Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing

Saurabh Gupta
Oak Ridge National Laboratory
Oak Ridge, USA
guptas1@ornl.gov

Huiyang Zhou
North Carolina State University
Raleigh, USA
hzhou@ncsu.edu

Abstract— **In modern multi-core processors, last-level caches (LLCs) are typically shared among multiple cores. Previous works have shown that such sharing is beneficial as different workloads have different needs for cache capacity, and logical partitioning of capacity can improve system performance. However, what is missing in previous works on partitioning shared LLCs is that the heterogeneity in spatial locality among workloads has not been explored. In other words, all the cores use the same block/line size in shared LLCs. In this work, we highlight that exploiting spatial locality enables much more effective cache sharing. The fundamental reason is that for many memory intensive workloads, their cache capacity requirements can be drastically reduced when a large block size is employed, therefore they can effectively donate more capacity to other workloads.**

To leverage spatial locality for cache partitioning effectively, we first propose a simple yet effective mechanism to measure both spatial and temporal locality at run-time. The locality information is then used to determine both the proper block size and the capacity assigned to each workload. Our experiments show that our Spatial Locality-aware Cache Partitioning (SLCP) significantly outperforms the previous works. We also present several case studies that dissect the effectiveness of SLCP compared to the existing approaches.

Keywords – shared last level cache; cache partitioning; spatial locality; cache management; high bandwidth memory.

I. INTRODUCTION

The ease of dynamically allocating cache capacity to applications has made shared last-level caches (LLCs) a popular choice in multi-core processors. Many previous works have shown that the controlled allocation of cache capacity among multiple cores can significantly improve the system performance [3][4][8][13][14][15][17][20][22][25][29]. These works propose to partition the cache in a static or dynamic fashion to achieve high performance/throughput, and provide isolation and fairness to the applications. Majority of these schemes focus on determining an optimal capacity allocated to each thread/processor within the constraint of an overall LLC capacity. The goal is to improve performance by allocating the capacity to a thread that benefits most from the extra capacity. What is missing in previous works is that they only explore the different capacity needs of different workloads, and do not take into account the impact due to different cache block/line sizes. Rather than controlling capacity allocation alone, we argue that we should consider both the capacity and the block size so that the shared LLC can provide heterogeneous organization, i.e., both the block size and capacity, for different workloads. Although the impact from the block size has been recognized in single core designs [2][11][24], we exploit the following fundamental observation in multi-core processors: for many memory-intensive workloads, increasing the block size can drastically reduce their requirement on capacity, thereby enabling them to donate more LLC capacity to others.

The goal of our proposed Spatial Locality-aware Cache Partitioning (SLCP) is to decide both the capacity, and the block size for each core in order to achieve the best overall performance. Therefore, it is a two-dimensional optimization problem rather than a one-dimensional optimization in the previous works such as Utility-based Cache Partitioning (UCP) [17] and Probabilistic Shared Cache Management (PriSM) [15]. To achieve our goal, we propose an approach to measure both spatial locality and temporal locality dynamically at runtime. The locality information is then used to determine the optimal heterogeneous organization for each core under the constraint of the overall LLC capacity.

We make the following contributions in this work:

a. We highlight the importance of spatial locality for cache partitioning by showing that exploiting spatial locality can reduce the cache capacity requirement of benchmarks.

b. We propose a simple online locality monitoring mechanism to measure spatial and temporal locality at runtime. Then, we present a novel-partitioning algorithm, SLCP, which considers both spatial and temporal locality in the optimization problem.

c. In the presence of an aggressive LLC prefetcher, our implementation of SLCP achieves 18.2% and 18.4% higher IPC throughput compared to an un-partitioned baseline 4MB LLC and 8MB LLC, for a 4-core and 8-core system, respectively. This is significantly higher than the related works we studied.

d. We demonstrate that cache partitioning and exploiting spatial locality are not orthogonal optimizations, and considering them together can be significantly better than their independent application. We show that SLCP has 8.5% better throughput compared to a system with adaptive line sizing [24] combined with the UCP partitioning scheme, which highlights the merit of the joint optimization.

e. We present a detailed case study to show (1) why a recently proposed partitioning algorithm PriSM makes suboptimal partitioning decisions compared to UCP and SLCP; and (2) that due to the relationship between the block size and the cache capacity requirement, the optimal partitions can only be obtained when the two parameters are considered together, as in our proposed SLCP approach.

The remainder of the paper is organized as follows. Section II provides motivation for this work. Section III presents related works and distinguishes our work from them. Section IV describes our proposed online locality monitoring scheme and the partitioning mechanism in detail. Section V presents our experimental methodology and Section VI discusses the experimental results. Section VII concludes the paper.

II. Motivation: Capacity Requirement vs. Line Size

In this section, we describe our key observation, that many memory-intensive applications show significantly reduced cache capacity requirement when bigger block sizes are used. This makes them effective donors of capacity and increases the effectiveness of cache partitioning. First, we present our framework for quantifying spatial locality.

In our recent work, a unified definition is proposed to quantify both temporal and spatial locality of references as a conditional probability: for any address, given the condition that it is referenced, how likely an address within its neighborhood will be accessed in the near future [6]. In this manner, the *Locality Score* $LS(X,Y)$ is a function of near future window size (X) and neighborhood size (Y). In the context of cache performance, the near future window size X is the reuse distance, and the neighborhood size Y is the cache line/block size. The locality score $LS(X,Y)$ provides the hit rate of a fully associative cache with the capacity of ($X*Y$) and the block size of Y. As an example, the locality plot for the level-3 (L3) cache reference stream of the benchmark *gcc* is shown in Fig. 1. The locality plot shown in Fig. 1 provides the hit rates for various L3 cache configurations. For the commonly used cache block size of 64 Bytes (B), the locality score is close to zero when the near future window size is less than 8192 (i.e., a cache capacity of 512kB = 8192*64B). This is because L2 cache is able to exploit most of the temporal locality in the access stream at shorter reuse distances, and the L3 cache has to capture the temporal locality at much longer reuse distances. Therefore, when the future window size is 16384, the locality score $LS(16384, 64)$ is 16.5%. This locality score or cache hit rate corresponds to the hit rate of a 1MB (=64B*16384) L3 cache with a block size of 64B.

In contrast, the locality score $LS(8, 512)$, corresponding to a 4kB (=512B*8) L3 cache with a cache block size of 512B, is 76.8%. This shows that a small cache, with a large cache block size, can outperform a much larger cache that has a smaller block size. This highlights the fact that the cache capacity requirements (or the working set sizes) are highly dependent on the block size. This fundamental observation is the key to our cache partitioning algorithm, and distinguishes our work from prior ones. Moreover, this example showcases that working set analysis focused on temporal locality only provides a partial picture of locality. Therefore, a unified analysis of both spatial and temporal locality is a more robust choice to drive cache partitioning.

