Optimizing Off-Chip Accesses in Multicores

Wei Ding Xulong Tang
Mahmut Kandemir
Pennsylvania State University
University Park, PA, USA
{wzd109, xzt102,
kandemir}@cse.psu.edu

Yuanrui Zhang *
Intel Corp.
Santa Clara, CA, USA
yuanrui.zhang@intel.com

Emre Kultursay
Pennsylvania State University
University Park, PA, USA
euk139@cse.psu.edu

Abstract

In a network-on-chip (NoC) based manycore architecture, an off-chip data access (main memory access) needs to travel through the on-chip network, spending considerable amount of time within the chip (in addition to the memory access latency). In addition, it contends with on-chip (cache) accesses as both use the same NoC resources. In this paper, focusing on data-parallel, multithreaded applications, we propose a compiler-based off-chip data access localization strategy, which places data elements in the memory space such that an off-chip access traverses a minimum number of links (hops) to reach the memory controller that handles this access. This brings three main benefits. First, the network latency of off-chip accesses gets reduced; second, the network latency of on-chip accesses gets reduced; and finally, the memory latency of off-chip accesses improves, due to reduced queue latencies. We present an experimental evaluation of our optimization strategy using a set of 13 multithreaded application programs under both private and shared last-level caches. The results collected emphasize the importance of optimizing the off-chip data accesses.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors); D.3.4 [*Programming Languages*]: Processors

Keywords Manycores, off-chip accesses localization, memory controller

1. Introduction

After hitting the power wall, processor designs focused more on integrating multiple simple cores, instead of a complex processor on a single die. As shown in Figure 1, an NoC (network-on-chip [1]) based manycore is constructed from multiple point-to-point data links connected by switches such that messages can be relayed from any source node to any destination node over several links (hops). Optimizing data accesses in NoC-based manycore systems has received considerable attention lately. The proposed strategies include careful design of cache access/lookup strategies [2–4] and on-chip access localization [5, 6]. While these optimizations are certainly important and can provide significant performance and power benefits, they are mostly oriented towards minimizing the number of cache misses and do not have much impact on the latency of individual off-chip (main memory) accesses. The off-chip access latencies in an NOC-based manycore can be very important due to the following reasons:

- Since off-chip accesses must travel through the NoC to reach their target memory controllers, they can spend significant amount of time in the NoC, depending on the network congestion as well as the distance between the node that makes the off-chip request and the memory controller.
- Since off-chip accesses and on-chip accesses (cache accesses) share the same on-chip network, they contend for the same links and routers/buffers. Consequently, off-chip accesses also cause additional delays for on-chip cache accesses, further affecting the application performance.
- A network-delayed off-chip access will also join in the memory queue late, increasing its memory latency.[1]

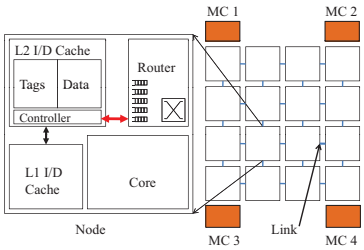


Figure 1: Structure of a two-dimensional 4×4 NoC based manycore with 4 memory controllers (MCs). Physical address space is distributed across memory controllers (MC1 through MC4).

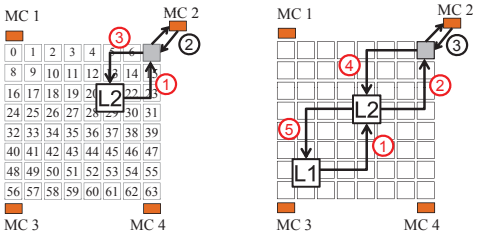


Figure 2: (a) Flow of a memory access on an NoC-based manycore with per core private L2 caches. The number attached to a node represents its core ID. (b) Flow of a memory access with shared L2 caches [1]. The L2 cache is shared by all cores.

[1] Memory latency includes the time spent in the queue as well.

