Co-optimizing Memory-Level Parallelism and Cache-Level Parallelism

Xulong Tang
The Pennsylvania State University
University Park, Pennsylvania, USA
xzt102@psu.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
University Park, Pennsylvania, USA
kandemir@cse.psu.edu

Mustafa Karakoy
TOBB University of Economics and Technology
Ankara, Turkey
m.karakoy@yahoo.co.uk

Meenakshi Arunachalam
Intel
Hillsboro, Oregon, USA
meena.arunachalam@intel.com

Abstract

Minimizing cache misses has been the traditional goal in optimizing cache performance using compiler based techniques. However, continuously increasing dataset sizes combined with large numbers of cache banks and memory banks connected using on-chip networks in emerging manycores/accelerators makes cache hit–miss latency optimization as important as cache miss rate minimization. In this paper, we propose compiler support that optimizes both the latencies of last-level cache (LLC) hits and the latencies of LLC misses. Our approach tries to achieve this goal by improving the parallelism exhibited by LLC hits and LLC misses. More specifically, it tries to maximize both cache-level parallelism (CLP) and memory-level parallelism (MLP). This paper presents different incarnations of our approach, and evaluates them using a set of 12 multithreaded applications. Our results indicate that (i) optimizing MLP first and CLP later brings, on average, 11.31% performance improvement over an approach that already minimizes the number of LLC misses, and (ii) optimizing CLP first and MLP later brings 9.43% performance improvement. In comparison, balancing MLP and CLP brings 17.32% performance improvement on average.

CCS Concepts • **Computer systems organization → Multicore architectures**.

Keywords Manycore systems, data access parallelism

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314599

ACM Reference Format:
Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meenakshi Arunachalam. 2019. Co-optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314599

1 Introduction

Compiler researchers investigated a variety of optimizations to improve cache performance [4, 9, 19, 26, 28, 32, 41, 52, 55]. Most of these optimizations are geared towards minimizing the total number of cache misses, the rationale being that, lower the misses, higher the application performance. While this is certainly true and many commercial compilers already employ a large suite of optimizations that target cache miss minimizations (e.g., loop permutation, iteration space tiling, loop fusion), the impact of these techniques is becoming increasingly limited as (i) emerging applications are processing enormous amounts of data, (ii) the increases in cache capacities are lagging far behind the increases in application data volume [33, 58], and (iii) as a result, caches are becoming unable to maintain application working sets even after aggressive cache miss minimization.

As a result, a complementary approach would be embracing cache misses and trying to reduce their latencies (in addition to their counts). Recent works [13, 20, 22–25, 37, 38, 42–44, 50, 51, 60, 61] have shown that significant amount of the overall data access latency is spent on cache miss related traffic (either as network latency to reach the last-level cache (LLC) banks/memory controllers (MCs), or as memory access itself). In other words, cache misses contribute to a large fraction of the overall data access latency. Therefore, an optimization approach that targets cache miss latencies can potentially bring significant reductions in application execution times.

Meanwhile, to enable application scalability, modern manycores are employing scalable interconnects (e.g., mesh-based network-on-chip (NoC)), instead of conventional buses.

However, such NoC-based manycores lead to *non-uniform* latencies for both LLC hits and LLC misses. Typically, a data access missing in its local private cache (e.g., L1) is routed to remote LLC bank, and if it is a miss in LLC, it will be further routed to corresponding MC for an off-chip access. In this flow, multiple simultaneous accesses to a given LLC bank further increase the access latency, even when these accesses hit in the LLC bank. This is because that (i) routing all the requests to the same node can cause network contention, and (ii) multiple requests to the same LLC/memory bank can lead to cache contention.

Therefore, it is very important for an optimizing compiler that aims to maximize the performance of data access in an NoC-based manycore to minimize the latencies of both LLC hits and LLC misses (in addition to reducing the number of LLC misses, which is a traditional optimization goal). One way of reducing the latencies of both LLC hits and LLC misses is to improve their parallelism, that is, the number of LLC banks and memory banks that are concurrently accessed in a given period of time should be maximized.

In this paper, we define cache-level parallelism (CLP) as the number of the LLC banks serving L1 misses when at least one LLC bank is serving an LLC access. Similarly, we define memory-level parallelism (MLP) as the number of memory banks serving LLC misses in parallel when at least one request is being served by a memory bank[1]. We then propose a compiler framework for reducing the latencies of both LLC hits and LLC misses, by increasing their parallelism. At a high level, this is achieved by maximizing MLP and CLP in a given period of time (i.e., execution epoch). Our contributions can be summarized as follows:

- We propose an optimization strategy that optimizes MLP for LLC misses and CLP for LLC hits *together*. Our approach employs code restructuring and computation scheduling, with the goal of reducing the latency experienced by data accesses. More specifically, for LLC hits, we want to maximize the number of cache banks concurrently accessed; and, similarly, for LLC misses, we want to maximize the number of memory banks concurrently accessed within a given period of time (in addition to the number of cache banks, as all memory accesses visit LLC banks before memory banks).
- We explain how our strategy can be used to strike a balance between MLP and CLP. Specifically, by considering the total number of accesses to each of the cache and memory banks, our compiler automatically determines the proper "trade-off" between MLP and CLP for each loop nest, that leads to the best application performance.
- We evaluate our approach using a set of 12 multithreaded applications on both a detailed manycore simulator and a commercial manycore system (Intel Knight's Landing [45]). The experimental data collected from the simulator indicate that (i) optimizing MLP first and CLP later can bring, on average, 11.31% performance improvement over an approach that already minimizes the number of LLC misses, and (ii) optimizing CLP first and MLP later can bring 9.43% improvement. Finally, balancing MLP and CLP can bring 17.32% improvement. The corresponding improvement from our approach that balances CLP and MLP on Intel manycore is 26.15%, on average.
- Using both the simulator and the commercial manycore architecture, we present a detailed comparison of our approach against two previously-published compiler optimizations [8, 36] as well as a hardware-based memory parallelism optimization [34]. The experimental results collected clearly show that our proposed approach performs better than these alternative approaches. Specifically, it performs 12.87%, 8.31% and 6.21% better, respectively, compared to [36], [8] and [34], when using the simulator. On the commercial manycore system, our approach outperforms [36] and [8] by 20.13% and 13.27%, respectively.

To our knowledge, this is the first work that presents a compiler scheme designed to *co-optimize* MLP and CLP. Further, our approach, which primarily targets "hit and miss latencies" is *complementary* to conventional data locality optimizations (that target minimizing the "number of cache misses") as well as techniques designed to increase compute level parallelism (e.g., loop parallelism).

2 Manycore Architecture

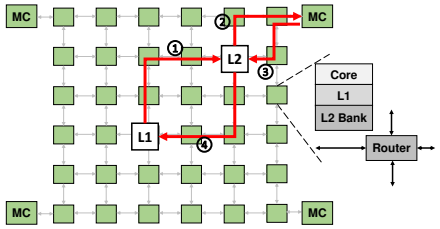


Figure 1. Network-on-chip (NoC) based manycore architecture template and the flow of a representative data access.

In this paper, we target emerging Network-on-Chip (NoC) based manycore/accelerator architectures. Figure 1 depicts a manycore architecture template. Each node in this architecture contains a core, a private L1 cache, and a shared LLC bank. The LLC is divided in to *cache banks* and shared across all cores. The LLC in our baseline architecture refers to L2 cache, though our approach can work with cache hierarchies of any depth. We assume a static-NUCA based [21] LLC management, where the LLC is partitioned into (cache) banks and banks are distributed across nodes. Each cache line is statically mapped to a particular LLC bank based on its address. Figure 1 also illustrates a typical request/data flow involved in a data access. Specifically, an access that misses the private L1 cache is directed to an LLC (L2) bank (①). If it hits in the LLC bank, the requested data is sent back to the requesting node (④). However, if it misses there, an LLC miss

[1] We use the terms "memory-level parallelism" (MLP) and "bank-level parallelism" (BLP) interchangeably.

occurs, and the request is forwarded to the corresponding MC (②). The request to a DRAM is queued in a buffer at the MC, and is issued to the DRAM by MC.

The core in each node can simultaneously issue data reference requests and those requests are received and served by LLC banks (for hits), or memory banks (for misses). Our optimization focuses on cache-level parallelism (CLP) and memory-level parallelism (MLP) in an epoch of n cycles. Typically, n can be set to a value considering the size of the ROB (reorder buffer) in the target architecture [48]. Note that, CLP captures the number of LLC banks serving L1 misses when there is at least one bank serving an L1 miss. Clearly, a higher CLP value indicates a better utilization of the LLC in the system. Similar to the CLP case, a higher MLP means better utilization of hardware resources memory banks. Each memory bank has a sense-amplifier called *row-buffer*, which is used to hold the memory row loaded. Subsequent accesses to the same row experience short latency, and are referred as row-buffer hits in an open-row policy.