Our locality scores show that a 4kB LLC with a cache block size of 512B outperforms a 1MB LLC with a cache block size of 64B. It should be noted that this is achieved when there are a 32kB L1 D-cache, a 256 kB L2 cache, both with a block size of 64B, and an aggressive streaming data prefetcher (see our methodology in Section V). Therefore, such a result can be somewhat unexpected. One may wonder how this is possible, and whether this happens only for very special types of accesses, like streaming.

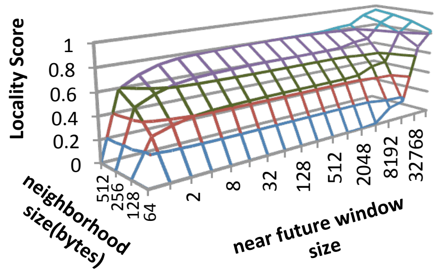

Fig. 1: The locality plot of the L3 cache access stream for *gcc*.

Here, we first use a synthetic example to show why such behavior is possible and why it happens for more generic access patterns than streaming. Consider the access sequence shown in Fig. 2a, with a block size of 64B, the (temporal) reuses have a large reuse distance N, e.g., 1 Million. Therefore, exploiting such reuses requires a very high capacity (64MB = 1M * 64B). As a result, with the baseline configuration (32kB L1, 256kB L2, and 1MB L3) having the same block size of 64B for all the cache levels, each level will have a miss rate of 100% due to their limited capacity, as shown in Fig. 2b. In contrast, if the differences between the block-addresses (i.e., block-offset bits being removed from the addresses) A and B, C and D, etc., are less than 8, a 512B L3 cache with a block size of 512B (= 8*64B) will have a miss rate of 50%. If the spatial locality exists between A and C, B and D, E and G, etc. (i.e., the block-addresses differ by less than 8), it requires two 512B-blocks which is a 1kB L3 cache to capture the spatial reuses and achieve the miss rate of 50%, as shown in Fig. 2c. From the example in Fig. 2, we can see that in a multi-level cache hierarchy, a small LLC with large block sizes can outperform a much larger LLC with small block sizes and the address patterns are not necessarily to be streaming (i.e., A, $A+\Delta$, $A+2\Delta$, $A+3\Delta$, …), which can be effectively handled with a streaming data prefetcher.

Block-address: A, B, C, D, E, F, …, A, B, C, D, E, F, … ….

$\longleftrightarrow$

Reuse distance N = 1M

(a) A block-level address trace with the block size of 64B

Baseline cache hierarchy: 32kB L1, 256kB L2, and 1MB L3, all with a block size of 64B.
Miss rate: 100% for L1, L2, and L3.

(b) Miss rates of the baseline cache hierarchy

Alt. cache hierarchy: 32kB L1, 256kB L2, both with a block size of 64B. A small L3 with a block size of 512 B.
Miss rate: 100% for L1 and L2
Miss rate: 50% for a 512B L3 if $|A\text{-}B| < 8$, $|C\text{-}D| < 8$, $|E\text{-}F| < 8$, … etc.
Miss rate: 50% for a 1kB L3 if $|A\text{-}C| < 8$, $|B\text{-}D| < 8$, $|E\text{-}G| < 8$, …, etc

(c) Miss rates of an alternative cache hierarchy

Fig. 2. An example to show a small L3 cache with a large block size outperforming a much larger L3 cache with a small block size.

Next, we validate our observation with simulation results from a set of memory-intensive benchmarks. Their performance, measured in instructions per cycle (IPC), is presented in Table I for different cache configurations (the methodology in Section V). The second column in Table I presents the performance results, including IPC and the miss rates measured using the number of misses per kilo instructions (MPKI), for a baseline 16-way set-associative 1MB LLC with a 64B block size. The third column presents low-capacity cache configurations, labeled as <capacity, block size>, determined from the locality measures for each benchmark, which provide a similar or higher locality score than the baseline. The set associativity remains as 16. The fourth column presents the performance of these cache configurations. From Table I, we can see that for these benchmarks, small-capacity caches with large block sizes can achieve similar or superior performance to large-capacity ones with the block size of 64B.

TABLE I: PERFORMANCE COMPARISON AMONG DIFFERENT CACHE CONFIGURATIONS FOR L3 CACHE (LLC)

	Baseline IPC and (MPKI)	Low-Capacity Cache Configuration	Performance of Low Capacity Cache: IPC (MPKI)
equake	0.60 (6.86)	<128kB, 512B>	0.96 (1.78)
gap	2.87 (0.24)	<128kB, 512B>	3.18 (0.06)
gromacs	1.52 (0.76)	<64kB, 256B>	1.66 (0.45)
mcf	0.19 (1.17)	<16kB, 256B>	0.20 (1.07)
mesa	2.13 (0.18)	<64kB, 256B>	2.16 (0.15)
perl	1.39 (0.62)	<128kB, 256B>	1.40 (0.62)
sphinx	0.70 (8.06)	<128kB, 256B>	1.05 (3.12)
wupwise	1.80 (0.81)	<64kB, 512B>	2.14 (0.33)

TABLE II: A CASE STUDY OF THE BENCHMARK *GCC*. (THE BLOCK ADDRESS IS BASED ON THE 64B BLOCK SIZE)

Block address	Issue cycle	<1MB, 64B> LLC	<4kB, 512B> LLC	<4MB, 64B> LLC
x416200	514782	Miss	Miss	Miss
x416201	514783	Miss	Hit*	Miss
x4161fe	515077	Miss	Miss	Miss
x4161fd	515374	Miss	Hit	Miss
x4161ff	515806	Miss	Hit	Miss
x416202	516289	Miss	Hit	Miss
…				
x416200	13525943	Miss	Miss	Hit
x416201	13525944	Miss	Hit*	Hit
x4161fe	13526238	Miss	Miss	Hit
x4161fd	13526535	Miss	Hit	Hit
…				

To further explain this behavior, we take the LLC access trace of the benchmark *gcc* as an example. A snippet of LLC access sequence and its hit/miss patterns for different cache configurations, labeled as <capacity, block size>, are presented in Table II. From Table II, it can be seen that the <1MB, 64B> baseline LLC suffers from a high miss rate for this access sequence. Although there are temporal reuses, the reuse distance is too large for the 1MB cache to capture. The 512B block size in the <4kB, 512B> LLC leverages the spatial locality and has a higher hit rate. As the access to x416201 is adjacent to the access to x416200 (only 1 cycle apart), it is merged with the prior one using miss status handling registers (MSHRs). Therefore, we mark it with a '*' in Table II to show that it is essentially a miss when the 512B block size is used. The accesses to x4161fd, x4161ff and x416202 hit in the small LLC because they are issued hundreds of cycles after the accesses to x4161fe and x416200. Note that, large blocks not only reduce cold misses, they also reduce capacity misses when the same access sequences repeat. For the <4MB, 64B> LLC, the much increased capacity is able to capture the temporal reuses.

Although the access patterns shown in Table II may be challenging for streaming data prefetchers to capture, large cache blocks can effectively exploit spatial locality. In other words, large blocks and data prefetchers are complementary in exploiting spatial locality. Here, it is noteworthy that our observation on capacity impact from spatial locality also holds for data prefetchers: a perfect prefetcher obviates the need of cache capacity and the required cache size can be arbitrarily small. In this paper, we include an aggressive streaming data prefetcher in our baseline processor model. A better prefetcher can complement our proposed approaches more by freeing up more capacity for the competing workloads.