* This work has been done when the author was with Penn State.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737989

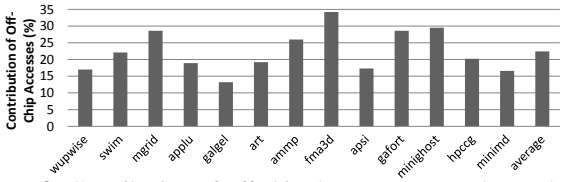


Figure 3: Contribution of off-chip data accesses to the total data accesses in an 8×8 mesh-based manycore system.

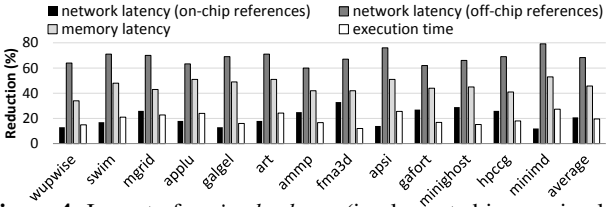


Figure 4: Impact of *optimal scheme* (implemented in our simulator) on network latency, memory latency as well as on execution time, under page interleaving of physical addresses across memory controllers.

Figures 2a and 2b show the memory access flows with private (per core) or shared L2s, respectively. In the case of private L2s (a), when an L1 miss is detected, a request is sent to the local L2 cache. In case of an L2 miss, this request is forwarded to a centralized L2 tag directory, which is cached in the memory controller (MC) to which the requested data is mapped (path 1). The directory decides whether to get the data from another on-chip L2 cache (in which case we have an on-chip data access), or issue an off-chip memory request. In the latter case, the MC schedules this request (path 2), and the response from the corresponding memory bank is sent to the private L2 cache (path 3). In comparison, for the shared L2 case (b), based on the physical address, a given data block is assigned to an L2 bank (called the home bank) where it resides (when it is on-chip); and all off-chip requests for that data block are issued by its home (L2) bank. In order to look-up the corresponding L2 bank, a request is sent from L1 to L2 (path 1). If the data is a hit in the L2, then the response data is sent back to the L1 (path 5). In case of an L2 miss, the request is forwarded to the memory controller (path 2). The memory controller schedules this request (path 3) and the response of the memory module is sent first to the L2 (path 4) and then to the L1 (path 5).

The additional cost incurred by an off-chip access (over an on-chip access) in an NoC-based manycore system is mainly a function of two parameters: (1) the time spent in the network (paths 1 and 3 in Figure 2a and paths 2 and 4 in Figure 2b), and (2) the time spent in MC and in accessing the memory bank (path 2 in Figure 2a and path 3 in Figure 2b).[2] Our experiments show that cumulative network latency, in a large manycore, can be comparable to the memory access latency, which means the network latency can play a significant role in off-chip accesses. This latency can be reduced by improving the *locality of off-chip accesses*, i.e., reducing the distance between the requester of an off-chip data element and the MC that manages the block/page that holds the data. For example, in Figure 2a, if the off-chip access request from the highlighted L2 bank is sent to MC3, the distance between the requester of the data and the node that is connected to the memory controller that holds the data is 10 links, whereas the distance for the off-chip data access through path 1 is only 4 links. Everything else being equal, this L2 would prefer to satisfy most of its off-chip data requests from the *nearest* memory controller (MC2).

Our **goal** in this work is to optimize the network behavior of off-chip data accesses by minimizing the distance they need to travel on the network. Note that, our novelty is not in the employment of data layout transformations in a compiler (after all, many prior parallelization and locality optimizations for data-intensive codes use data transformations in one way or another); rather, our novelty lies in demonstrating how layout transformations can be used to achieve a different goal, namely, optimizing the behavior of off-chip memory accesses. To our knowledge, this is the first compiler-based work that targets *off-chip* accesses in NoC-based manycores with both private and shared L2(s). Our specific contributions can be summarized as follows:

- We present experimental evidence showing that an *optimal scheme* that reduces the latency of off-chip memory accesses on both on-chip network and off-chip memory queue can have a significant impact on not only the off-chip accesses but also on-chip accesses.
- Motivated by this characterization, we propose a compiler-guided data layout transformation strategy that improves the locality of off-chip data accesses in NoC-based manycores with per core private L2s as well as shared L2s. Our strategy handles both page and cache interleaving of physical addresses across memory controllers. In the case of page interleaving, our approach also needs help from the Operating System (OS).
- We present an experimental evaluation of our approach using 13 multi-threaded application programs. Our approach brings an average of 20.5% saving in execution time (under cache-line interleaving of physical addresses across memory controllers). The corresponding execution time improvement in the case of shared L2 is about 24.3%.

While, conceptually, loop restructuring could also be used to achieve our goals, we opted to use data layout transformations in this work, mainly because loop transformations are constrained by data and control dependences. In contrast, data transformations are essentially a kind of renaming and not affected by dependences, thus providing more flexibility to the compiler.

Also, while one can envision a pure hardware-based custom scheme which can implement a physical address distribution across last-level caches and memory controllers such that the distance covered by cache misses is minimized, such a scheme would require changes to original hardware and more importantly it would not be flexible, as address assignment needs to be fixed at architecture design time. In contrast, our approach works for a given hardware and does not require hardware modifications. Furthermore, we can also tune our cache-to-memory controller mappings to strike the right balance between locality (short NoC distance) and memory parallelism (e.g., going to the nearest memory-controller may not always be a good idea when memory parallelism dominates).

2. Importance of Off-Chip Data Accesses

We present in Figure 3 the contribution of off-chip data accesses to the total data accesses for a set of multithreaded applications in an 8×8 mesh-based manycore system with private L2s and page granularity of interleaving of physical addresses across memory controllers (details of our experimental platform will be given later). These results indicate that off-chip accesses can contribute, on an average, to about 22.4% of the total (dynamic) data accesses. While this number is not very high, the impact of these accesses on execution time can be much higher. It is important to observe that an off-chip accesses that does *not* access the nearest memory controller can cause three types of performance problems in our architecture. First, such an accesses spends more time in the on-chip network compared to an access that goes to the nearest memory controller. Second, spending more time in the network means more contention for on-chip accesses (cache and coherence traffic) as well. And third, such an access can also end up spending more time in the memory queue.

[2] The impact of (2) can be reduced through smart memory controller scheduling algorithms as proposed in [7], which is orthogonal to our work.

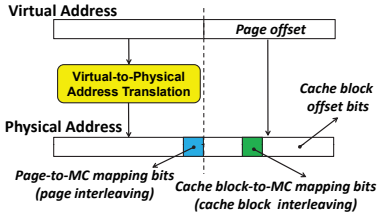


Figure 5: Virtual-to-physical address translation and the interpretation of the physical address bits.

Figure 4 quantifies the impact of an *optimal scheme* (implemented in our simulator) on these three types of latencies discussed above as well as on execution time. This optimal scheme assumes that each memory request originating from any core (in the case of private caches) or L2 controller (in the case of shared caches/SNUCA) always accesses the nearest MC and does not incur any additional latency (due to bank contention). In other words, this optimal version enjoys both high locality and high memory-level parallelism. The results presented in Figure 4 show that the optimal scheme reduces, on average, the network latency of on-chip memory accesses by 20.8%, the network latency of off-chip accesses by 68.2% and the memory latency of off-chip accesses by 45.6%. These improvements collectively translate to an average execution time improvement of 19.5% over the default case. In other words, optimizing off-chip accesses has significant potential for improving performance, and the remainder of this paper discusses a compiler-based strategy and evaluates how close it can come to the savings achieved by the optimal scheme quantified above.

3. Background

Data-to-Core mapping: There are many ways to parallelize a loop-intensive application. For example, in OpenMP's static scheduling, the computations are evenly divided into contiguous chunks and assigned to the cores in order. An application parallelized in this way indirectly indicates the portion of data elements that will be accessed by each core, which is called the "Data-to-Core mapping" in this paper.