In addition to a manycore simulator, we also use Intel Knight's Landing (KNL) [45]. KNL consists of 36 nodes (referred to as tiles in Intel terminology) connected through mesh on-chip network. Each tile consists of two cores where each core features two 512-bit AVX vector units (VUs). There is a 1 MB "tile-private" L2 cache shared by two cores within a tile and cache coherence is maintained among L2 caches across different tiles. KNL has a 16GB of multi-channel dynamic random access memory (MCDRAM), which is divided into 8 channels and attached to the 8 MCDRAM MCs spread across 4 corners of the mesh on-chip network. This MCDRAM, which is separate from the DDR4 memory, can be configured into one of three different modes: (i) cache mode, where MCDRAM simply acts entirely as a conventional LLC (L3); (ii) flat mode, where MCDRAM acts entirely as an addressable memory; and (iii) hybrid mode, where 25% (or 50%) of the MCDRAM capacity is configured as LLC and the rest is configured as an addressable memory. KNL also has three different cluster modes. The base mode is referred to as the "all-to-all" mode, where the addresses are spread over the caches and MCs uniformly. On the other hand, in the "quadrant" mode, the entire mesh is divided into 4 virtual regions, and a memory access travels within the same region. Finally, in the "SNC-4" mode (also known as the sub-NUMA mode), the mesh is split into 4 non-uniform memory access (NUMA) clusters. In this case, all accesses (both cache accesses and memory accesses) travel within each NUMA cluster.

3 Motivation

Let us consider the data access[2] pattern shown in Figure 2a on a two-dimensional array. We assume that the array is stored in memory in a row-major fashion (as in C language). There are two cores in the system, and each core accesses

[2]We use the terms "data access" and "array reference" interchangeably.

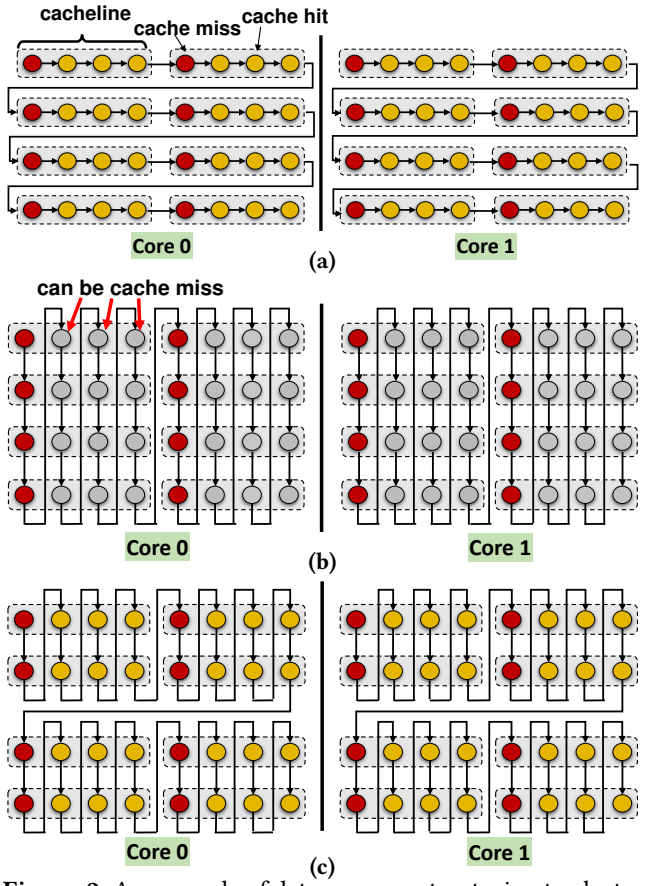


Figure 2. An example of data access restructuring to cluster cache misses. The ovals represents the array elements. The shaded rectangle box represents the cache line. The arrow denotes the reference order. (a) Default reference pattern captured by solid arrows. (b) Reference pattern after applying loop permutation. (c) Reference pattern after applying loop tiling on (b).

a 4×8 portion of the array (for illustrative purposes). Let us assume that this access pattern repeats itself in a (timing) loop (that is, after the last element of the array is accessed, the first element is accessed again, until a convergence criterion – captured by the timing loop condition – is met). The figure also highlights the cache lines (blocks) using gray boxes, each holding 4 array elements.

The access pattern in Figure 2a is very good from a data locality viewpoint, as the only misses incurred are for the accesses to the first element of each cache line (i.e., cold misses). The problem with this hit/miss pattern is that it does not exploit MLP (memory-level parallelism) well. Assuming, for example, there are 4 MCs in the system each controlling 4 banks, from a single core perspective, at a period of accessing consecutive 4 array elements, only one memory bank is accessed (by the access corresponding to the miss), leading to a total of 2 bank accesses at most when considering both cores. This is clearly much lower than the maximum possible of 16 memory banks.

One way of improving MLP is to *cluster* cache misses. In Figure 2b, each core traverses its portion of the array in a *column-major fashion*, instead of the *row-major fashion* (shown in Figure 2a). This new traversal order clusters the cache misses as they are now accessed in bursts. Therefore, at the initial period of 4 data accesses, each core accesses 4 banks (assuming each request/miss goes to a different bank), resulting in 8 banks being accessed when two cores are considered. However, now, one can expect additional cache misses since the new access pattern (which is column-wise) does not align with the underlying row-major layout of the array (in fact, in the worst case, after this transformations, all hits in Figure 2a can get converted into misses in Figure 2b). In other words, by going from Figure 2a to Figure 2b, we improve MLP but distort cache locality. However, this can be fixed, as suggested by [36], by tiling/strip-mining the innermost loop. The new post-tiling access pattern is illustrated in Figure 2c. Thus, after the back-to-back optimizations of loop permutation and tiling, we have the cache misses *clustered* and, at the same time, we maintain the original cache locality (number of cache hits).

Similar to optimizing MLP for LLC misses (red ovals in Figure 2), we can also optimize CLP for LLC hits (yellow ovals in Figure 2). There are two benefits of considering CLP together with MLP. First, a higher CLP indicates better utilization of LLC banks, and consequently reduces the LLC hits latency by overlapping (in time) different LLC accesses. And, second, a higher CLP means that requests are spread across different nodes in the network more uniformly. This potentially reduces the previously mentioned non-uniformity of cache hit latencies, and also better balances the utilizations of network links and routers (i.e., reduces network contentions as well). In this paper, we explore three optimizations: *MLP-first*, *CLP-first*, and *Balanced*. In MLP-first, we first optimize MLP, then CLP is optimized without distorting optimized MLP. Alternately, in CLP-first, we optimize CLP as the primary target and MLP as the secondary. Finally, in Balanced, we try to strike a balance between MLP and CLP. The following discussion focuses mainly on MLP-first. The CLP-first is quite similar and therefore we omit its detailed discussion. We discuss Balanced in Section 5.6.

4 High-Level Overview of Our Approach

It is to be noted that, clustering misses may not necessarily guarantee high MLP. This is because it is possible that the clustered misses still access only few memory banks. Motivated by this observation, we propose a loop iteration scheduling strategy where the clustered misses provide the maximum values of MLP from both the *inter-core* and the *intra-core* perspectives. Specifically, LLC misses issued within tiles across different cores (inter-core), and LLC misses clustered within a tile of a given core (intra-core) access as many different memory banks as possible.

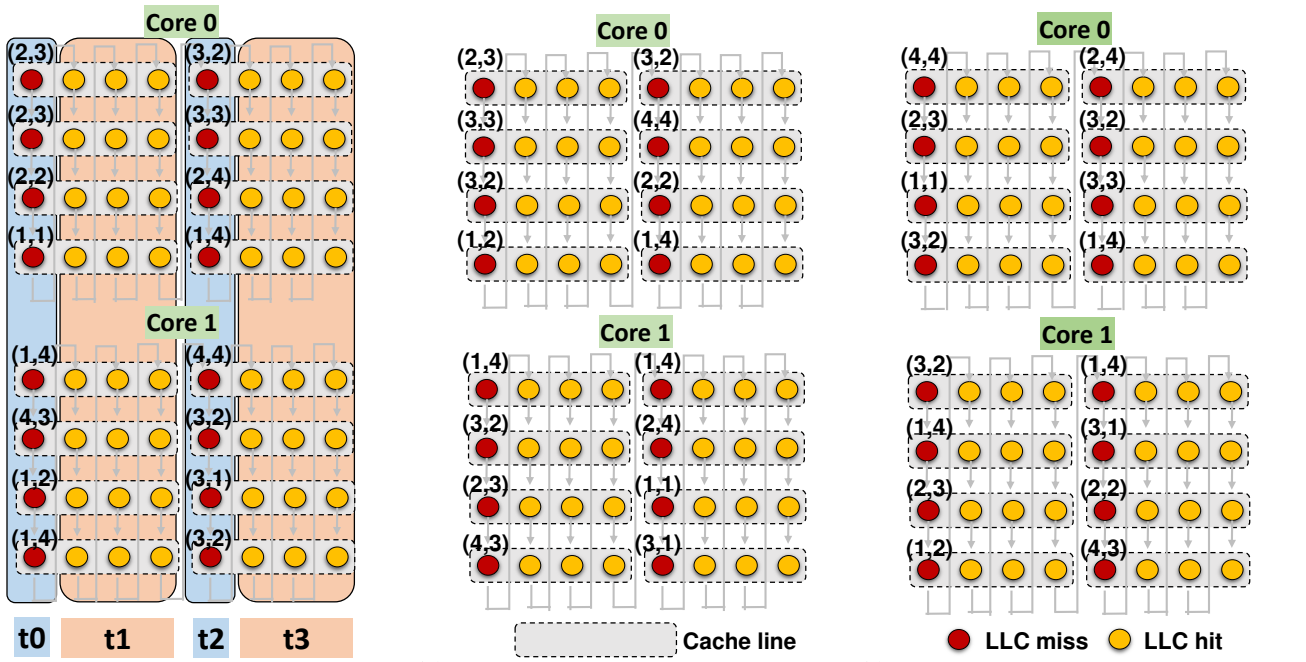
Consider the example in Figure 3, which shows the array access order after the loop permutation and strip-mining (Section 3). For simplicity, let us assume that there are 4 memory banks and 4 LLC banks in the system. For a given cache line (denoted as gray box), the corresponding memory bank ID and LLC bank ID are labeled as a pair of number on top of the first data element in a given cache line.