III. RELATED WORK

Given its significant impact on system performance, there has been a significant amount of research work on cache partitioning. For shared LLCs, both coarse-grain and fine-grain cache partitioning schemes have been proposed. Coarse-grain schemes include cache-way partitioning [3][4][17][22][28][29], cache-set partitioning [18][23] or page coloring based partitioning [13]. Fine-grain schemes, such as PIPP [25], Probabilistic Shared Cache Management (PriSM) [15], Gradient-Based Cache Partitioning (GBCP) [8], and Vantage [20] can partition the cache at a fine granularity up to cache blocks. Utility-based Cache Partitioning (UCP) [17], COOP [29], PIPP, GBCP, and PriSM use temporal locality/utility based analysis to make their partitioning decisions. Vantage provides fine-grain capacity partitioning but requires caches to have good hashing as well as high associativity to work well. Molecular Caches [23] can provide different capacity, set associativity, and block size for different cores, but they use significantly different cache architecture compared to commonly used cache designs. Different from our work, this scheme does not leverage the block size impact on the capacity needs for cache partitioning. Some other works explore fair cache sharing [14] and quality of service (QoS) [9], which is out of the scope of this work.

The commonality among all the works described above is that they take temporal reuse characteristics of cache accesses into account, and are not aware of spatial locality optimization. Our work differentiates from previous work in that (1) we leverage the trade-off between spatial and temporal locality to reduce the capacity requirement so as to effectively share the LLC among multiple cores; and (2) we show in Section VI.B that partitioning shared cache capacity is not orthogonal to exploiting spatial locality. Our experimental results and case studies show that better cache partitioning decisions can be made using our approach.

The locality measure used in our work focuses on improving cache hit rates rather than the utilization rate of sub-blocks in a large cache block. Spatial footprint [11] and spatial pattern prediction [2] are complementary to our approach, and they can be utilized to reduce the memory traffic caused by large blocks. Adaptive granularity memory systems [26][27] offer the capability of providing data at different granularities on the off-chip memory system side, which improves bandwidth utilization and system efficiency for small cache blocks. The techniques presented in this paper are also complementary to such adaptive granularity memory system proposals.

IV. Spatial Locality-driven Cache Partitioning

In Section IV-A, we describe our online locality-monitoring framework to drive the cache partitioning algorithm. Since we exploit the tradeoff between cache capacity requirements and cache block sizes, we then discuss how to efficiently realize different cache block sizes in Section IV-B. We present a key observation that simplifies our cache partitioning implementation significantly. Thereafter in Section IV-C, we discuss Spatial Locality-aware Cache Partitioning algorithm (SLCP) for shared LLCs in multi-core processors in detail.

A. Online Locality Monitors

Our online locality-monitoring framework uses a hardware approach for estimating the unified temporal and spatial locality in cache access streams that drives the SLCP scheme. The conditional probability based locality measure inherently assumes a fully-associative tag array. A naïve way to estimate the locality measure is to maintain a fully associative structure for each capacity and block size combination. This is clearly impractical for hardware implementation. Therefore, we make the following two simplifications to make online locality monitoring feasible:

(1) Rather than a fully associative structure of tags; we use set-associative structures. The set associativity is the same as the baseline cache so as to facilitate way-partitioning used in our partitioning scheme (in Section IV-C). If we assume uniform accesses across all the sets, this method does not lead to any significant error in locality computation. This holds true for most of the benchmarks in our study. The index function can also be manipulated to achieve such uniform distribution.

(2) The amount of tag storage required is substantial if we maintain tags for all cache blocks in the cache. Therefore, we use set sampling [17] to mitigate the hardware overhead of online locality monitoring.

With these two simplifications, we propose the online locality monitors as shown in Fig. 2. To capture the locality information of the access stream of a cache, we introduce a few sets of auxiliary tag directories (ATD) [17], and locality score (LScore) counters. As the locality scores for one cache block size cannot be used to derive the locality scores for other cache block sizes, multiple ATDs and LScore counters are employed. A set of ATDs and LSscore counters are used for each cache block size. This is a key difference between our approach and UCP. All the ATDs have the same number of ways as in the original cache so as to capture the locality information for different cache capacity. As shown in Fig. 2, ATD-64 is used for our baseline LLC with a 64B block size and ATD-128/ATD-256/ATD-512 maintains the tags at the 128B/256B/512B cache block granularity.

With set sampling, the ATDs monitor the address stream to the sampled sets in the baseline LLC. It is important to ensure that all ATDs belonging to a core monitor the same address stream, which constrains the beginning index of the sampled set to be a multiple of 8 in this case. The reason is that the address stream to a set (e.g. set 1) in ATD-512 should be the same as the access stream to 8 consecutive sets (i.e. sets 8-15) in the baseline cache/ATD-64.

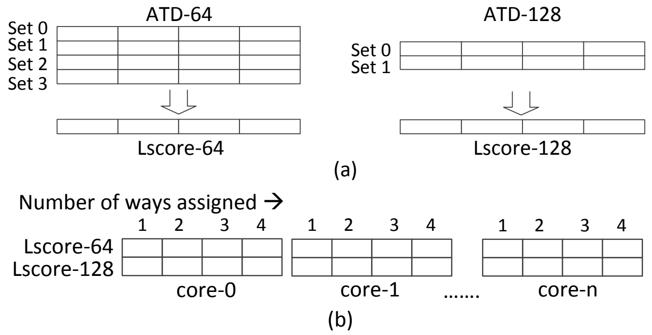

Fig. 2. Organization of (a) ATD (ATD-64 and ATD-128) and Locality Score counters (LScore) for locality computation (b) 2D-array of LScore counters per core (only showing for 64B and 128B block sizes).

The LScore counters record the number of hits in the ATDs. ATD-64 updates LScore-64; ATD-128 updates LScore-128; and so on, as shown in Fig. 2a. Taking the LScore-64 counters as an example, its first counter maintains the number of hits if the MRU (most recently used) position is assigned to a thread (i.e., assigned 1 cache way or 1/α of cache capacity, where α is the associativity of the LLC). Its second counter maintains the number of hits when both the MRU position and the MRU−1 position (i.e., 2 cache ways or 2/α of cache capacity) are assigned to a thread. Therefore, for each hit at the LRU stack position μ, the LScore counters at positions μ through α are incremented. In this manner, the LScore counters provide the cache hit information when varying numbers of ways and various block sizes are assigned. To obtain a normalized locality score (or a hit rate), the LScore counter can be divided by the number of accesses to the sampled sets. For the notation purpose, a $LScore(L,K)$ refers to the L^{th} entry in the LScore-K counter and it maintains the number of cache hits for the cache block size of K and the capacity of $L*(C_0/\alpha)$, where C_0 is the baseline cache capacity and (C_0/α) is the capacity of one cache way. Fig. 2b shows that for a multi-core system, there is a set of Lscore counters per core to monitor the miss stream of each core's private L2 cache (or the per-core access stream to the shared LLC).