Data-to-MC mapping: The underlying Data-to-MC mapping tells how physical addresses are distributed across MCs. In a system with N memory controllers, $log(N)$ bits of the physical address are used to determine the *mapping of data to memory controllers*. Depending on where these bits are taken from the physical address, different data interleavings (physical address-to-memory controller mapping) can be implemented in hardware. Figure 5 shows the virtual-to-physical address translation and the usage of physical address bits in the target architecture. There are two alternative virtual-to-physical address translations. In the first one, the first $log(N)$ bits following the cache block (line) offset bits are used, which results in cache block interleaving across memory controllers. In other words, if a cache block with address B_i is mapped to MC j, the next cache block at address B_{i+1} will be mapped to MC $j+1$. This can be expressed as $j = B_i \ mod \ N$. In this *cache-line interleaving*, the bits used for memory controller selection are *not* modified by the virtual-to-physical address translation process. Therefore, one can statically determine the memory controller where a particular data element will reside by simply examining its virtual address. Alternatively, memory controller selection bits can be taken from the first bits after the page offset field. In this case, the granularity of interleaving will be a *page*. Our compiler-based off-chip access optimization strategy is evaluated with both cache-line and page interleavings of physical addresses.

4. Framework Overview

Our approach primarily targets data-parallel *affine programs*, where loop bounds and array indices (subscript expressions) are affine functions of the enclosing loop indices and loop-independent variables/constants. However, our approach can also handle certain irregular references in the program being optimized by approximating them using affine expressions and profile data, as will be discussed later in the paper. We also assume that all array sizes are known before our approach is applied. If this information is not available, we profile the code to derive it. Also, most of the loop nests in our target codes are data parallel with similar data accesses rates across different cores. Our approach does not do anything specific for conditionals. It conservatively assumes that both branches can be taken (equally likely). This can potentially reduce the benefits (as conflicting references in different branches may prevent a unique/ideal data layout selection), but we observed this to happen in only 1 application in our experimental suite.

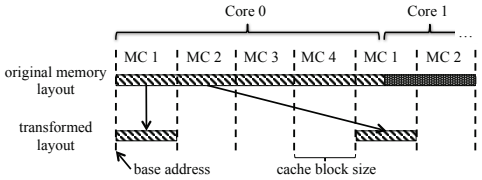


Figure 6: Using layout transformation to localize off-chip memory accesses.

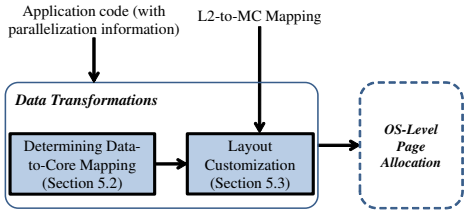


Figure 7: High-level view of our proposed approach.

Consider the private L2 case shown in Figure 2a as an example. A Data-to-Core mapping is illustrated in Figure 6, where the underlying Data-to-MC mapping is also given. In the original layout, the off-chip access requests to the data elements mapped to Core 0 will be sent to all four MCs, while the desired memory controller for these accesses should be MC1. That is to say, ideally, all the off-chip accesses to the data elements mapped to a core should be sent to the MC that is the *closest* to this core, but the underlying Data-to-MC mapping prevents this from being realized. One way to solve this problem is to *rearrange/reorder* data elements on the virtual address space (called memory layout transformation) such that each data element can be mapped to the desired MC. For example, the transformed layout at the bottom of Figure 6 successfully eliminates the mismatch mentioned above: the data elements accessed by Core 0 are always mapped to MC1. How to generate this kind of layout is the main task of our proposed framework.