For explanation purposes, in Figure 3a, we divide the scheduling process into 4 phases (labeled as $t0$ to $t3$).

In this example, among the total 16 cache lines, there are 5 in memory bank-1, 4 in memory bank-2, 5 in memory bank-3, and 2 in memory bank-4. In MLP-first, our focus is on the distinct memory banks accessed by LLC misses (red ovals). Figure 3a depicts the default array accesses order *without* our optimization. In phase $t0$, LLC misses from $core_0$ access two different memory banks (bank-2 and bank-1). Therefore, the intra-core MLP for $core_0$ is 2. In phase $t2$, the misses from $core_0$ access bank-3, bank-2, and bank-1, resulting in an intra-core MLP value of 3. Hence, we can denote the intra-core MLP of $core_0$ as MLP_{core0} = {2,3}. Similarly, we have intra-core MLP of $core_1$ as MLP_{core1} = {2,2}. Compared to intra-core MLP, inter-core MLP considers concurrent data accesses from different cores within the same execution phase. Specifically, in phase $t0$, the misses from $core_0$ and $core_1$ access bank-1, bank-2, and bank-4. As a result, we have MLP_{t0} = {3}. Similarly, we calculate MLP_{t2} = {4} at phase $t2$.

We now try to optimize both intra-core and inter-core MLP. Figure 3b shows the new array reference order after iteration scheduling. Note that the new array referencing order is generated because of reordering the execution order of loop iterations, *not* data layout transformation. Using the same MLP calculation discussed above, the intra-core MLPs of the new array reference order are MLP_{core0}={3,4} and MLP_{core1}={4,3}. Similarly, the inter-core MLPs are MLP_{t0}={4} and MLP_{t2}={4}. As one can observe, the new loop iteration order in Figure 3b improves *both* intra-core MLP and inter-core MLP compared to the execution in Figure 3a.

An interesting observation is that two different iteration schedules could have exactly the *same* MLP values. For example, Figure 3c has the same intra-core MLP and inter-core MLP compared to Figure 3b, with a different loop iteration execution order. This potential allows us to optimize CLP for cache hits (denoted as yellow ovals in Figure 3), *without compromising MLP*. In Figure 3a, the second number in the pair denotes the LLC bank ID. In total, there are 2 accesses to LLC bank-1, 5 accesses to LLC bank-2, 4 accesses to LLC bank-3, and 5 accesses to LLC bank-4. All the three hits (yellow ovals) in a cache line access the same LLC bank. In the default loop execution order (Figure 3a), cache hits from $core_0$ access LLC bank-3, LLC bank-2, and LLC bank-1, giving an intra-core CLP of 3 in phase $t1$. Similarly, cache hits access LLC bank-2, LLC bank-3, and LLC bank-4 in phase $t3$. As a result, the intra-core CLP for $core_0$ is CLP_{core0}={3,3}. We apply the same calculation for $core_1$ and obtain CLP_{core1}={3,3}. We can also



(a) Default: Intra-core: MLP_{core0}={2,3}, MLP_{core1}={2,2}, CLP_{core0}={3,3}, CLP_{core1}={3,3}. Inter-core: MLP_{t0}={3}, MLP_{t2}={4}, CLP_{t1}={4}, CLP_{t3}={4}.

(b) MLP only: Intra-core: MLP_{core0}={3,4}, MLP_{core1}={4,3}, CLP_{core0}={2,2}, CLP_{core1}={3,2}.Inter-core: MLP_{t0}={4}, MLP_{t2}={4}, CLP_{t1}={3}, CLP_{t3}={3}.

(c) MLP-first: Intra-core: MLP_{core0}={4,3}, MLP_{core1}={3,4}, CLP_{core0}={4,3}, CLP_{core1}={3,4}. Inter-core: MLP_{t0}={4}, MLP_{t2}={4}, CLP_{t1}={4}, CLP_{t3}={4}.

Figure 3. Clustering array references to improve both intra-core MLP/CLP and inter-core MLP/CLP. The read oval in each cache line represents LLC miss, whereas the subsequent yellow ovals represent LLC hits. Each cache line is associated with a memory bank and an LLC bank. The pair (memory bank, LLC bank) above a cache line indicates the corresponding memory bank ID and LLC bank ID.

calculate the inter-core CLP of Figure 3a. Cache hits from $core_0$ and $core_1$ access all four LLC banks in phases $t1$ and $t3$, respectively, resulting in inter-core CLPs as CLP_{t1}={4} and CLP_{t3}={4}.

Note that, while Figure 3b gives us the maximized MLP, CLP is not optimized. In fact, the CLP in Figure 3b is even worse compared to the CLP in Figure 3a. Specifically, in Figure 3b, the intra-core CLPs are CLP_{core0}={2,2} and CLP_{core1}={3,2}, and the inter-core CLPs are CLP_{t1}={3} and CLP_{t3}={3}, which are lower compared to Figure 3a.

Finally, in Figure 3c, we take the optimization of CLP into account while performing our loop iteration scheduling. This gives us the optimized MLP as well as the optimized CLP. Specifically, in this case, we have intra-core CLP_{core0}={4,3}, CLP_{core1}={3,4}, and inter-core CLP_{t1}={4}, CLP_{t3}={4}. It should be emphasized that, in our discussion so far, we have mainly focused on the MLP-first approach.

5 Details of the Optimizations

5.1 Formalization

We now define four important concepts employed by our compiler framework: *iteration block (IB)*, *iteration window (IW)*, *data block (DB)*, and *data set (DS)*. Among these four concepts, IB and IW are defined on *iteration space*, whereas DB and DS are defined on *data space*. The iteration space of an m-level nested loop can be represented by an m-dimensional vector $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, delimited by loop bounds $\{(l_1, u_1), (l_2, u_2), \cdots, (l_m, u_m)\}$, where $l_k \leq i_k \leq u_k$ and $1 \leq k \leq m$. Each loop iteration is represented using an iteration vector $\vec{i}$. Similarly, the data space for an n-dimensional array can be represented by an n-dimensional vector $\vec{j} = (j_1, j_2, \cdots, j_n)^T$ where j_k $(1 \leq k \leq n)$ is the index of array element. Each array reference is represented by a mapping from iteration space to data space. Given a loop iteration vector $\vec{i}$, the corresponding array reference (i.e., array index) is $\vec{r} = A\vec{i} + \vec{o}$, where A is the reference matrix and $\vec{o}$ is the reference offset[3]. For example, the reference to array $A[i_1 + i_2][i_2 + 2]$ in two-level nested loop is represented as $\vec{r} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, where $\vec{i} = (i_1, i_2)^T$.

Iteration Block (IB): An iteration block is the granularity at which loop iterations are distributed across multiple cores for execution. Given an m-level loop nest with $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, an iteration block IB is defined as:

[3]In this paper, we focus on affine programs [11]. That is, the loop bounds and array references are assumed to be affine functions of the loop iterators. For an application program that has both affine and non-affine references, our approach optimizes only the affine ones.

(1) $IB = \{\vec{i_1}, \vec{i_2}, \cdots, \vec{i_q}\}$, where each $\vec{i_x}, (1 \leq x \leq q)$ is one loop iteration.
(2) For any two loop iterations $\vec{i_x} = (\underbrace{i_{x_1}, i_{x_2}, \cdots, i_{x_t}}_{t}, i_{x_{t+1}}, \cdots, i_{x_n})^T$ and $\vec{i_y} = (\underbrace{i_{y_1}, i_{y_2}, \cdots, i_{y_t}}_{t}, i_{y_{t+1}}, \cdots, i_{y_n})^T$ $(1 < x, y < q, 1 < t < n)$ that belong to the same IB, we have $i_{x_v} = i_{y_v}$ $(1 <= v <= t)$. The value of t determines the outermost t-level loops can be potentially parallelized across multiple cores. In other words, it specifies the size of an IB. Choosing a proper t involves a tradeoff between parallelism and cache locality. Specifically, a small value of t indicates a parallelization of the loop iterations across different cores in big chunks (IBs), in which, consecutive data accesses to the same cache line are contained in a single chunk and assigned to one core. While this scenario is good for cache locality, there are two drawbacks of using large-sized IB. First, a large block would typically access many memory banks. As a result, we lose the flexibility of scheduling IBs towards optimizing MLP. Second, large blocks can also lead to imbalanced computation among cores and sacrifice performance due to less parallelism. On the other hand, a large value of t (small-sized IBs) can hurt cache locality. This is due to the fact that subsequent accesses to the same cache line might be distributed across different cores, resulting in extra cache misses from different cores. In our framework, we choose to use proper small-sized iteration blocks so that the loop nest can be parallelized in a fine-granular and balanced fashion (Section 5.3). Meanwhile, loop iterations that access the same cache line are grouped into the same IB. Therefore, the temporal locality of accesses to a cache line is maintained within an IB.