In our experiments, we observed that online locality monitors estimate the locality accurately even when we sample 32 sets out of 4096 sets in our baseline LLC. As a result, the online locality monitors have low hardware overhead. In our design, we divide the 32 sampled sets into two groups of 16 sets and the two groups were randomly placed for sampling. For a 16-way cache, we need (32*16 + 16*16 + 8*16 + 4*16) = 960 ATD entries per core to monitor the block sizes of 64B, 128B, 256B, and 512B. Each ATD entry is up to 37 bits (= up to 32-bit tag + 1 valid bit + 4 LRU bits). Therefore, the storage cost of ATDs is 4.34kB per core. We use 32-bit LScore counters and the cost of LScore counter array is (16*4)*4B = 256B. Therefore, the online locality monitor has an overall storage overhead of 4.59kB per core. For a 4-core system with 4MB LLC or an 8-core system with 8MB LLC, the overhead of locality monitoring is merely 0.45% of the LLC capacity. The timing of the locality monitors are not on the critical path and they do not require low-latency accesses as the monitoring information is only used once for a program phase or epoch (e.g., every 1M cycles).

B. *Dynamically Adjusting Cache Block Sizes*

We observe that using next/previous n-line prefetching to emulate large block sizes is preferred to having specific physical designs for large block sizes. In Table III, we report the total area, access energy and leakage power of a 16-way set-associative 4MB LLC using 32nm technology for different cache block sizes, which are computed using CACTI 6.0 [16]. Clearly, using a smaller cache block size is better for the total area and leakage power considerations. From the read access energy standpoint, accessing a 128B block is slightly cheaper than accessing two 64B blocks, while accessing a 256B and 512B block is more expensive than accessing four or eight 64B blocks, respectively. Therefore, we choose to emulate cache block sizing instead of changing the physical design to have bigger block sizes.

TABLE III: CACHE PARAMETERS FOR A 4MB, 16-WAY SET-ASSOC. LLC

Cache block size (byte)	Total area (mm^2)	Total read dynamic energy per read port(nJ)	Total standby leakage power per bank (W)
64	10.76	0.84	0.93
128	15.86	1.50	0.95
256	29.00	5.58	1.37
512	79.21	21.07	2.84

When the appropriate block size is determined to be larger than the baseline block size by the partitioning algorithm, the cache controller will fetch *n* blocks, where *n* is the ratio of the large block size/the baseline block size, with the missing block being prioritized. Depending on the missing address' offset within the large block, previous lines and/or next lines are prefetched to emulate the bigger block size. The granularity of replacement is unchanged and we still maintain replacement information (e.g. LRU bits) at baseline cache block size granularity (i.e. 64-byte here).

C. *Spatial Locality-aware Cache Partitioning for Shared LLC in Multicore Processors*

In this section, we elaborate on our spatial locality aware cache partitioning (SLCP) algorithm to effectively share the LLCs. As discussed in Section II, with large cache block sizes, applications can achieve similar or better cache hit rates with reduced capacity. This enables these benchmarks to donate more LLC capacity to others benchmarks compared to when they use the baseline block size. In other words, the benchmarks with strong spatial locality become effective donors of capacity while the benchmarks with good temporal locality can utilize the extra cache capacity to improve their performance.

The goal of SLCP algorithm is to decide both the capacity and the block size for each core so as to achieve the best overall performance. Therefore, it is a two-dimensional optimization problem. In comparison, the previous works such as UCP [17] and PriSM [15] apply a one-dimensional optimization as they do not consider block sizes. Here, we leverage our online locality monitors. Fig. 3 shows a high level organization of processors and cache hierarchy in multi-core architecture where we add the online locality monitor hardware (i.e., ATDs and Lscore counters) and the logic for running the partitioning algorithm, which creates one partition per core and assigns the capacity and block size to each partition.

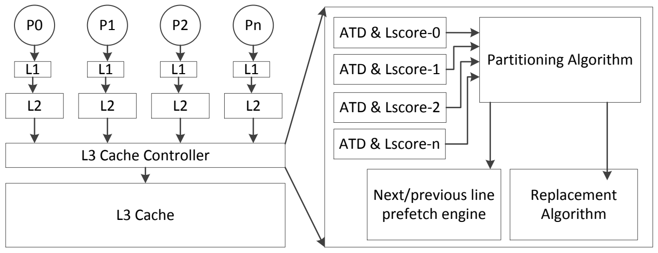

Fig. 3. The SLCP Architecture for LLCs

In the SLCP algorithm, we choose to maximize the weighted sum of locality scores across all the co-scheduled benchmarks. This is because locality scores are indicative of hit rates and maximizing hit rates improves the system throughput. The weights here are the LLC access rate of each core. Hence, per-core LScore counters can be simply added to get the metric. In this manner, the goal of our partitioning algorithm is to choose a point from each 2-D array of LScore counters so that the weighted sum of locality scores is maximized under the constraint of the overall cache capacity.

An exhaustive search for this optimization problem, by picking one point from each LScore counter array and trying all the possible combinations, is clearly impractical. Instead, in our approach, we reduce a two-dimensional LScore array (L, K) into a single-dimension LScore vector (L) to be able to leverage the *lookahead algorithm* [17]. In other words, for each co-scheduled workload, we first determine the proper block size for every possible capacity assignment. Then, using the lookahead algorithm, we determine the capacity assignment and corresponding block size. The details are as follows:

First, we employ a weight function to scale the LScore counters to account for extra memory traffic incurred by large blocks. The purpose of such scaling is to select a bigger block size only when the locality score improvement is significant. For a cache block size of K bytes, the weight function $W(K)$ is defined as $\{1 - \frac{\log2(K) - \log2(K0)}{\beta}\}$, where $K0$ is the baseline block size and β is a design parameter. The parameter β adjusts the improvement in locality scores based on block sizes. For example, comparing β=8 vs. β=16, the weight function, W(128B) is equal to 7/8 vs. 15/16 respectively (assuming a baseline block size of 64B). Second, for each capacity assignment, we record the block size that has the highest scaled LScore counter value. In this manner, a two-dimensional LScore array is reduced to single-dimensional LScore vector with each LScore annotated with its associated block size.

After obtaining one scaled LScore vector per core from the previous step, the *lookahead algorithm* is used to obtain the capacity for each core based on the LScore vectors. The *lookahead algorithm* starts with an assumption that each core will be assigned at least one way in the cache. Further capacity assignment is based on increment in locality per unit capacity (analogous to the marginal utility [17] in UCP algorithm). The core with the maximal increment in locality per unit capacity is assigned the required capacity. After an iteration of capacity assignment, at least one cache way is assigned to one of the cores/threads. This ensures that partitioning algorithm terminates within *'set associativity − number of cores'* steps. After the assignment of all cache ways to different cores, the cache block size assignment is determined by the annotated block size in the Lscore vector entry.

