit little reuse for streaming applications, they are evicted quickly as more lines are inserted. The useful remote lines are maintained in the highest priority position, which protects them from the streaming behavior of the local core.

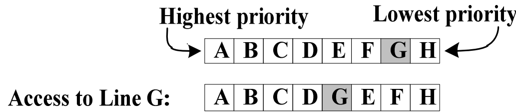

Figure 7: Example: promotion of a line by +2 on receiving a cache hit

In an LRU-based cache, lines that receive a hit are promoted to the highest priority position, as shown in Figure 5-b. However, the LRU-based promotion policy does not perform well for streaming applications that exhibit rarely reuse behavior. If the local lines for streaming applications are moved by a smaller *promotion distance* (i.e. the distance by which the priority position of the line is changed on receiving a cache hit), it will not only reduce the occupancy time for the rarely used lines, but also prevent eviction of useful cache lines belonging to remote partitions. As shown in Figure 7, Line G is promoted by a smaller promotion distance (promote by +2) on receiving a hit. Promotion of the rarely reused lines by a smaller distance will keep the highly reused lines towards the highest priority position and rarely reused lines towards the lowest priority position, hereby increasing the cache efficiency.

Let A be the associativity of the local partition. The insertion position IP of an incoming line can be defined as a value between 1 and A. An incoming line with $IP=A$ indicates that the line is inserted into the MRU position (highest priority), whereas an incoming line with $IP=1$ indicates that the line is inserted into the LRU position (lowest priority). Similarly, the promotion distance PD of a hit line can have a value between 1 and A. A local partition with $PD=A$ will always promote the cache hit line to MRU position, whereas a local partition with $PD=1$ will promote the cache hit line by a single priority position. The primary objective of our dynamic insertion and promotion policy is to reduce the cache resources that are allocated to applications with streaming or thrashing behavior. This will effectively reduce the occupancy time of the rarely reused lines for streaming applications. These cache resources can be provided to the spiller partitions. In our approach, the values for IP and PD are adapted at run-time for each local partition.

Algorithm 2 shows our proposed dynamic insertion/promotion policy for an A-way set associative local partition. An application's cache behavior is determined using runtime profiling via the processor's performance counters. If an application running on a core with a receiver partition (i.e. the local partition does not get any benefit from extra cache capacity) has a higher local miss rate than a certain threshold *thr1*, then the application is classified to exhibit extreme streaming (*ES*) behavior. In such a case, our dynamic scheme allocates less cache resources to that local core by inserting the incoming local lines into the low priority positions as illustrated in Figure 8. The remote requests (i.e. evicted victim lines from the spiller partitions) are inserted into high priority positions to give preference to the spiller partitions. In such a case, the spiller partitions steal more cache resources from the receiver partition that exhibits streaming or thrashing behavior. Different stream detection thresholds (thr_1, thr_2, and thr_3) are used to classify the cache access behavior (*ES*, *MS*, *LS*, and *LF* as shown in Algorithm 2). Similarly, the cache lines on receiving a hit are moved towards high priority position by the promotion distance that is adapted at runtime. The lines are moved by a smaller promotion distance in case of a high local miss ratio (poor locality), while they are moved by a larger promotion distance in case of a low local miss ratio (good locality). In such a case, heavily used lines will be promoted towards the high priority position and infrequently used lines will be moved towards the low priority positions, hereby reducing the occupancy time of the rarely reused lines.

A	Associativity of the local partition
IL	*Insertion* position for the *local* request
IR	*Insertion* position for the *remote* request
PL	*Promotion* distance for the *local* request
PR	*Promotion* distance for the *remote* request
ES	Extreme Streaming
MS	Moderate Streaming
RS	Reduced Streaming
LF	LRU Friendly
N_{AL}	Number of local accesses
N_{ML}	Number of local misses
thr_1, thr_2, thr_3	Streaming/Thrashing detection thresholds

1 At the end of an interval period T_i
2 **for each** partition p **do**
3 **if** (all local partitions are receiver partitions **or**
4 p is a spiller partition) **then**
5 LRU policy : $IL = A$, $PL = A$, $IR = A$, $PR = A$
6 **else if** ($N_{ML}/N_{AL} \geq thr_1$) **then**
7 Type ES : $IL = A/8$, $PL = A/4$, $IR = A$, $PR = A$
8 **else if** ($N_{ML}/N_{AL} \geq thr_2$) **then**
9 Type MS : $IL = A/4$, $PL = A/2$, $IR = A$, $PR = A$
10 **else if** ($N_{ML}/N_{AL} \geq thr_3$) **then**
11 Type RS : $IL = A/2$, $PL = 3A/4$, $IR = A$, $PR = A$
12 **else** // receiver partition has a low miss rate
13 Type LF : $IL = A$, $PL = 3A/4$, $IR = 3A/4$, $PR = 3A/4$
14 **end if**
15 **end for**
16 Reset N_{AL} and N_{ML}
17 Apply the insertion and promotion policy for the next time interval T_{i+1}