Iteration Window (IW): An iteration window is a group of iteration blocks assigned to a core. Formally, an IW assigned to core c_i is denoted as $IW_{(i,j)}$ with $1 \leq i \leq C$, where C is the total number of cores, and i and j denote the core ID and IW ID, respectively. In our approach, an IW follows two rules:
(1) For any $IW_{(i,j)}, IW_{(i,k)}$ $(j < k)$ from core c_i, loop iterations in $IW_{(i,j)}$ are executed before loop iterations in $IW_{(i,k)}$.
(2) For any two $IW_{(m,j)}$, $IW_{(n,j)}$ from cores c_m and c_n, loop iterations from $IW_{(m,j)}$ and $IW_{(n,j)}$ are expected to execute concurrently at runtime.
Rule (1) captures the execution order of IWs within a core, whereas rule (2) provides concurrent execution of IWs from multiple cores. Each $IW_{(i,j)}$ can be expanded as a set of IBs, i.e., $\{IB_{(i,j,1)}, IB_{(i,j,2)}, \cdots, IB_{(i,j,n)}\}$. In general, we use a large-sized IWs, so that rule (2) can be satisfied at runtime. However, the size of IW cannot be arbitrary large. This is because our loop permutation changes the data access order within an IW, and we do *not* want to introduce extra cache misses within an IW. We discuss how we choose a proper size of IW in Section 5.3.

Data Block (DB): A data block is a group of data elements (addresses) in data space. Specifically, an DB can be expressed as a set of array elements accessed by array references ($\{\vec{r_1}, \vec{r_2}, \cdots, \vec{r_p}\}$). In our framework, we use the *cache line size* to determine the DB size (p). That is, array references in the same DB are mapped to the same cache line, and can potentially benefit from spatial locality. It is possible that multiple DBs are mapped to a single cache line, but not the other way around. Recall that data accesses to an n-dimensional array can be represented as vectors in the data space ($\vec{r_k} = (r_{k_1}, r_{k_2}, \cdots, r_{k_n})^T$, where $1 \leq k \leq p$, and r_{k_i} $(1 \leq i \leq n)$ is the array index in the ith dimension). An DB can formally be defined as:
(1) $DB = \{\vec{r_1}, \vec{r_2}, \cdots, \vec{r_p}\}$, and
(2) For any two $\vec{r_x} = (\underbrace{r_{x_1}, r_{x_2}, \cdots, r_{x_u}}_{u}, r_{x_{u+1}}, \cdots, r_{x_n})^T$ and $\vec{r_y} = (\underbrace{r_{y_1}, r_{y_2}, \cdots, r_{y_u}}_{u}, r_{y_{u+1}}, \cdots, r_{y_n})^T$ $(1 \leq x, y \leq p, 1 \leq u \leq n)$ in the same DB, we have $r_{x_v} = r_{y_v}$ for any v where $1 \leq v \leq u$ and u is determined by the cache line size.

Data Set (DS): A data set is a group of DBs referenced by loop iterations from the *same* iteration block (IB). The data blocks (DBs) referenced by an IB can be inferred by applying reference matrix and offset to each iteration vector in the IB. Specifically, an DS can be defined as follows:
(1) $DS = \{DB_1, DB_2, \cdots, DB_n\}$, with $DB_x = \{\vec{r_{x1}}, \vec{r_{x2}}, \cdots, \vec{r_{xq}}\}$, where $\vec{r_{xj}} = A\vec{i_{xj}} + \vec{o}$ $(1 \leq j \leq q)$.
(2) Given an array reference $\vec{r_x}$ from DS, if $\vec{r_x} \in DB_i$, then DB_i is included in DS.

To summarize, loop iterations in a loop nest are grouped into IBs and IB is the granularity we use to parallelize and schedule a loop nest (distribute its iterations) across multiple cores. IBs within the same IW can execute in any order without compromising cache locality. Further, IWs are executed sequentially within each core. The array references made by an IB collectively constitute an DS. A DS may contain several DBs. Array references to the same cache line are grouped into the same DB.

5.2 Optimization Goal

With these four concepts (IB, IW, DB, and DS) in place, we are now ready to discuss our optimization target. Each DB is associated with a memory bank and an LLC bank, based on its address. We use *memory bank vector* to represent the memory bank of an DB. Given a memory bank vector $\vec{b} = (b_1, b_2, \cdots, b_n)$, where n is the total number of memory banks, bit b_i $(1 \leq i \leq n)$ is set to 1 in a bank vector, if the requested DB (cache line) is mapped to memory bank i. Similarly, for LLC banks, we define *LLC bank vectors* as $\vec{c} = (c_1, c_2, \cdots, c_l)$, where l is the total number of LLC banks. We use $\sum \vec{b}$ and $\sum \vec{c}$ to denote the total number of 1s in memory bank vector and LLC bank vector, respectively. Obviously, for $\vec{b}$ and $\vec{c}$ of a given DB, we have $\sum \vec{b}_{DB} = 1$ and

Algorithm 1 MLP and CLP aware iteration block scheduling (Single array, Manycore).

INPUT: Number of cores (N); Size of iteration window (W); Number of total iteration blocks (M);
OUTPUT: Iteration windows to core mapping.

```
1: //get MLP vector of iteration blocks
2: function GET_MLP_OF_IB(IterationBlockPool)
3:     for each IB_i in IterationBlockPool do
4:         b_IB_i ← 0
5:         data_blocks ← get_data_blocks(IB_i)
6:         for each data block DB_j in data_blocks do
7:             b_IB_i ∪ get_MLP_Vector(DB_j)
8:     return (b_IB)
9: function SINGLE_CORE(W, MLP_IB, IterationBlockPool, g)
10:    iterationWindow ← ∅
11:    l ← 0 Intra-core MLP vector
12:    TempSet ← ∅
13:    //Schedule an iteration block if it improves MLP
14:    for each IB_i in IterationBlockPool do
15:        temp_g ← MLP_IB_i ∪ g //calculate inter-core MLP
16:        if Σ temp_g > Σ g then
17:            IterationWindow ∪ b_i
18:            Update IterationBlockPool and W
19:            l ← l ∪ MLP_IB_i;  g ← temp_g
20:            Break if W == 0
21:        else if Σ temp_g == Σ g then
22:            temp_l ← MLP_IB_i ∪ l //calculate intra-core MLP
23:            if Σ temp_l > Σ l then
24:                IterationWindow ∪ b_i
25:                Update IterationBlockPool and W
26:                l ← temp_l
27:                Break if W == 0
28:            else if Σ temp_l == Σ l then
29:                TempSet ∪ IB_i
30:        if Σ (g ∪ l) ≥ β × MAX_MLP then
31:            Add remaining IBs in IterationBlockPool to TempSet
32:            Break
33:    //Schedule an iteration block if it improves CLP
34:    if W ≠ 0 then
35:        for each iteration block IB_i in TempSet do
36:            Choose IB_i if it improve CLP.
37:            remove IB_i from IterationBlockPool and TempSet
38:            Break if W == 0
39:    return (IterationWindow, g)
40: k ← 0
41: Generate dependency of IterationBlockPool
42: MLP_IB = GET_MLP_OF_IB(IterationBlockPool)
43: while There are iteration blocks in IterationBlockPool do
44:    g ← 0 //Inter-core MLP vector
45:    for C_i from C_1 to C_N do
46:        (IW_k,C_i, g) = SINGLE_CORE(W, MLP_IB, IterationBlockPool, g)
47:        schedule_k ∪ IW_k,C_i
48:    k ← k + 1 //Increase 1 schedule time unit
```

$\sum \vec{c}_{DB} = 1$, indicating that the references to an DB only access *one* memory bank and *one* LLC bank.

An IB accesses a set of DBs (i.e., an DS). To capture the memory banks accessed by an IB, we apply bit-wise *or* ($\cup$) operation over all the bank vectors associated with the DBs in an DS. Specifically, the bank vector of an IB is expressed as $\vec{b}_{IB} = \cup\{\vec{b}_{DB_1}, \vec{b}_{DB_2}, \cdots, \vec{b}_{DB_n}\}$.

To optimize MLP, we further apply bitwise *or* ($\cup$) on memory bank vectors among IBs. At each scheduling step[4], we try to maximize $\sum \cup\vec{b}_{(i,j,k)}$, where i is the core id, j is the IW id, and k is the IB id. For inter-core MLP, we choose the IBs such that $\sum \cup\vec{b}_{(i,\boldsymbol{J},k)}$ is maximum for a given IW J. Similarly, for intra-core MLP, we maximize $\sum \cup\vec{b}_{(\boldsymbol{I},\boldsymbol{J},k)}$ for a given core I and a given IW J.

In the CLP-first approach on the other hand, we calculate CLP using the same approach discussed above, and the only difference is that we replace memory bank vector with LLC bank vector ($\vec{c}$) as our primary optimization target is CLP.

5.3 Loop Strip-Mining

Let us consider the two-level loop nest shown in Figure 4a. There are three references to array a and one reference to array b in each innermost loop iteration (ith dimension). Focusing on array a, the corresponding data access pattern is plotted in Figure 5a. We assume that there is a total of 4 memory banks (the number in the oval is the bank ID). Figure 4b and Figure 5b show the results of parallelizing the outer j loop between two cores without applying our approach. In our framework, we first apply loop strip-mining based on the iteration window size and the iteration block size to eliminate the potential extra cache misses.