Realizing such heterogeneous cache partitions requires little hardware change to the regular set-associative cache structure as shown in previous works [17][22]. The replacement policy is modified to maintain a core-id (or thread-id) with each cache block specifying the core responsible for the fill. Upon a miss, the replacement algorithm first determines if the number of blocks that belong to the core requesting the fill is less than the assigned partition size. If so, the LRU block among other cores/threads' blocks is chosen as a victim. Otherwise, the LRU block of the missing core/thread will be replaced.

The partitioning algorithm is invoked to determine the partition configurations after a fixed amount of time/cycles (e.g. every 1M cycles or 5M cycles). To prioritize the recent behavior over the past behavior without losing the history completely, we right shift all the LScore and access counters once, rather than resetting them, after new partitions are created. This is a simple and effective operation used in previous works [17].

As we scale to many-core processors, one ATD per core may not be a scalable solution. We propose to employ a time-sharing scheme to sample locality of multiple cores using the same set of ATDs. The evaluation of such a scheme is not included due to space limitation.

V. Experimental Methodology

We conduct our experiments using an in-house execution-driven simulator. It uses SimpleScalar [1] as the functional simulator, while the timing model is completely revamped to model both the processor cores and memory system in detail. Each core has an out-of-order issue, 4-way superscalar pipeline with a 64-entry active list. Our baseline memory hierarchy configuration has three levels of caches, private L1 and L2 caches, and a shared non-inclusive L3 cache. Detailed configuration is shown in Table IV. Both L1 and L2 caches use 1-bit NRU replacement policy (LRU is approximated by 1-bit NRU commonly in modern high performance processors [10]). Our baseline system has no partitioning scheme deployed for the LLC and the LLC uses the LRU replacement policy. The baseline configuration also employs a stride-based stream buffer prefetcher [21] for each core. The prefetcher observes the L1 miss-stream and prefetches the data in the L2 and L3 cache. We find that beyond the prefetch-degree of 4, the overall performance improvement of the system is saturated. We model a detailed main memory system and off-chip bandwidth using the DRAMsim2 framework [19]. Address mapping is chosen to map the contiguous cache blocks across different channels for higher parallelism. We include all the benchmarks that we were able to compile for PISA and run successfully in the SPEC2000 and SPEC2006 benchmark suites for our evaluation, 16 from SPEC2000 and 8 from SPEC2006. We use the reference inputs to determine the representative phases of 100M instructions using SimPoint [7] for each benchmark. The multiprogrammed workloads for evaluating SLCP are created by mixing these benchmark phases. We categorize these benchmark phases into two groups: high MPKI (Misses per Kilo Instructions) and low MPKI benchmarks. Among 24 benchmarks, 9 are found to be high MPKI (MPKI > 5) and the other 15 fall in the category of low MPKI benchmarks. Note that we mix the benchmarks based on their memory intensiveness. We do not distinguish them based on spatial locality behavior, and maintain fairness in creating representative multi-programmed workloads.

TABLE IV: Baseline Memory Hierarchy Configuration

L1 I-cache	32kB, 64 byte block, 2-way, 1-cycle, private
L1 D-cache	32kB, 64 byte block, 4-way, 1-cycle, private
Prefetcher	16 eight-entry stream buffers with a PC-based 4-way 1024-entry stride prediction table, prefetch degree of 4
L2 Cache	256kB, 64 byte block, 8-way, 10-cycle, private
L3 Cache	1MB per core, 64 byte block, 16-way, 30-cycle, shared
DRAM	DDR3-1600 system with one rank and 8 banks, 4kB row-buffer, 6.4GB/s per channel, open-row policy FR-FCFS

We create the following five categories of 4-way multi-programmed workloads: 4H, 3H1L, 2H2L, 1H3L and 4L. The 4H category has 4 benchmarks with high MPKI, and the 3H1L category has 3 benchmarks with high MPKI and 1 benchmark with low MPKI and so on. In each of these categories, eight mixes are created by randomly picking benchmarks (i.e., a total of 40 workloads). Similar methodology is adapted for creating 8-way multi-programmed workloads (i.e. categories 8H, 6H2L, 4H4L, 2H6L). Statistics are collected for the 100M-instruction Simpoint phase only, though contention for shared cache is maintained as other faster benchmarks continue to run till slowest benchmark is finished. We do not include multithreaded benchmarks (e.g. Splash-2 [30]) for our evaluation because they do not exhibit significant heterogeneity in their locality behavior among threads.

We use the instruction per cycle (IPC) throughput and weighted speedup (WS) [5] metric for comparing the throughput and overall performance of the system:

$$\text{(a) IPC thoughput} = \sum \text{IPCshared[i]} \qquad \text{(b) WS} = \Sigma\left(\frac{\text{IPCshared[i]}}{\text{IPCalone[i]}}\right)$$

VI. Experimental Results

In this section, we first compare the SLCP performance with the baseline and three closely related works. Then, we show that a naïve combination of leveraging spatial locality and cache partitioning is sub-optimal, and SLCP outperforms them significantly. Next, we present a case study highlighting why SLCP makes better partitioning decisions and achieves higher performance improvements. We also discuss how SLCP impacts the memory bandwidth and energy consumption compared to the baseline.

A. Performance Improvements of SLCP

In this section, we compare SLCP with prior cache partitioning schemes UCP and PriSM, as well as the adaptive line sizing (ALS) scheme by Veidenbaum et al. [24]. For our 4-way multiprogrammed workloads, Fig. 4a and 4b show the normalized IPC throughput and weighted speedup (WS), respectively, when different schemes are applied to the LLC. For PriSM, we use the PriSM-H [15], because it focuses on maximizing hits and system throughput. For brevity, we refer to PriSM-H as PriSM. We implement both UCP and PriSM in our simulation framework. Both UCP and PriSM deploy 1 sampled set per 32 sets for dynamic set sampling. All other algorithm-specific parameters of these two algorithms are used as described in the previous works. We use ALS with an aggressive block sizing by setting its increase-threshold at 0.5, and the decrease threshold at 0.7. SLCP and ALS explore the same range of block sizes (i.e. 64B/128B/256B/512B) to provide a fair comparison. We use β=16 for scaling the LScore counters array to obtain the LScore vectors. The *lookahead algorithm* is applied every 5M cycles in UCP and SLCP and the partitions are reconfigured. We also assume that the online locality monitoring hardware and cache partitioning scheme do not affect the cycle/access time of the LLC because they are off the critical path of cache accesses.

Our simulation results shown in Fig. 4 show that UCP can achieve a 7.2% (6.9%) IPC throughput (weighted speedup) improvement while PriSM does not improve performance overall. ALS shows an improvement of 8.1% (9.1%) on average. UCP and PriSM focus on temporal locality for partitioning the capacity to maximize LLC hits while ALS only exploits spatial locality. ALS enjoys reduced contention for LLC capacity, even in the absence of any partitioning scheme, due to the reduced capacity requirements of benchmarks. This can be specially seen for categories 2H2L, 1H3L and 4L where ALS outperforms UCP. For memory-intensive categories 4H and 3H1L, the memory bandwidth limits the performance of ALS, and UCP performs better.