Algorithm 2: Dynamic Insertion/Promotion Policy for an A-way set associative local partition

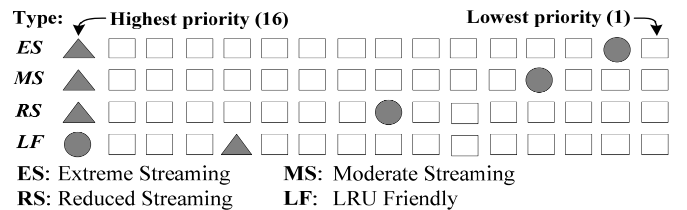

Figure 8: Example of a 16-way set associative partition, illustrating dynamic insertion position for local (circles) and remote (triangles) requests

Table 2 shows the average cache miss rate of different applications (when running alone) for a 4 MB LLC. The streaming benchmarks (*lbm*, *milc*, *libquantum*, and *mesa*) exhibit high cache miss rates (closer to 0.5 for most of the streaming applications) even if the entire 4 MB LLC is available exclusively for that application. The threshold values used in Algorithm 2 are determined based on the high miss rates of streaming applications. The threshold values (thr_1 = 0.5, thr_2 = 0.25, thr_3 = 0.125) are chosen for easy computation. For instance, computing $N_{ML}/N_{AL} \geq 0.5$ (or 0.25 or 0.125, respectively) will require right-shifting N_{ML} by 1 (or 2 or 3, respectively) and compare with N_{AL} (e.g. $N_{ML}/0.5 \geq N_{AL}$). The priority positions *IL*, *IR*, *PL*, and *PR* are determined at run time based on the threshold values as shown in Algorithm 2. Our dynamic scheme allocates the resources to each application considering the streaming behavior of the individual applications. A dynamic insertion policy has been carried out recently for a shared LLC in Uniform Cache Access (UCA) architectures [12]. That work uses a Utility Monitoring Circuit (UMON) [13] for adapting insertion policy in the shared LLC by collecting run-time miss rate information of the individual applications with different associativity. However, the use of UMON comes at the cost of additional hardware overhead, which requires per-core storage overhead of 7.4 KB, altogether leading to a 29.6 KB storage overhead for a quad-core system. In this paper, we apply our dynamic insertion and promotion policy to a NUCA-based multi-core architecture to reduce inter-partition contention between local and remote requests with least hardware overhead (for hardware overhead see Section 2.3).

Application	**lbm**	**milc**	**libquantum**	**mesa**	**art**
Miss rate	0.519	0.733	0.499	0.482	0.0001
Application	**deal**	**gos**	**bzip**	**mcf**	**eon**
Miss rate	0.0018	0.0026	0.026	0.051	0.0006

Table 2: Cache miss rate for entire 4 MB LLC

Our proposed dynamic insertion and promotion policy exhibits its benefits in situations where streaming applications run concurrently with Taker applications. In such a case, the Taker applications are able to snatch large fractions of cache resources from the streaming applications without degrading their performance. For the corner case where all of the partitions are receiver partitions, our dynamic scheme reverts to the LRU policy (Line 3 in Algorithm 2) to prevent underutilization of each local partition (because none of the local partition benefits from extra cache capacity beyond local partition size). The traditional LRU scheme allocates cache resource to each application based on demand rather than locality. In LRU based caches, applications with higher access frequencies (greater demand) and higher miss rate (thrashing behavior) will occupy more cache resources, which may lead to ineffective utilization of the cache resources.

2.3. Hardware overhead

Our proposed scheme requires four registers (to keep track of the local and remote partition hit ratio) per-core to decide the migration policy as illustrated in Algorithm 1. Each partition requires a 4-bit migration policy vector (for a 4-core system) to decide the migration policy for the remote partitions. Our dynamic insertion and promotion policy illustrated in Algorithm 2 can be implemented either in hardware or in software and requires monitoring of the overall miss ratio to decide the insertion position and promotion distance for local and remote requests. A hardware implementation will require 4 priority encoders (to decide *IL*, *IR*, *PL*, and *PR*) per LP. In terms of storage overhead, we need registers to keep track of the miss and access statistics for the current and previous time intervals. Altogether, our proposed scheme comes with negligible per-core hardware/software overhead.