Iteration block size: Recall from Section 5.1 that small-sized IBs can lead to extra cache misses, whereas large-sized IBs can reduce MLP. In the example shown in Figure 4a, there are three data references to array a in each innermost loop iteration ($a[j][i-1]$, $a[j][i]$, and $a[j][i+1]$). As the iterator (i) moves forward, these three references also move forward as a "group", as illustrated in Figure 5a. Since these three references move across the boundaries between two neighboring DBs (highlighted with square in the figure), it is simply impossible to have each IB access only one data block. Let us assume in this case that all three data references go to the same DB from iteration 1 to $k-1$ of the inner ith loop. In the kth iteration, $a[j][k+1]$ refers to a data element in the second DB, whereas $a[j][k-1]$ and $a[j][k]$ still refer to the data elements in the first DB. Similarly, in the $(k+1)$th iteration, $a[j][k+2]$ and $a[j][k+1]$ refer to the data elements in the second DB, whereas only $a[j][k]$ refers to a data element in the first DB. Now, the question is: should we group iterations k and $k+1$ in the first IB or in the second IB? Let us assume there are two different iteration blocks (IB_1, IB_2) that are assigned to two different cores (c_1, c_2). We want iterations k and $k+1$ both in either IB_1 or IB_2 so that only one core (either c_1 or c_2) accesses both the DBs and the other core accesses only one DB. For instance, the consequence of grouping iterations k and $k+1$ in IB_1 is that IB_2 starts with iteration $k+2$ which does not access the first DB, as all of the array indices ($a[j][k+1]$, $a[j][k+2]$, $a[j][k+3]$) are in the second DB. Otherwise, c_1 needs to access 2 DBs, and c_2 also needs to access 2 DBs.

Iteration window size: IBs are grouped into iteration windows. The size of an IW is decided by the cache capacity. Specifically,

$$IW\_size = C/(D * n), \tag{1}$$

[4] A scheduling step is a round of assigning iteration blocks to cores. More specifically, at each scheduling step, we assign each core a number of iteration blocks. This number is determined by the iteration window size.

```
(a) original code
for (j =0; j < m; j++)
  for (i = 2; i < n; i++)
    b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]

(b) naive parallelization
core 1:
for (j = 0; j < m/2; j++)
  for (i = 2; i < n; i++)
    b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]

core 2:
for (j = m/2; j < m; j++)
  for (i = 2; i < n; i++)
    b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]

(c) loop strip mining
for (iw = 0; iw < m; iw+=iw_size )
  for (j = iw; j < iw+iw_size; j++)
    for (ib = 2; ib < n; ib+=ib_size)
      for (i = ib; i < ib+ib_size; i++)
        b[j][i] = a[j][i-1] + a[j][i] + a[j][i+1]

(d) loop iteration block scheduling
core 1:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
  for (each ib in iw) /**number of iteration blocks in a window */
    for (i = ib; i < ib+ib_size; i++)
      j' = calculate(iw, ib)
      b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]

core 2:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
  for (each ib in iw) /**number of iteration blocks in a window */
    for (i = ib; i < ib+ib_size; i++)
      j' = calculate(iw, ib)
      b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]

(e) loop permutation
core 1:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
  for (i = ib; i < ib+ib_size; i++)
    for (each ib in iw) /**number of iteration blocks in a window */
      j' = calculate(iw, ib)
      b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]

core 2:
/**After computation-to-core assignment*/
for (iw from 1 to k) /**number of iteration windows/
  for (i = ib; i < ib+ib_size; i++)
    for (each ib in iw) /**number of iteration blocks in a window */
      j' = calculate(iw, ib)
      b[j'][i] = a[j'][i-1] + a[j'][i] + a[j'][i+1]
```

Figure 4. An example code fragment and its transformed versions after applying our optimizations.

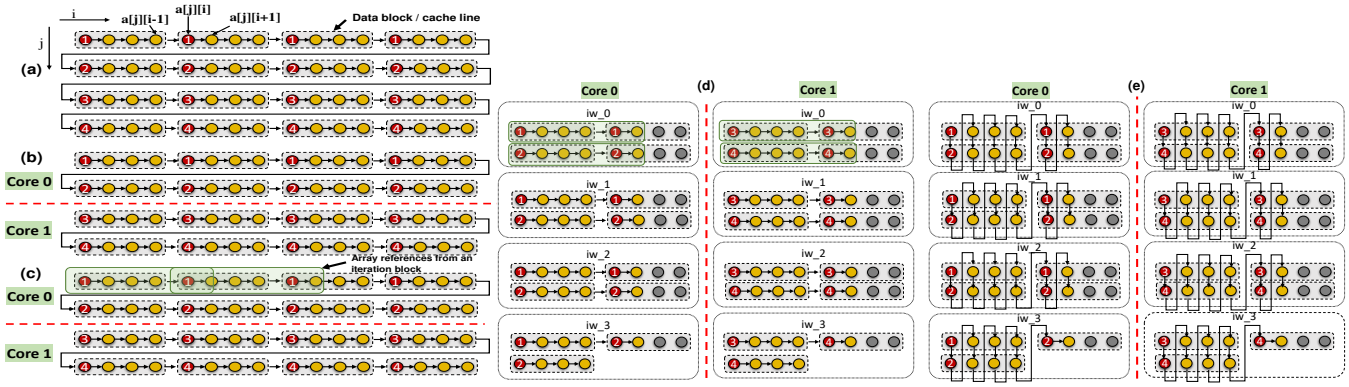


Figure 5. The corresponding memory access pattern of the code example in Figure 4. (a) Default access pattern. (b) Parallelization between two cores. (c) Forming iteration block and iteration window. (d) Iteration block scheduling. (e) Loop permutation to cluster cache misses.

where C is the cache capacity in terms of the number of cache lines, D is the size of DS, and n is the number of cores. Since the size of DS captures the number of DBs (cache lines) accessed by an IB, the IW size captures the number of maximum IBs per window without hurting cache locality.

5.4 Loop Iteration Block Scheduling

After the sizes of IB and IW are determined, we apply loop-strip-mining (Figure 5c). Then, the loop nest after strip-mining is treated as consecutive iteration blocks which are input to our proposed IB scheduling (IterationBlockPool in Algorithm 1). We give the formal description of our scheduling approach in **Algorithm 1**. At high-level, Our scheme tries to achieve two objectives: 1) at each time unit of scheduling, we choose an IB that improves inter-core MLP as well as intra-core MLP, and 2) once the MLP cannot be improved any further (in the MLP-first approach), CLP is considered as the next optimization target.[5] To do that, there are three steps involved in the algorithm: 1) dependence analysis at IB granularity (line 41), 2) obtaining the MLP vector of each IB (line 42), and 3) scheduling IBs for each core (lines 45 to 47).

Our approach starts by building dependence graph (DAG) of iteration blocks. If two IBs are dependent, the necessary ordering is enforced by inserting synchronization between those two dependent iterations blocks. In subsequent IB scheduling, we always try to select independent IBs for inter-core optimization. If we cannot find independent IB, dependent IBs are chosen and correctness is guaranteed by synchronizations. We want to emphasize that, to reduce the overhead of synchronization, after scheduling all IBs, we perform a "transitive closure" based synchronization minimization strategy to remove redundant synchronizations.

Our loop iteration scheduling chooses IB from in *IterationBlockPool* (line 46). We define one scheduling cycle (time unit) as a round of assigning IBs to each core which fills an iteration window (lines 9 to 39). The scheduling ends when all the iteration blocks in *IterationBlockPool* have been scheduled. Our IB scheduling consists of three major steps: for each core and for each iteration window, (1) we first choose IBs that optimize inter-core MLP (lines 15 to 20), and

[5] In the CLP-first approach on the other hand, we try choose an IB that improves CLP, and once the CLP cannot be improved any further, MLP optimization is attempted.

then, (2) we select IBs that maximize intra-core MLP (lines 22 to 29), and finally, (3) if the iteration window still have slots and the preferred value of MLP is reached (line 30), we choose IBs that optimize CLP (lines 33 to 38). To be more specific, if a candidate IB contributes more to the inter-core MLP or intra-core MLP, we schedule that IB in the current IW. Otherwise, if the candidate provides the same MLP, we add it to a $TempSet$, which holds all the candidates that can be used for improving CLP. We use a factor β (line 30) to balance CLP and MLP (details are provided in Section 5.6). The complexity of this algorithm is $O(NM^2)$, where N is the number of cores and M is the total number of iteration blocks.

Figures 4d and 5d give an example code fragment and the corresponding array reference pattern, respectively, after applying our iteration block scheduling. As can be seen from Figure 5d, the inter-core MLP is 4 for each of the four iteration windows, and at the same time, the intra-core MLP is 2. Clearly, compared to the default access pattern depicted in Figure 5c (where MLP is 2 for inter-core and 1 for intra-core), *both* inter-core and intra-core MLP are improved.

5.5 Loop Permutation

At this point, iteration blocks are scheduled across cores and iteration windows are formed. The last step is loop permutation. Recall from our discussion in Section 3 that we interchange the innermost loop with the second innermost loop such that the misses are grouped together to reach an improved MLP. We apply loop permutation to loop iterations within an iteration window. Since all the data blocks accessed by the iterations within an iteration window across cores can fit in the cache (due to our selection of the size of iteration window), our permutation will t not cause any extra cache misses. Figures 4e and 5e depict the loop nest body and the array reference pattern, respectively, after applying loop permutation.

Table 1. An example illustrating the trade-off between MLP and CLP.

	Memory Banks				LLC Banks			
	b_0	b_1	b_2	b_3	b_0	b_1	b_2	b_3
IB_0	0	1	3	0	2	0	4	0
IB_1	1	2	0	1	2	0	4	0
IB_2	0	0	2	2	0	2	0	4
IB_3	2	2	0	0	0	3	0	3

As a summary, our approach takes iteration blocks (IterationBlockPool in Algorithm 1) as input, where each iteration block has its unique identifier (id). Then, Algorithm 1 picks up iteration blocks from IterationBlockPool and forms iteration window across the cores. As a result, each core is associated with a sequence of iteration blocks. During execution, array reference index is derived from iteration window id and iteration block id. Note that, our approach generates the transformed source code of each loop nest. In the experimental results discussed in Section 6 (in both the simulation and KNL experiments), we enabled all vectorization and data locality optimizations. In KNL, we used Intel compiler (icc) to generate code. In simulation-based experiments, gcc code generator (with the highest optimization level) is used.