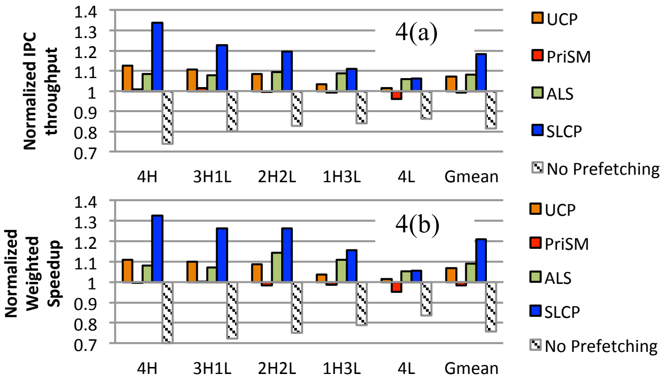

Fig. 4. Normalized IPC throughput and weighted speedup of UCP, ALS, PriSM, SLCP and baseline with 'No prefetching' for 4-way multiprogrammed workloads.

In comparison, SLCP leverages spatial locality to determine the appropriate configuration (both capacity and block size) for each workload and shows an average of 18.2% (20.9%) performance improvement. This highlights the importance of spatial locality-aware partitioning. The performance improvements from SLCP are due to two key factors. First, SLCP applies joint optimization, and assigns less capacity to the benchmarks that can exploit spatial locality in limited cache capacity. The capacity partitioning prevents the prefetched blocks from interfering the co-scheduled applications. Second, as mentioned in Section II, for the benchmarks whose access behavior has strong spatial locality but hard to detect stride patterns, large blocks are effective in complementing data prefetchers. We also show results of our baseline system with no prefetching which has 19% less IPC throughput, indicating that the prefetcher in our baseline already aggressively exploits spatial locality.

From Fig. 4, it can be observed that the multi-programmed workloads with a higher number of high MPKI benchmarks tend to improve more with SLCP (e.g., 4H or 3H1L). There are two main reasons. The first is that high MPKI benchmarks compete for cache capacity and they tend to thrash others' data, resulting in lower performance in the baseline. The second is that many high MPKI benchmarks show good spatial locality (e.g., listed in Table I) and their working sets can be significantly reduced by using large cache block sizes. Therefore, these workloads enjoy significantly higher hit rates as well as IPC improvements while at the same time others can benefit from extra capacity donated by these benchmarks.

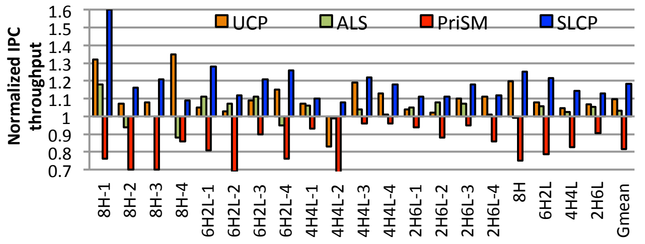

Fig. 5. Normalized IPC throughput of UCP, ALS, PriSM and SLCP relative to the baseline for 8-way multiprogrammed workloads and geometric-mean summary for each category.

For an 8-core system, Fig. 5 shows the comparison of normalized IPC throughput of SLCP compared with that of UCP, ALS and PriSM. On average, UCP achieves a good performance improvement of 9.6% while the impact of ALS on performance is limited in this case (3.2% improvement). Similar to 4-core results, PriSM does not show performance benefit for most of the workloads and suffers an average throughput degradation of over 18%. Unlike the 4-core system, the 8-core system performs better with UCP than ALS on average. This is because 8-core system is much more bandwidth restricted compared to a 4-core system (recall both systems have a 2-channel DRAM memory system), and has less room for aggressive block sizing from ALS. In this scenario, optimizing on temporal locality using UCP is more helpful due to bandwidth constraints. This indicates that with more programs sharing an LLC, capacity partitioning becomes even more important for better performance. Nevertheless, SLCP shows significantly better performance than both techniques and achieves 18.4% improvement in IPC throughput on average. In the following section, we address whether SLCP's spatial locality aware partitioning has merit over simply combining the cache partitioning (e.g. UCP) and adaptive cache block sizing (e.g. ALS).

B. *Is Exploiting Spatial Locality Orthogonal to Cache Partitioning?*

In this section, we address a key question that whether including spatial locality into cache partitioning has any merit? In other words, is joint optimization of spatial locality and cache partitioning significantly better than optimizing them independently? In the results discussed previously, UCP does not employ any dynamic block sizing scheme and rely on LLC prefetchers to exploit the spatial locality in the benchmarks. In this experiment, we use UCP augmented with statically fixed large block sizes as well as dynamic block sizing (ALS), and compare the performance with SLCP. The results are presented in Fig. 6.

In Fig. 6a, we show the geometric mean of improvement in IPC throughput for each workload category as well as geometric mean across all 40 multi-programmed benchmarks. When UCP is used in conjunction with bigger block sizes such as 128-byte, 256-byte or 512-byte, it leads to significant performance improvements in some cases showing the importance of exploiting spatial locality in combination with cache partitioning. We also observe that UCP+256B is the best performing static design on average as 512B block size saturates the off-chip bandwidth for most workloads. The 4L category has all low MPKI benchmarks, resulting in lower performance impact.

UCP and ALS improve IPC throughput by 7.2% and 8.1% respectively. We observe that the combination of UCP and ALS is able to achieve higher performance than either of them in most cases, indicating some potential in combining the two techniques. However, for the 1H3L and 4L categories, the combination of UCP and ALS degrades the performance (both throughput and weighted speedup). On average, the combination of UCP and ALS shows 8.9% higher IPC throughput than the baseline. Note that, this is significantly less than what a combination of these two techniques would achieve if they were truly orthogonal optimizations (i.e., $1.072*1.081 \approx 1.159$ or 15.9% improvement). Our proposed SLCP partitioning algorithm, which considers cache block sizing and capacity in a unified fashion for cache partitioning, achieves the highest performance in all categories, and on average provides 18.2% higher throughput than the baseline. Compared to the combined UCP and ALS, SLCP achieve 8.5% higher throughput. Therefore, we confirm that considering the two optimizations in a joint fashion can extract the full performance potential while it is not possible by considering them independently. The fundamental reason is that ALS applies line sizing without considering its impact on the capacity, while UCP applies partitioning without taking adaptive line sizing into account. With independent operation of ALS and UCP, the capacity assignment and block sizing decisions are suboptimal, as shown in detail in our case study in the Section VI-C.

Similar observations can be made from results based on weighted speedup metric (Fig. 6b), where UCP and ALS achieve 6.9% and 9.1% improvement in weighted speedups, respectively, whereas UCP+ALS only shows 8.4% improvement (i.e., slowdown from combining the two techniques compared to using ALS alone). This strengthens our argument that adaptive block sizing and cache partitioning are not orthogonal optimizations. In comparison, SLCP improves weighted speedup by 20.9%.