Our off-chip access optimization strategy can be applied to hand-parallelized or compiler-parallelized codes, and consists of two steps, as shown in Figure 7. In the first step, which is called *Determining Data-to-Core Mapping*, we identify the data/data regions accessed by each core/thread locally, and determine the mapping between data elements and cores. In the next step, which is called *Layout Customization*, we reorganize the layout of the data elements accessed by each core on the virtual address space such that the off-chip access requests of these data elements can always be sent to the "most desirable" MC. To achieve this, we employ a concept called *L2-to-MC mapping* which is considered as an input for the second step and provided by the user/programmer. This mapping is provided through the command line, as the number of cores in a cluster. Figure 8 shows two sample L2-to-MC mappings with four MCs. The nodes covered by the same shaded area form a *cluster*. All the off-chip access requests from the L2s connected to the nodes in the *same* cluster will be sent to the *same* set of MCs. Note that, not any L2-to-MC mapping is valid. In order to obtain a

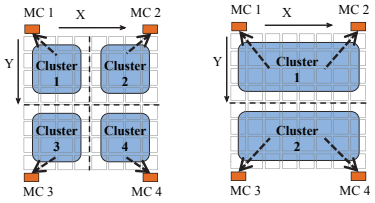


Figure 8: Different L2-to-MC mappings. The data accessed by the cores in a cluster are allocated/accessed from the corresponding controllers (indicated by arrows).

```
for (i = 2; i<N-1; i++)
  for (j = 2; j<N-1; j++)
    Z[j][i] = Z[j-1][i] + Z[j][i] + Z[j+1][i];
```

(a) Original parallel code

```
for (i = 2; i<N-1; i++)
  for (j =2; j<N-1; j++)
    Z'[i][j] = Z'[i][j-1] + Z'[i][j] + Z'[i][j+1];
```

(b) Code fragment after determining the Date-to-Core mapping

```
for (i = 2; i<N-1; i++)
  for (j = 2; j<N-1; j++)
    Z"[j/(k*p)][((i/b)%q)/2][((i/b)%q)%2][j%(k*p)] =
          Z"[(j-1)/(k*p)][((i/b)%q)/2][((i/b)%q)%2][(j-1)%(k*p)]
          + Z"[j/(k*p)][((i/b)%q)/2][((i/b)%q)%2][j%(k*p)] +
          + Z"[(j+1)/(k*p)][((i/b)%q)/2][((i/b)%q)%2][(j+1)%(k*p)];
```

(c) Code fragment after layout customization

Figure 9: Sample code.

desired layout, we require, first, each cluster must contain an equal number of cores, and second, each cluster should be assigned to an equal number of memory controllers. This is due to our strip-mining and permutation transformations which we discuss later.

Each of the L2-to-MC mappings illustrated in Figure 8a and Figure 8b has its own advantages and drawbacks. Specifically, in (a), since all the off-chip access requests from the same cluster will always be sent to the "nearest" memory controller, the average distance that these requests need to travel is less than the case in (b); that is, (a) *localizes* off-chip accesses better than (b). However, if too many off-chip access requests are sent by a group of cores to the same MC simultaneously, the time required to process these requests can significantly degrade application performance. In this case, (b) could be a better choice since now the requests sent from one core will be processed by two memory controllers. This reduces the pressure on each memory controller to some extent and, therefore, helps with processing of off-chip accesses. In other words, (b) can enjoy better memory level parallelism in processing the off-chip accesses. To summarize, different L2-to-MC mappings exhibit different locality vs. parallelism tradeoffs. The user can specify any L2-to-MC mapping as long as the specified mapping satisfies the two constraints discussed above. Our approach will then try to localize the off-chip accesses based on this L2-to-MC mapping. In the rest of our discussion, we mainly focus on the case where the underlying Data-to-MC mapping is at the cache-line interleaving granularity.

As stated above, our implementation takes L2-to-MC mapping as input from the user. In principle, there are lots of potential valid L2-to-MC mappings. As a result, automatically determining the best L2-to-MC mapping is very difficult in practice. However, given a set of potential mappings (provided by the user), the compiler may be able to identify the best one from among them. In fact, we implemented a compiler analysis that identifies, given a set of L2-to-MC mappings, the most effective one by weighing two metrics: (1) distance-to-MC and (2) memory-level parallelism (MLP). Our preliminary evaluation of this compiler analysis shows that it can successfully favor the mapping in Figure 8b over the one in Figure 8a in two of our applications, *fma3d* and *minighost*. That is, while it is difficult to determine an ideal mapping fully automatically, the compiler may be able to choose the best mapping from a given set of mappings.