5.6 Striking a Balance between CLP and MLP

Although our discussion above mainly focuses on MLP-first which takes CLP into consideration only as secondary objective, as an alternative approach, one can also choose CLP as the primary optimization goal over MLP, or even choose to trade MLP for CLP. To explain the benefits of doing so, let us consider the access patterns of four iteration blocks, shown in Table 1. Each value shown in the table represents the number of accesses to a particular memory/LLC bank. Let us assume that IB_0 has already been scheduled, and we are now choosing the next IB from IB_1, IB_2, and IB_3. Based on our previous discussion, we choose IB_2 since $\sum \vec{b}_{IB_0} \cup \vec{b}_{IB_1}$ is 4, meaning that MLP is maximized to 4. However, doing so is not good from a CLP perspective, as it adds more latency to LLC hits and the value of CLP is only 2 . This is because both IB_0 and IB_1 access LLC banks b_0 and b_2, resulting in a total of 4 (2+2) and 8 (4+4) accesses to LLC banks b_0 and b_2, respectively. Since the hits in an LLC bank compete with one another, this cache contention can easily offset the potential benefits coming from optimized MLP.

As a result, one may want to explore a more "balanced approach" between MLP and CLP. We enable a tradeoff between MLP and CLP by determining the value of parameter β in **Algorithm 1** (line 30). More specifically, we employ the following optimization target:

$$Target\_Metric = \beta \times MLP + (1 - \beta) \times CLP, \qquad (2)$$

where we have $0 \leq \beta \leq 1$. To determine an optimum value for β, we first need to augment our approach explained so far to take the number of bank accesses into consideration. More specifically, we use integer values (instead of boolean values which are used in previous discussion) for the entries in a bank vector of an IB, so that we can capture the number of accesses made to a given bank. Also, we define an operation, $\uplus$, which performs the entry-wise addition between two bank vectors, and compute the *standard deviation* (SD) for the weighted MLP vector. Note that, SD captures the distribution of accesses across different banks. In particular, a higher SD indicates that the accesses are not balanced and some banks have long service queues populated by many accesses, whereas other banks have only a few accesses.

Let us assume that the request service latency at memory banks is η_m and the service latency at LLC banks is η_n. Let us further assume that the weighted bank MLP and CLP vectors for the current iteration block $IB_{(c)urrent}$ are $\vec{m}_c$ and $\vec{c}_c$, respectively, and we are selecting the next iteration block from IB_x and IB_y. We use $\vec{m}_x$ and $\vec{c}_x$ to denote, respectively, the MLP vector and CLP vector of IB_x. Similarly for IB_y, we have $\vec{m}_y$ and $\vec{c}_y$. We can calculate the corresponding standard deviation between IB_c and IB_x using $SD_m(IB_c, IB_x) = SD_m((\vec{m}_c \uplus \vec{m}_x) * \eta_m)$. In terms of CLP, the corresponding SD

is $SD_c(IB_c, IB_x) = SD_c((\vec{c_c} \uplus \vec{c_x}) * \eta_c)$. Therefore, we have $SD(IB_c, IB_x) = SD_m(IB_c, IB_x) + SD_c(IB_c, IB_x)$. We then define $\delta = SD(IB_c, IB_x) - SD(IB_c, IB_y)$ to capture the *difference* of SDs between two iteration blocks IB_x and IB_y. If $\delta > 0$, it indicates that choosing IB_y is better since a smaller value of SD indicates a more balanced distribution of accesses to different banks. On the other hand, if $\delta = 0$, we randomly choose one iteration block from IB_x and IB_y.

Note that, to reduce potential overheads, we only perform SD analysis for the first two iteration blocks of each iteration window. After we select the first two blocks, we check the MLP vector and CLP vector of the chosen iteration block and determine the value of β. In other words, we choose the most beneficial iteration blocks (in terms of a both CLP and MLP) at the beginning of constructing an iteration window. We then use the obtained β to choose the successive iteration blocks for that iteration window. That is, our compiler automatically determines the value of β for each iteration window. Later, we present the distribution of β values determined by our approach.

5.7 Discussion

We now discuss the generality of our approach. If the target system employs dynamic NUCA (DNUCA) where a cache line doesn't have a fixed home bank and can reside in any cache bank in the system, our approach can be augmented to predict the locations (LLC bank) for DBs (for the next scheduling epoch). Equipped with such prediction, our approach can be used for DNUCA as well. If the LLCs are private (which is not very common), our approach will have limited impact on CLP. However, we want to emphasize that, even when the target-system employs DNUCA or private LLC, our framework will still improve MLP. Also, our approach works with different NoC topologies (e.g., mesh, butterfly, etc). As long as the information of cache/memory placement, cache management policy (i.e., SNUCA or DNUCA), and network topology are exposed to our compiler, we can apply our optimization on CLP and MLP.

6 Experimental Evaluation

6.1 Setup

We conducted both a simulation based study as well as experiments on a commercial manycore architecture. The reason why we performed simulation based experiments is twofold. First, it is not possible to extract CLP and MLP information from a real hardware, as current performance counters, debugging tools and performance evaluation tools do not provide CLP or MLP statistics. Second, to see how our proposed compiler based approach performs under different architectures, we wanted to change some of the architectural parameters, and this could be done only in a simulation based environment. However, in addition to the simulation based experiments, we also performed experiments on Intel Knight's Landing [45], a commercial manycore/accelerator system. In KNL, each experiment is repeated 15 times, and the median-value is used in the presented-plots. The variance between the lowest and highest values was less than 2% in all experiments.

Table 2. System setup.

Manycore Size, Frequency	36 cores (6 × 6), 1 GHz, 2-issue
L1 Cache	16 KB; 8-way; 32 bytes/line
L2 Cache	512 KB/core; 16-way; 128 bytes/line
Hardware Prefetcher	stream prefetcher with 32 streams, prefetch degree of 4, prefetch distance of 64 cache lines
Coherence Protocol	MOESI
Router Overhead	3 cycles
Memory Row Size	2 KB
On-Chip Network Frequency	1 GHz
Routing Policy	X-Y routing
DRAM Controller	open-row policy using FR-FCFS scheduling policy 128-entry MSHR and memory request queue
DRAM	DDR4-2400; 250 request buffer entries 4 MCs; 1 rank/channel; 8 banks/rank
Row-Buffer Size	2 KB
Operating System	Linux 4.7
Data Distribution Across Memory Banks	2KB granularity, round-robin
Data Distribution Across LLC Banks	128 bytes granularity, round-robin

For our simulation based experiments, we used gem5 [2] infrastructure to execute 12 multithreaded applications in the full system mode. Ten of our twelve multithreaded applications are from Splash-2 [57] and the remaining two are matrix multiplication (mxm) and syr2k, a kernel that performs a rank-2k matrix-matrix operation. The input dataset sizes of these applications vary between 33.1 MB and 1.4 GB, and their LLC (L2) cache miss rates range from 16.6% to 37.2% (in simulator). Also, the number of iteration blocks (as defined in Section 5.1) ranges between 8,032 and 31,554.

We used LLVM [3] to implement our compiler support. Table 2 gives the important parameters of the manycore/memory configurations modeled in this work using gem5. In most of our simulation-based experiments, we used 36 cores and parallelized each application program such that each core executes one thread at a time (i.e., one-to-one thread-to-core mapping In all experiments, we set the scheduling epoch length 2500 cycles. We also enable both vectorization and the hardware-based prefetcher (stream-prefetcher).

In this work, we compare *eight different versions* for the execution of our application programs:

- *Default:* In this version (also called the *original version*), the iterations of a loop nest are divided into iteration blocks of equal size, and the resulting iteration blocks are assigned to available cores in a *round-robin fashion*. Unless stated otherwise, the results with all the remaining versions described below are *normalized* with respect to this version.
- *Clustering:* This version implements the approach in [36]. While it clusters the LLC misses, it does not specifically consider cache-level parallelism.
- *MLP-first:* In this version, MLP is optimized aggressively using the approach explained in Section 5.4. As noted there,

CLP is considered, as a secondary optimization, only if doing so does not hurt MLP.

- *CLP-first:* This is the other extreme where CLP is aggressively exploited first, and MLP is considered only if doing so does not hurt CLP.
- *Balanced:* This is the approach defended in Section 5.6, where the compiler tries to determine the value of β parameter to *co-optimize* (balance) MLP and CLP so that maximum performance can be achieved. It is important to note that MLP-first and CLP-first are just two different incarnations of Balanced, with $\beta = 1$ and $\beta = 0$, respectively.
- *Locality-Aware-MLP:* This is a recently published compiler approach [8] that targets optimizing bank-level parallelism in a locality-conscious (row-buffer aware) manner. It does not consider CLP; however, it considers row-buffer locality.
- *PAR-BS:* This is a pure *hardware based* optimization scheme proposed by [34]. It (i) handles DRAM requests in batches to provide fairness across competing memory requests coming from different threads, and (ii) employs a parallelism-aware DRAM scheduling policy with the goal of processing requests from threads in parallel in the DRAM banks, to improve memory bank-level parallelism.
- *Ideal:* This version represents the *maximum potential savings*. It is implemented in the simulator by maximizing both MLP and CLP. It, in a sense achieves, at the same time, the MLP performance of MLP-first, and the CLP performance of CLP-first. Note that, this version is *not* practical as CLP and MLP can conflict with one another, and it is *not* always possible to maximize the both at the same time.