3. Experimental Setup

We use the x86 version of SimpleScalar (zesto) [15] to simulate a quad-core system. The processor is clocked at 3.2 GHz with 32 KB I-Cache and 32 KB D-Cache with 3 cycles latency. We use an 80-entry reorder buffer, 32-entry reservation station, 24-entry load queue, and 20-entry store queue. The branch misprediction penalty is assumed to be 14 cycles with a four-wide decode and commit width. The main memory is modeled as DRAM with 400 MHz front side bus. The last-level-cache local partition size is chosen to be 1 MB with a 10 cycle hit latency. Cache hits in the remote partition incur an additional latency of 30 cycles. Our performance evaluations make use of various multi-programmed workloads from SPEC2000 and SPEC2006 [16], as shown in Table 3. Each application is determined either as Taker (T) or Giver (G) application in the DSR architecture. The applications that exhibit thrashing behavior are denoted as Streaming (S) application. Note that streaming application are determined as Giver (S/R = 0) application by the DSR architecture. For each benchmark, we used the Simpoint tool [17] to select representative samples. For each benchmark, we collect simulation statistics for 250 million instructions with a fast-forward of 500 million instructions (to warm up the caches and branch predictors in functional mode). When a shorter benchmark finishes early by completing its 250 million instructions, then it is restarted and continues to contend for the cache and bus resources. However, the simulation statistics are reported for the first 250 million instructions after the fast-forward.

Name	Benchmarks
Mix_1	art (*T*), eon (*G*), mcf (*T*), povray (*G*)
Mix_2	mcf (*T*), eon (*G*), gos (*G*), libquantum (*S*)
Mix_3	gos (*G*), art (*T*), bzip (*T*), lbm (*S*)
Mix_4	bzip (*T*), milc (*S*), mesa (*S*), lbm (*S*)
Mix_5	deal (*G*), lbm (*S*), bzip (*T*), art (*T*)
Mix_6	mcf (*T*), bzip (*T*), deal (*G*), sjeng (*G*)
Mix_7	eon (*G*), art (*T*), libquantum (*S*), art (*T*)
Mix_8	bzip (*T*), lbm (*S*), bzip (*T*), libquantum (*S*)
Mix_9	mcf (*T*), libquantum (*S*), deal (*G*), milc(*S*)

Table 3: Benchmark Workload
T: Taker, G: Giver, S: Streaming

4. Experimental Results

To evaluate the performance of our proposed scheme, we use three metrics for comparison: overall throughput, harmonic mean (HM) and best-case performance improvement compared to a private last-level-cache (LLC) organization [5, 7]. In a private LLC organization, the local partition is only accessible to the requesting core and there is no capacity stealing between partitions. We have compared our scheme with the state-of-the-art *Dynamic spill-receive* (*DSR*) cache management technique [6] that shares the same philosophy of stealing cache resources from the remote partitions but does not consider application's cache access behavior for cooperative inter-partition cache sharing. DSR uses static policies, whereas we use our proposed dynamic migration, insertion, and promotion policies.

We observe improved benefits for most workload types. Figure 9 shows the percent improvement in instruction throughput for DSR [6] and our proposed scheme compared to a private LLC design. On average, our proposed scheme in-

creases the instruction throughput by 28% (minimum 8.4%, maximum 75%) compared to a private LLC organization. In comparison with the DSR architecture, it increases the instruction throughput by 12.5%. We also evaluate the performance of our proposed scheme against the DSR architecture using the harmonic mean fairness metric (M is the number of applications) which is given as:

$$\text{HM} = \frac{M}{\sum_{i=1}^{M} \frac{1}{IPCi}}$$

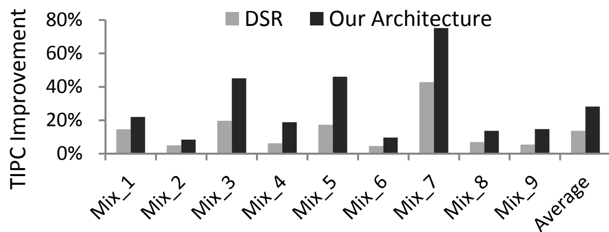

Figure 9: Total instruction per cycle (TIPC) improvement relative to a private LLC organization