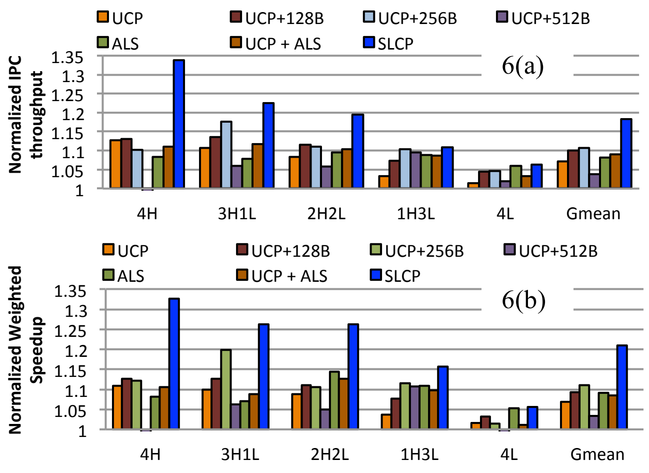

Fig. 6. Normalized IPC throughput and weighted speedup for UCP, UCP with static 128B/256B/512B cache block size, ALS, UCP+ALS, and SLCP

In summary, our experimental results show that SLCP can realize heterogeneous cache partitions, and achieve better performance than other approaches. In particular, SLCP outperforms several combinations of the cache partitioning scheme, UCP, and both static and dynamic ways to exploit spatial locality (i.e. static block sizing and ALS).

C. *Understanding Partitioning Decisions of UCP, PriSM and SLCP*

We further investigate why PriSM fails to improve performance for many of our 4-core and 8-core benchmarks, while UCP and SLCP both achieve good speedups. We illustrate the fundamental shortcoming of PriSM in the Fig. 7 using the case study on the benchmark *mcf* from SPEC2000. The figure shows the cache hit rate when the number of ways allocated to benchmark *mcf* is varied from 1 to 16 in a 4MB 16-way set-associative LLC with a 64B block size. *Mcf* exhibits thrashing behavior, as the hit rate is extremely low until the number of allocated ways is increased to 7. This curve is used successfully by UCP to assign low amount of capacity to *mcf* when other benchmarks are contending for capacity heavily. In this manner, UCP correctly estimates that there is practically no gain by assigning more than 1 way in the set (up to 6) to *mcf.* In the scenarios when UCP observes that other benchmarks are either less memory intensive or do not gain hits by extra capacity it may decide to give *mcf* almost half of the capacity because that is overall the best for LLC performance. PriSM's hit maximization scheme, however, compares the difference in the hit rate of shared execution vs. running alone and assumes a linear relationship between number of ways allocated and the hit ratio (shown as the line labeled 'PriSM' in the Fig. 7). This is clearly inaccurate for *mcf* and many other benchmarks which feature similar

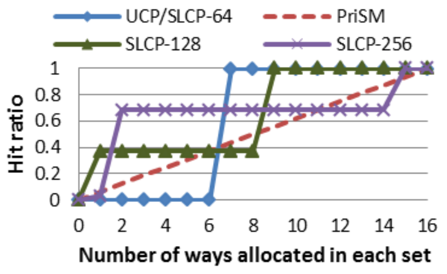

Fig. 7. Different hit rate curves used by UCP, PriSM and SLCP while allocating ways to benchmark *mcf*

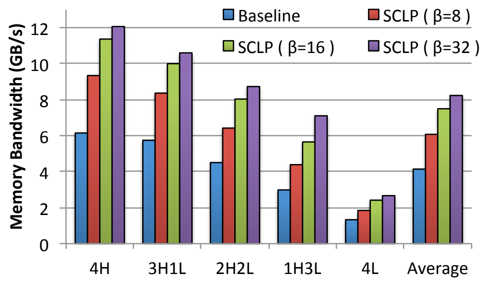

Fig. 8. Average memory bandwidth usage of SLCP compared to baseline for different workload categories

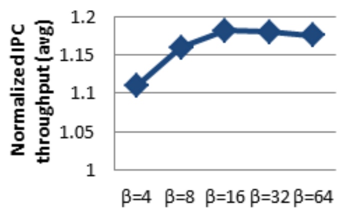

Fig. 9. Variation of average WS improvements of SLCP for different values of β parameter

thrashing behavior (or high temporal reuse distance). As a result, PriSM makes incorrect partitioning decisions for those benchmarks (assign capacity to applications where no cache hits can be gained), leading to ineffective partitioning and poor performance for the benchmarks in our study, as shown in Section VI-A.

On the other hand, SLCP solves the problem by adding a new dimension to the optimization problem. As shown in Fig. 7, SLCP-128 and SLCP-256 show significantly higher cache hit rates with smaller number of ways allocated to *mcf* when the block size is increased to 128 bytes and 256 bytes. In other words, the relationship between hit rates and allocated cache ways is highly dependent on the block size. SLCP is designed based on this fundamental observation, and it assigns 2 ways with either a 128-byte or 256-byte block size in the partition for *mcf* to achieve significantly better hit rate.

As discussed earlier in Section VI.B, the performance of a UCP combined with ALS has lower performance than SLCP. This can also be explained using Fig. 7. In the combined UCP and ALS, UCP monitors the access stream (i.e., UCP-64) and therefore decides to assign the minimum possible capacity to *mcf* (1 way in this case). However, with a single way allocated, large block sizes by ALS are not effective. The fundamental issue that UCP+ALS fails to discover is, by assigning just one extra way and bigger block sizing, it could significantly improve the hit ratio. Therefore, UCP+ALS cannot find the optimal configuration for the partition for *mcf*. In other words, due to the relationship between the block size and the capacity (i.e., the capacity requirement is highly dependent on the block size), the optimal <capacity, block size> combination can only be determined by jointly considering capacity and block sizing, as in our proposed SLCP approach, rather than considering them orthogonally.

D. SLCP Bandwidth Usage and Performance Impact of the β-Parameter

In this section, we study the impact of SLCP on memory bandwidth consumption. As described earlier, the tuning parameter β controls the aggressiveness of dynamically increasing the cache block size. In Fig. 8, we present the bandwidth consumption of the baseline and SLCP. We can clearly see that β=32 shows really high memory bandwidth consumption, very close to the maximum of 12.8 GB/s for 4H category. Lower values of β decrease the memory bandwidth pressure and a moderate amount of bandwidth usage can be observed. Overall, we see a 46% increase in average memory bandwidth for β=8, and 80% and 98% increase in memory bandwidth for β=16 and β=32, respectively. In this manner, we see that the β parameter can successfully tune the bandwidth usage for SLCP.