While we tested all these versions in our simulator (gem5), the *PAR-BS* and *Ideal* versions could not be used on Intel manycore system, as the former is a pure hardware based scheme that requires architectural modifications and the latter represents a limit study that can be evaluated only in a simulated environment. We also want to emphasize that, unless stated otherwise, all these versions have the *same degree of compute parallelism* and use the *same set of conventional data locality optimizations* (such as loop permutation and tiling that collectively minimize the number of LLC misses), and that they only differ how they map and schedule iteration blocks. The accuracy of the CME implementation employed in estimating LLC misses ranged between 79.14% and 88.36%.

6.2 Results with the Manycore Simulator

We present the MLP results in Figure 6. As expected, MLP-first generates the best MLP results. While CLP-first performs much worse than MLP-first, it is still better than the default version, as CLP-first considers MLP if doing so does not hurt CLP. Also, Clustering does not perform very well, as mere clustering of memory accesses does *not* guarantee MLP improvement (though it performs, as can be expected, better than the default version). Overall, PAR-BS performs slightly better than Clustering; but, since the throughput

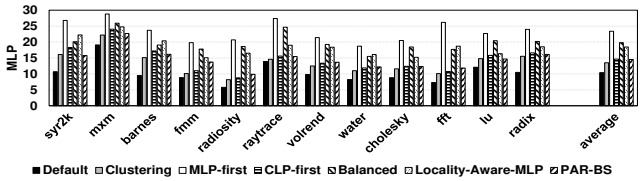


Figure 6. MLP Results.

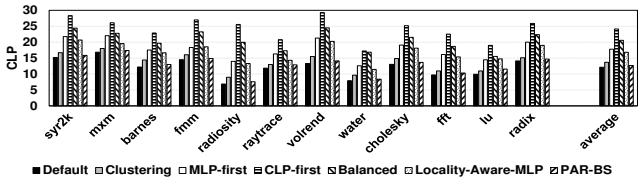


Figure 7. CLP Results.

optimization it brings is balanced with fairness optimization, its performance is not as good as Balanced. Also, since PAR-BS is a pure hardware optimization, it is not as good as compiler based schemes that have the flexibility of performing whole program analysis. Finally, Locality-Aware-MLP generates comparable MLP results to Balanced, and Balanced outperforms all the versions tested (except MLP-first).

Figure 7 presents the CLP results produced by the same versions. It can be observed that Clustering, MLP-first, and Locality-Aware-MLP do not perform very well as far as CLP is concerned, though they are in general better than the default version. This is hardly surprising, as these versions do not specifically target CLP. PAR-BS does not perform any better than Default, primarily because the former mainly targets MLP, not CLP. In comparison, Balanced performs quite well in terms of CLP, and the average CLP values Balanced and CLP-first bring are about 20.59 and 24.16, respectively.

Before presenting the execution cycle results, we want to discuss the impact of these versions on cache miss statistics and row-buffer statistics (as those two metrics are generally affected by how computations are scheduled). Figure 8 gives the percentage increases in LLC miss rates (over the default version) when using optimized versions. We observe that, none of these versions (Clustering, MLP-first, CLP-first, Balanced, Locality-Aware-MLP, and PAR-BS) has any noticeable impact on cache miss statistics, compared the default version (the highest increase on the L2 misses over the default version was about 1.48%). This is because Clustering, MLP-first, CLP-first, and Balanced are designed, as explained earlier, to make sure that cache misses are not increased. On the other hand, Locality-Aware-MLP improves row-buffer locality (to be presented shortly), and that slightly improves, as a side effect, cache hits as well. Figure 9, on the other hand, shows the variations (increase) in row-buffer misses, with respect to the default version. These results indicate that, most of the versions tested do not cause significant variations on row-buffer misses (in fact, all observed variations are between -2% and 2%). As expected, Locality-Aware-MLP leads to some improvement on row-buffer misses.

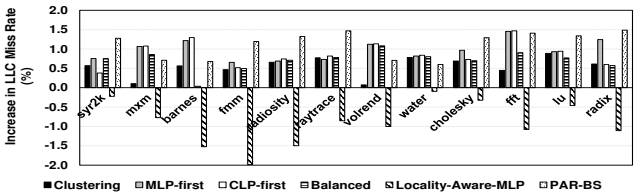


Figure 8. Increase in LLC miss rates (lower, the better).

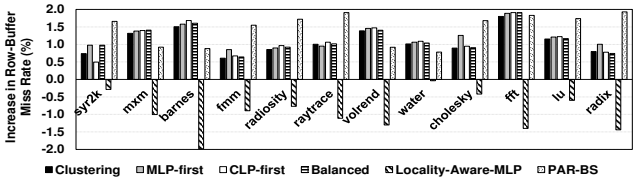


Figure 9. Increase in row-buffer miss rates (lower, the better).

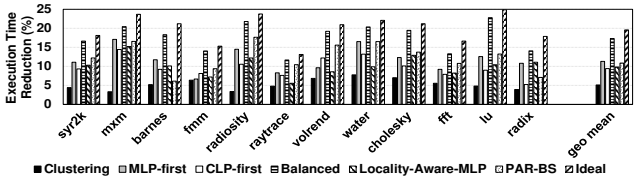


Figure 10. Execution time reduction (higher, the better).

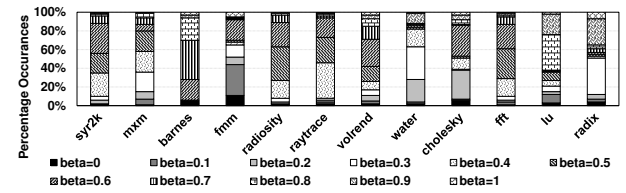


Figure 11. Distribution of the compiler-determined β values across all loop nest of our applications.

Figure 10 gives, for each version, the performance improvement (parallel execution time reduction) it brings over the default version (higher, the better). Note that, in these results, for a given version, all its impact on different metrics (e.g., CLP, MLP, row buffer miss rate, LLC miss rate) as well as all other overheads it incurs are included. We observe from these results that, our defended approach (Balanced) outperforms all remaining versions in all the 12 benchmarks tested (except Ideal, of course). This is because, as explained in Section 5.6, Balanced tries to perform the best trade-off between CLP and MLP, instead of trying to optimize one of them very aggressively (which is the case in CLP-first and MLP-first). Clustering does not perform well, as it fails to tap the full potential of bank-level parallelism and cache-level parallelism. On the other hand, Locality-Aware-MLP performs worse than our approach, as it does not consider CLP at all. Similarly, the improvements brought by PAR-BS are lower than those obtained using Balanced, as the former cannot optimize CLP. Further, note that, PAR-BS requires architecture level modifications, whereas our approach is a software-only solution. Overall, these results clearly underline the importance of optimizing both MLP and CLP

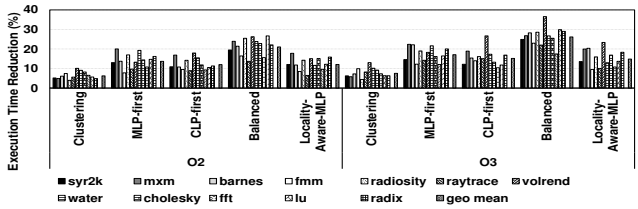


Figure 12. Results with "all-to-all" cluster mode and "cache" memory mode.

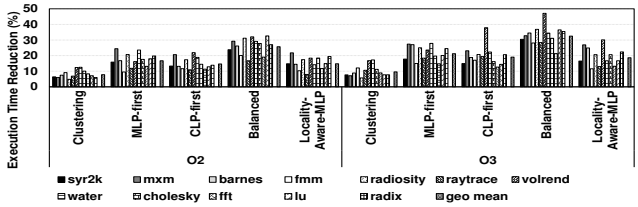


Figure 13. Results with "quadrant" cluster mode and "cache" memory mode.

together (in fact, Balanced brought an average performance improvement of 17.32% over the default scheme). Finally, the difference between Balanced and Ideal indicates that there is still some additional optimization opportunities that could be exploited by a more sophisticated compiler scheme.

Next, we delve into the behavior of Balanced a bit more, and explain the distribution of the compiler-determined β values across *all* the loop nests of a given application. These distribution results, plotted in Figure 11, indicate that, for an overwhelming majority of the loop nests in these 12 applications, the determined β values fall between 0.3 and 0.8, indicating that our approach (Balanced) really balances MLP and CLP quite well. These results also explain why Balanced performs better than CLP-First and MLP-first.

6.3 Results with Intel Knight's Landing

Recall from Section 2 that this architecture supports various "memory modes" and "cluster modes". We tested each of the three cluster modes (all-to-all, quadrant and sub-NUMA) under two memory modes. The first memory mode is the "cache mode" where all of the MCDRAM behave as a memory-side direct mapped cache in front of DDR4. Consequently, there is only a single visible pool of memory, and MCDRAM is simply treated as a high bandwidth (L3) cache. The second memory mode used is "hybrid mode" where some of MCDRAM space is configured as memory extension and the remaining MCDRAM space is configured as L3 cache. In this case, we profiled the applications in our experimental suite and allocated the some select data structures from the memory extension part of MCDRAM. Since the observed trends and overall conclusions with the cache and hybrid memory modes were similar, we present results from only the cache mode.