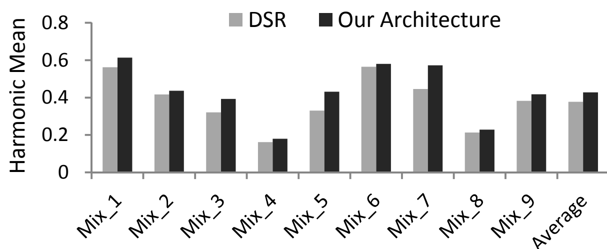

Figure 10: Harmonic Mean (HM) for DSR and our architecture

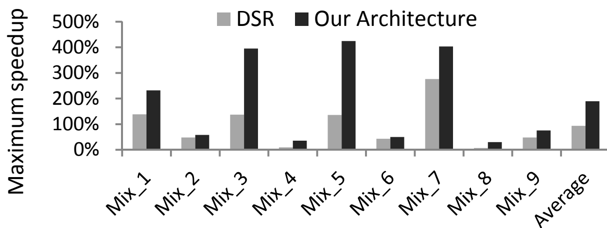

Figure 11: Individual benchmark best–case performance for DSR and our scheme compared to a private LLC organization

Figure 10 shows the HM fairness metric for DSR and our proposed architecture. On average, our proposed architecture improves HM speedup by 13.3% (minimum 3%, maximum 30%) compared to the DSR architecture. Maximum gains are observed, where the applications with streaming behavior execute together with Taker applications (Mix_3, Mix_5, Mix_7, and Mix_9). In such a case, Taker applications are able to steal large fractions of cache resources from the Streaming applications. Thus, our approach not only increases the overall throughput but it also balances fairness (13.3% improvement in HM speedup) compared to the DSR architecture. Figure 11 shows the individual benchmark best-case performance improvement for DSR and our proposed policy compared to a private LLC organization. On average, our proposed policy increases the best-case performance by 190%, while DSR increases the best-case performance by 94% compared to a private LLC organization.

5. Conclusions

Low latency on-chip cache access with reduced off-chip memory traffic is a goal for future multi-core architectures. Increasing on-chip interconnect delays, inter-cache contention, and the requirements of memory intensive workloads necessitate efficient cache management in order to satisfy the conflicting requirements to improve the overall throughput and fairness. This paper presents an efficient cache management scheme for multi-core systems that reduces average cache latency by increasing the number of hits in the local partitions. It provides dynamic insertion, promotion, and migration policies that dynamically allocate cache resources based on the requirements of the individual applications. Our adaptive cache management scheme reduces inter-partition contention and provides better resource sharing by stealing cache resources from remote receiver partitions via monitoring each partition at runtime. Our proposed scheme increases the average instruction throughput by 28% and 12.5 % compared to a private LLC organization and the DSR architecture [6], respectively. Our proposed cache management scheme considering various dynamic policies comes with limited hardware/software overheads.

References

[1] "Standard Performance Evaluation Corporation", http://www.itrs.net.

[2] R. Kumar and D. Tullsen, "Interconnections in Multicore Architectures: Understanding Mechanisms, Overheads and Scaling", in *ISCA-32*, 2005, pp. 408–419.

[3] J. M. Rabaey and S. Malik, "Challenges and Solutions for Late- and Post-Silicon Design", *Design and Test*, vol. 25, no. 4, pp. 292–308, 2008.

[4] B. Rogers, A. Krishna., G. Bell *et al.*, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling", *SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 371–382, 2009.

[5] J. Chang and G. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors", in *Int'l Conference on Supercomputing (ICS)*, 2007, pp. 242–252.

[6] M. K. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs", in *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2009, pp. 45–54.

[7] H. Dybdahl and P. Stenström, "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors", in *IEEE Symposium on High-Perf. Computer Architecture (HPCA)*, 2007, pp. 2–12.

[8] M. K. Qureshi, D. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement", in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, June 2006, pp. 167–178.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Contention on a Chip Multi-Processor Architecture", in *IEEE Symposium on High-Perf. Computer Architecture (HPCA)*, 2005, pp. 340–351.

[10] J. Lin, Q. Lu, X. Ding *et al.*, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems", in *IEEE Symp. on High-Perf. Comp. Arch. (HPCA)*, 2008, pp. 367–378.

[11] C.Yu and P. Petrov, "Off-Chip Memory Bandwidth Minimization through Cache Partitioning for Multi-Core Platforms", in *Proceedings of the 47th Design Automation Conference (DAC)*, 2010, pp. 132–137.

[12] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches", in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 174–183.

[13] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-performance, Runtime mechanism to Partition Shared Caches", in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 423–432.

[14] A. Jaleel, W. Hasenplaugh, M. Qureshi *et al.*, "Adaptive Insertion Policies for Managing Shared Caches", in *Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 208–219.

[15] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration", in *Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[16] "International Technology Roadmap for Semiconductors", http://-www.spec.org.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and More Flexible Program Analysis", *Journal of Instruction Level Parallelism*, vol. 7, 2005.