5. Off-chip Access Localization

5.1 Iteration Space, Arrays, and Hyperplanes

The *iteration space* of an m-level loop nest can be viewed as an m-dimensional polyhedron bounded by the loop bounds. Each iteration (each point in this polyhedron) can be expressed by an *iteration vector* $\vec{i} = (i_1, i_2, \cdots, i_m)^T$, where $i_1, i_2, \cdots, i_m$ are the loop iterators. Similarly, the *data space* of an n-dimensional array can be viewed as an n-dimensional polyhedron bounded by the array bounds. Each data element (each point in this polyhedron) can be expressed by a *data vector* $\vec{a} = (a_1, a_2, \cdots, a_n)^T$, where $a_1, a_2, \cdots, a_n$ are the array indices. The mapping between iteration space and data space is represented by the array references, which can be written as $\vec{r} = A\vec{i} + \vec{o}$, where A is an $n \times m$ constant matrix called the *access matrix*, and $\vec{o}$ is an $n \times 1$ constant vector. For example, the reference $A[i_1][2i_2 + 1]$ in a two-level loop nest (with loop iterators i_1 and i_2) can be expressed as: $\vec{r} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, where $\vec{i} = (i_1, i_2)^T$. When $\vec{i} = (1, 2)^T$, we have $\vec{a} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = (1, 5)^T$.

A *hyperplane* h in an k-dimensional polyhedron is a flat subset of $(k-1)$ dimensions, which can be characterized by a $k \times 1$ vector $\vec{h} = (h_1, h_2, \cdots, h_k)$ and a constant c. In our context, $\vec{h}$ is called the *hyperplane vector* and c is called the *hyperplane offset*. Any point $\vec{p} = (p_1, p_2, \cdots, p_k)^T$ on h satisfies $\vec{h}\vec{p} = c$. In this work, we focus on parallelized *affine loop nests* where the array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-independent variables. One of the frequently-used loop parallelization and distribution schemes in such codes is called *block-cyclic distribution*, where, with one thread per core, an m-dimensional iteration space is *evenly* partitioned into chunks (the last chunk may have a smaller number of iterations) by w set of parallel hyperplanes. For simplicity, in the rest of our discussion, we assume w=*1*, i.e., there is only one set of parallel hyperplanes that is orthogonal to the u-th dimension of the iteration space, which is called the *iteration partition dimension*. Therefore, the hyperplane vector that represents this set of parallel hyperplanes is a $1 \times m$ unit vector (denoted as $\vec{h}_I$) in the form of $(\underbrace{0, 0, \cdots, 0}_{u-1}, 1, 0, \cdots, 0)$. Figure 9(a) gives an example, where the parallelized loop is marked as bold, and the corresponding iteration distribution is illustrated in Figure 10(a).

5.2 Determining Data-to-Core Mapping

In this step, we isolate the data elements touched by different threads/cores. Specifically, we *evenly* partition the data space into *data blocks* by a set of parallel hyperplanes orthogonal to a given dimension, such that, most of the data elements in a given data block are accessed by the same thread/core. This dimension is called the *data partitioning dimension*. If we specify the v-th dimension as the data partitioning dimension,[3] then the set of parallel hyperplanes that partition the transformed data space can be represented by the hyperplane vector $\vec{h}_A$, which is a $1 \times n$ unit vector in the form

[3] In an attempt to reduce the padding overhead mentioned in Section 5.3, this dimension is always chosen to be the slowest-varying dimension (e.g., first dimension in a row-major memory layout).