The KNL results are plotted in Figures 12, 13 and 14 for the "all-to-all+cache", "quadrant+cache" and "sub-NUMA+cache"

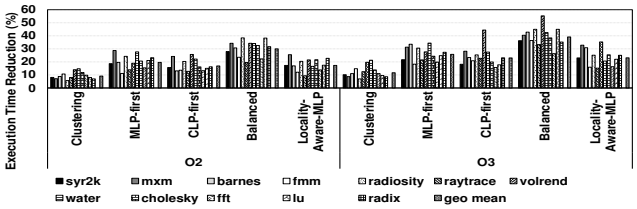


Figure 14. Results with "sub-NUMA" cluster mode and "cache" memory mode.

configurations, respectively. For each configuration, we performed two types of experiments: one with O2 compiler flag and one with O3 compiler flag. O2 corresponds to default set of *icc* optimizations; it includes vectorization as well as some loop transformations such as loop unrolling and inlining within source file. In O3 on the other hand, the compiler activates all optimizations in O2 level; in addition, it also uses more aggressive loop optimizations such as cache blocking (tiling), loop fusion, and loop interchange.

One can make several observations from the results presented in Figures 12, 13 and 14 (higher, the better). First, our approach improves the performance of *all* cluster nodes in *all* application programs tested. Second, the relative performance variations we observed in our simulation based experiments are valid in Intel Knight's Landing case as well. In particular, Balanced outperforms the remaining versions under all cluster modes, and MLP-First comes the second. Third, our approach (which is oriented towards reducing the latencies of *both* cache hits and cache misses) blends well with the traditional locality optimizations. More specifically, it can be observed that, as we move from O2 to O3, the overall execution time savings significantly improve. For example, in the case of the quadrant cluster mode, the average performance improvements brought (over the default version) are 25.69% and 32.54%, under O2 and O3, respectively.

7 Discussion of Related Work

Software Approaches to Memory-Level Parallelism: Pai et al. [36] proposed code transformations to increase memory parallelism by overlapping multiple read misses within the same instruction window, while preserving cache locality. Compared to their work, ours focuses on multithreaded applications running on manycores. Further, we propose iteration scheduling upon miss clustering (permutation), to improve inter-core MLP, and we consider CLP as well. In our experimental evaluations, we compared our proposed approach against [36] (which is annotated as *Clustering* in the experimental results). Ding et al. [8] proposed loop tile scheduling to improve bank level parallelism. Their approach schedules the tiled loop iterations across cores targeting BLP optimization. It does not consider CLP. We compared our approach to this prior work in our experimental evaluations. Targeting irregular applications, Tang et al. [49] proposed an inspector-executor based loop scheduling to improve bank-level parallelism across cores and row-buffer locality from each core's perspective. Compared to those two works, instead of focusing row-buffer locality, we demonstrate that intra-core MLP is also important, and our approach considers both inter-core and intra-core MLP. Further, we consider CLP to improve performance by reducing cache hits latencies.

Hardware Approaches to Memory-Level Parallelism: There have also been hardware researches optimize memory accesses in manycore systems [5, 10, 14, 15, 40, 59]. Mutlu et al. [34] proposed memory request batching to improve intra-thread bank-level parallelism while preserving row-buffer locality. Recall that, in our experimental evaluations presented earlier, we compared our approach to this pure hardware optimization. Qureshi et al. [39] proposed a MLP-aware cache management to reduce the memory stalls. Lee et al. [27] proposed BLP-aware memory prefetching to maximize the BLP of prefetch requests exposed to the DRAM controller. Compared to all these hardware efforts, we reduce the hardware design complexity by employing a "software-only" solution to improve MLP.

Traditional Data Locality Optimizations: The compiler literature is full of optimization techniques that target reducing the number of cache misses [1, 4, 6, 7, 12, 18, 19, 26, 28–31, 35, 47, 53–56]. There also exist cache bandwidth optimizations [16, 17, 46]. Our work presented in this paper is fundamentally different from these prior works, as it tries to optimize cache hit and miss latencies, instead of reducing the number of cache misses. Clearly, these two approaches (reducing the number of misses and reducing the miss latency) are complementary, and one would normally need to employ the both to maximize performance benefits (as already demonstrated with our O2/O3 results in KNL).

8 Conclusions

Targeting data access parallelism, we propose three alternative optimization strategies: (i) MLP-first, which primarily optimizes memory-level parallelism for LLC misses, (ii) CLP-first, which primarily optimizes cache-level parallelism for LLC hits, and (iii) Balanced, which strikes a balance between MLP and CLP. Our simulations show that the proposed three approaches bring 11.31%, 9.43%, and 17.32% reduction in execution times. We also tested our approach on a commercial manycore architecture, and the results collected indicate 17.06%, 15.19% and 26.15% average execution time savings with MLP-first, CLP-first and Balanced, respectively.

Acknowledgment

The authors thank PLDI reviewers for their constructive feedback, and Jennifer B. Sartor, for shepherding this paper. This research is supported in part by NSF grants #1526750, #1763681, #1439057, #1439021, #1629129, #1409095, #1626251, #1629915, and a grant from Intel.

References

[1] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *PLDI*.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer. Architecture. News* (2011).

[3] Uday Bondhugula, J. Ramanujam, and et al. 2008. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of Programming Language Design And Implementation (PLDI)*.

[4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[5] Yuan Chou, B. Fahs, and S. Abraham. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*.

[6] Michal Cierniak and Wei Li. 1995. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*.

[7] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS*.

[8] Wei Ding, Diana Guttman, and Mahmut Kandemir. 2014. Compiler Support for Optimizing Memory Bank-Level Parallelism. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[9] Wei Ding, Xulong Tang, Mahmut Kandemir, Yuanrui Zhang, and Emre Kultursay. 2015. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[10] Stijn Eyerman and Lieven Eeckhout. 2007. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*.

[11] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* (1992).

[12] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 1995. Detecting Coarse-grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Supercomputing*.

[13] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the 43rd International Symposium on Computer Architecture*.

[14] Ibrahim Hur and Calvin Lin. 2004. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[15] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[16] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. 2012. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6168944

[17] Toni Juan, Juan J. Navarro, and Olivier Temam. 1997. Data Caches for Superscalar Processors. In *Proceedings of the 11th International Conference on Supercomputing (ICS '97)*. ACM, New York, NY, USA, 60–67. https://doi.org/10.1145/263580.263595

[18] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prith Banerjee. 1999. A matrix-based approach to global locality optimization. In *Journal of Parallel and Distributed Computing*.

[19] Mahmut Kandemir, J. Ramanujam, Alok Choudhary, and Prithviraj Banerjee. 2001. A layout-conscious iteration space transformation technique. In *IEEE Transactions on Computers*.

[20] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. 2015. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[21] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *The Sixteenth International Symposium on High-Performance Computer Architecture*.

[23] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[25] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2017. POSTER: Location-Aware Computation Mapping for Manycore Processors.. In *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation*.

[26] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-centric Multi-level Blocking. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI)*.

[27] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. 2009. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[28] Shun-Tak Leung and John Zahorjan. 1995. *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[29] Wei Li. 1994. *Compiling for NUMA parallel machines*. Technical Report. Cornell University.

[30] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *ICS*.

[31] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. 1993. Array-data Flow Analysis and Its Use in Array Privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[32] Kathryn S. McKinley and Olivier Temam. 1996. A Quantitative Analysis of Loop Nest Locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[34] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*.

[35] M.F.P. O'Boyle and P.M.W. Knijnenburg. 2002. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distribute Computer* (2002).

[36] Vijay S. Pai and Sarita Adve. 1999. Code Transformations to Improve Memory Parallelism. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*.

[37] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[38] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*.

[39] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*.

[40] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. 2018. Quantifying and Optimizing Data Access Parallelism on Manycores. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[41] F. Jesus Sanchez, Antonio Gonzalez, and Mateo Velero. 1997. Static locality analysis for cache management. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[42] Akbar Sharifi, Emre Kultursay, Mahmut Kandemir, and Chita R. Das. 2012. Addressing End-to-End Memory Access Latency in NoC-Based Multicores. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[43] Akbar Shrifi, Wei Ding, Diana Guttman, Hui Zhao, Xulong Tang, Mahmut Kandemir, and Chita Das. 2017. DEMM: a Dynamic Energy-saving mechanism for Multicore Memories. In *Proceedings of the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[44] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: A Near-memory Content-aware Page-merging Architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

[45] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* (2016).

[46] Gurindar S. Sohi and Manoj Franklin. 1991. High-bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/106972.106980

[47] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*.

[48] Sam Van Den Steen and Lieven Eeckhout. 2018. Modeling Superscalar Processor Memory-Level Parallelism. *IEEE Computer Architecture Letters* (2018).

[49] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[50] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[51] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[52] Xulong Tang, Ashutosh Pattnaik, Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita R. Das. 2019. Quantifying Data Locality in Dynamic Parallelism in GPUs. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[53] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. 2003. Multi-dimensional incremental loop fusion for data locality. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. (ASAP)*.

[54] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS*.

[55] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*.

[56] Michael E. Wolf and Monica S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* (1991).

[57] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[58] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News* (1995).

[59] Praveen Yedlapalli, Jagadish Kotra, Emre Kultursay, Mahmut Kandemir, Chita R. Das, and Anand Sivasubramaniam. 2013. Meeting midway: Improving CMP performance with memory-side prefetching. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[60] Haibo Zhang, Prasanna Venkatesh Rengasamy, Nachiappan Chidambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut T. Kandemir, and Chita R. Das. 2018. FLOSS: FLOw Sensitive Scheduling on Mobile Platforms. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 173, 6 pages. https://doi.org/10.1145/3195970.3196052

[61] Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 517–531. https://doi.org/10.1145/3123939.3123948