We also evaluate the performance impact of varying the β parameter in our SLCP algorithm. In Fig. 9, we show the variation of overall performance gains of SLCP with different values of the β parameter. As discussed in Section IV.C, a lower value of β implies more conservative cache block sizing while a higher value will make SLCP choose bigger block sizes for marginal gains. Our 4-core system has a constrained DRAM bandwidth of 12.8GB/s, and we observe that performance improvements of SLCP increase when β is increased up to 16. Further increase in β causes performance drop due to high queuing delays at the memory controller and DRAM system. Therefore, it is important to choose a suitable β parameter value in a system with limited DRAM bandwidth. For our benchmarks and memory system setup, we empirically chose β=16 as it achieves high performance with moderate increase in bandwidth usage. For 3D stacked DRAM with high bandwidth memory (HBM) [31], there is much more room to trade bandwidth usage for higher performance and a high β value would be appropriate. As these HBM modules tend to have longer latency due to lower bus speeds, there is a stronger need to trade higher bandwidth for low access latency. Moreover, the energy required for moving data between memory and LLC is also lowered significantly as the memory and the CPU are within the same chip module. These factors make SLCP a nice fit for the forthcoming HBM technology.

E. Energy Savings of SLCP

We model the power consumption of multi-core processors using McPAT [12] and the power consumption of DRAM using DRAMsim2. We observe that though the dynamic power for the cores is increased, overall energy consumption is reduced due to significantly reduced run time for the workloads. For the DRAM system, we have observed increased bandwidth utilization that can lead to increased DRAM power consumption. But, the multiprogrammed workloads suffer from low row buffer hit rates due to interference of memory access streams in the baseline. SLCP combines multiple distant data accesses into a burst of adjacent memory accesses due to large cache block sizes. This offsets the potential increase in power consumption of DRAM. As a result, the overall energy consumption for DRAM system is also reduced due to reduced execution time. Overall, SLCP reduces the system energy consumption by 12.6% on average compared to the baseline 4-core system. In comparison, UCP provides a total saving in energy consumption of 6.9% on average.

VII. Conclusions

In this paper, we make a case for spatial locality aware cache partitioning. Our approach is based on the following key observation: exploiting the spatial locality can drastically reduce the cache capacity required to sustain the same or achieve better performance for many benchmarks. Therefore, more capacity can be shared among workloads with good temporal locality. We propose an online locality-monitoring framework to drive both the LLC capacity allocation and the block size to be used for each partition. Our proposed design of spatial locality aware cache partitioning (SLCP) algorithm outperforms high performing cache partitioning algorithms including UCP and PriSM significantly in both 4-core and 8-core systems even when these schemes are combined with an adaptive line sizing scheme. Through case studies, we highlight that jointly optimizing spatial locality and cache partitioning is significantly better than independent optimizations. Moreover, our results show that SLCP can also reduce the overall energy consumption of the system, making it a suitable design choice for both performance and energy consumption standpoints.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments to improve our paper, and Prof. Fei Wang from Xian Jiaotong University and Dr. Devesh Tiwari from Oak Ridge National Lab for inspiring discussions. This research is supported by an NSF grant CCF-1216569, a research fund from Intel Corporation, and a Chinese research program "introducing talents of discipline to universities B13043". Saurabh Gupta is currently supported by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

References

[1] D. Burger and T. M. Austin. "The Simplescalar Tool Set Version 2.0.", Technical Report, *Computer Science Department, University of Wisconsin-Madison, 1997.*

[2] C. F. Chen, S-H. Yang, B. Falsafi, and A. Moshovos. "Accurate and complexity-effective spatial pattern prediction." In *Software, IEE Proceedings-, pp. 276-287. IEEE,* 2004.

[3] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. "Dynamic cache partitioning via columnization." In *Proceedings of Design Automation Conference* 2000.

[4] H. Dybdahl and Per Stenstrom. "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors." In *Proceedings of HPCA* 2007.

[5] S. Eyerman and L. Eeckhout. "System-level performance metrics for multiprogram workloads." In *Proceedings of IEEE MICRO* 2008.

[6] S. Gupta, P. Xiang, Y. Yang, and H. Zhou. "Locality principle revisited: A probability-based quantitative approach." In *Proceedings of IEEE IPDPS* 2012.

[7] G. Hamerly, E. Perelman, J. Lau, and B. Calder. "SimPoint 3.0: Faster and More Flexible Program Analysis." In *Proceedings of MoBS* 2005.

[8] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr, and J. Emer. "The gradient-based cache partitioning algorithm." *ACM Transactions on Architecture and Code Optimization (TACO)*, 8.4, 2012: 44.

[9] R. Iyer. "CQoS: a framework for enabling QoS in shared caches of CMP platforms." In *Proceedings of ICS* 2004.

[10] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. "High performance cache replacement using re-reference interval prediction (RRIP)." In *Proceedings of ISCA* 2010.

[11] S. Kumar and C. Wilkerson. "Exploiting spatial locality in data caches using spatial footprints." In *Proceedings of ISCA* 1998.

[12] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures." In *Proceedings of IEEE MICRO* 2009.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems." In *Proceedings of HPCA* 2008.

[14] S. Kim, D. Chandra, and Y. Solihin. "Fair cache sharing and partitioning in a chip multiprocessor architecture." In *Proceedings of PACT* 2004.

[15] R. Manikantan, K. Rajan, and R. Govindarajan. "Probabilistic Shared Cache Management." In *Proceedings of ISCA 2012.*

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. "Cacti 6.0: A tool to understand large caches." *Univ. of Utah and HP Lab.,* Tech. Rep (2009).

[17] 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 IEEE MICRO* 2006.

[18] P. Ranganathan, S. Adve, and N. P. Jouppi. "Reconfigurable caches and their application to media processing." In *Proceedings of ISCA* 2000.

[19] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "Dramsim2: A cycle accurate memory system simulator." *Computer Architecture Letters (CAL)* 10.1, 2011: 16-19.

[20] D. Sanchez and C. Kozyrakis. "Vantage: scalable and efficient fine-grain cache partitioning." In *Proceedings of ISCA* 2011.

[21] T. Sherwood, S. Sair, and B. Calder. “Predictor-directed stream buffers." In *Proceedings of IEEE MICRO* 2000.

[22] G. E. Suh, L. Rudolph, and S. Devadas. “Dynamic partitioning of shared cache memory." *Journal of Supercomputing*, 28(1), 2004.

[23] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. "Molecular Caches: A caching structure for dynamic creation of application specific heterogeneous cache regions." In *Proceedings of IEEE MICRO* 2006.

[24] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. "Adapting cache line size to application behavior." In *Proceedings of ICS* 1999.

[25] Y. Xie and G. Loh. "PIPP: promotion/insertion pseudo partitioning of multi-core shared caches." In Proceedings of *ISCA* 2009.

[26] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. "The dynamic granularity memory system." In *Proceedings of ISCA* 2012.

[27] D. H. Yoon, M. K. Jeong, and M. Erez. "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput." In *Proceedings of ISCA* 2011.

[28] D. Zhan, H. Jiang, S. C. Seth. "CLU: Co-optimizing Locality and Utility in Thread-Aware Capacity Management for Shared Last Level Caches." *IEEE Transactions on Computers*, 63(7), 2014.

[29] D. Zhan, H. Jiang, and S. C. Seth. "Locality & utility co-optimization for practical capacity management of shared last level caches." In *Proceedings of ICS* 2012.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations." In *Proceedings of ISCA* 1995.

[31] http://www.anandtech.com/show/9266/amd-hbm-deep-dive/3. (June 2015).