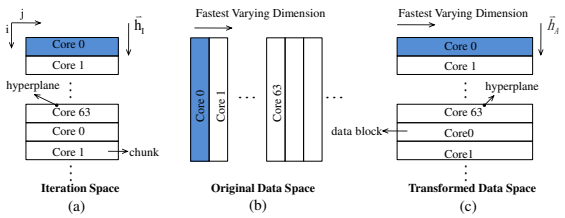


Figure 10: Determining the Data-to-Core mapping for array Z in Figure 9(a).

of $(\underbrace{0, 0, \cdots, 0}_{v-1}, 1, 0, \cdots, 0)$. Figure 10(c) illustrates a transformed data spaces for array Z in the parallel code shown in Figure 9(a), where the partitioning dimension is specified as the slowest-varying dimension, i.e., v = 2. It should be noted that, in the original data space, shown as Figure 10(b), although most data elements in the same data block are accessed by the same thread/core, these data blocks are not formed by the set of parallel hyperplanes orthogonal to the v-th dimension (data partitioning dimension). Therefore, they do not form a valid Data-to-Core mapping. We employ a *unimodular layout transformation* to find a partitioning, which can be characterized by a *transformation matrix* U [8]. Each data vector $\vec{a}$ in the original data space is mapped to a unique vector $\vec{a}'$ in the transformed data space, i.e., $\vec{a}' = U\vec{a}$, and the array reference $\vec{r}$ is changed accordingly to $\vec{r}'$, i.e., $\vec{r}' = U\vec{r}$.

Single Array Reference: Let us first discuss the case where there is only one array reference $\vec{r}$, how to obtain the transformed data space by determining the entries of the transformation matrix U. Any two iterations $\vec{i}_1$ and $\vec{i}_2$ that reside on a hyperplane defined by a hyperplane vector $\vec{h}_I$ should satisfy:

$$\vec{h}_I(\vec{i}_1 - \vec{i}_2) = 0. \tag{1}$$

Let $\vec{i}_{1,2} = \vec{i}_1 - \vec{i}_2$, and $\vec{e}_i$ be an $m \times 1$ unit vector, where 1 appears at the i-th position; the solution set for $\vec{i}_{1,2}$ can be expressed as $\sum_{i=1, i\neq u}^{m} k_i\vec{e}_i$, where k_i is an arbitrary integer. Similarly, in the transformed data space, the two data elements $\vec{a}'_1$ and $\vec{a}'_2$ accessed by these two iterations through $\vec{r}'$ should always reside on the same hyperplane defined by the hyperplane vector $\vec{h}_A$. Therefore, we have $\vec{h}_A(\vec{a}'_1 - \vec{a}'_2) = 0$. Assuming $\vec{r} = A\vec{i} + \vec{o}$ and $\vec{a}' = U\vec{a}$, we further have:

$$\vec{g}_v A(\vec{i}_1 - \vec{i}_2) = 0, \tag{2}$$

where $\vec{g}_v$ is the v-th row vector of U, and v is the data partitioning dimension. In other words, any two iterations $\vec{i}_1$ and $\vec{i}_2$ that satisfy Eq. (1) must also satisfy Eq. (2). As a result, we have $\vec{g}_v A\vec{e}_i = 0$, where $1 \leq i \leq m$, $i \neq u$. Let B be a matrix that consists of all the column vectors of A except the u-th one (called a *submatrix* of A). Then, the last expression can be re-written as:

$$B^T\vec{g}_v^T = 0. \tag{3}$$

The above homogeneous linear system can be solved by Integer Gaussian Elimination [9]. This also indicates that the desired transformation matrix U is completely determined by its row vector $\vec{g}_v$. In cases where the solver returns a non-trivial solution for $\vec{g}_v$, we determine the remaining $n-1$ row vectors such that U is unimodular. In the example of Figure 9(a), $u = 1$ and $B = (0, 1)^T$. If $v = 1$, the solution returned for $\vec{g}_1$ is $(1, 0)$; if $v = 2$, the solution returned for$\vec{g}_{21}$ is $(1, 0)$. Since we prefer the v-th dimension being the slowest-varying dimension ($v = 2$), the transformation matrix U is then determined as $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The transformed array indices are shown in Figure 9(b), and Figures 10(b) and 10(c) illustrate data space for array Z before and after the transformation, respectively.

Multiple Array References: Next, we study how to determine U in a global sense when there are multiple references to an array in different parallelized loop nests. Assuming that there are k different submatrices $B_1, B_2, \cdots, B_k$ for these references, based on Eq. (3), we have k homogeneous linear systems to solve, each corresponding to a submatrix. A unique $\vec{g}_v$ (v is the data partitioning dimension) that satisfies all these systems may or may not exist. That is, we may not always end up in a transformed data space where *all* the data elements on the hyperplane orthogonal to $\vec{h}_A$ are accessed by a single thread. To address this potential problem, our strategy is to assign a *weight* to each submatrix to determine the most "beneficial" linear system that, when solved, satisfies the majority of references. Specifically, assuming that there are s references in the given set of loop nests that have the same submatrix B_i, then the weight of B_i, denoted as $W(B_i)$, is the total number of occurrences of theses references, i.e., $W(B_i) = \sum_{j=1}^{s} n_j$, where n_j is estimated by the product of the trip counts (the number of iterations) of the loops that enclose the said reference.

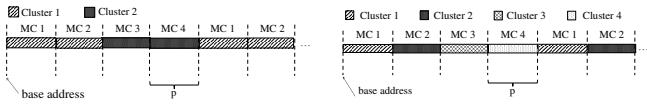
5.3 Layout Customization

It is important to emphasize that we apply our layout transformation to each array separately. The formal compiler algorithm will be given later. So far, we have determined the Data-to-Core mapping and assumed that the user provides a valid L2-to-MC mapping. If it is the *local L2* cache that sends the off-chip access request to the MC, which is true for the case of private L2s,[4] then these two mappings actually indicate a *desired Data-to-MC mapping*. However, because of the *underlying Data-to-MC mapping* mechanism, this desired mapping may *not* always be realized directly. We address this problem (of implementing the desired Data-to-MC mapping) by *customizing* the layout obtained in Section 5.2 according to the L2-to-MC mapping specified by the user.

We employ two layout transformation techniques, *strip-mining* and *permutation*, originally used in works such as [10], for an entirely different purpose. Let N_i denote the size of the array along the i-th dimension. Then, strip-mining with a block size of s transforms this dimension into two dimensions with the size of N_i/s and s; and a reference $\vec{r} = (\cdots, r_i, \cdots)^T$ becomes $(\cdots, r_i/s, r_i\%s, \cdots)^T$. After this transformation, r_i/s and $r_i\%s$ can be used to identify the index of the block and the offset within a block, respectively. In comparison, permutation switches the positions of two dimensions in an array to change the data placement in the linear memory space; and a reference $\vec{r} = (\cdots, r_i, \cdots, r_j, \cdots)^T$ becomes $(\cdots, r_j, \cdots, r_i, \cdots)^T$. The run-time overhead associated with these transformations comes mainly from the division and module operations employed. In our current implementation, we use the techniques proposed in [10] and [5] to reduce these costs. We also employ *padding* [11] to keep the base addresses of arrays aligned to the desired memory controller, and align data elements within an array to make the strip-mined dimension divisible by s. Next, we discuss how to use the above techniques to customize the layout for private L2s and shared L2.

Private L2 Case: For a manycore system with private L2 caches, our first step is to transform original reference (obtained from Section 5.2) $(\cdots, r_v, \cdots, r_n)^T$ to $(\cdots, R(r_v), \cdots, r_n)^T$, where the v-th dimension is the data partitioning dimension identified in Section 5.2, and $R(r_v)$ is a set of transformed array indices that can be used to identify the cluster for each data element accessed through this reference. Recall that, in order to identify the data blocks touched by each core/thread, the data space is partitioned by a set of parallel hyperplanes orthogonal to this dimension. Therefore, $R(r_v)$ can be obtained by performing multi-

[4] As will be shown later, one can use layout transformation techniques to make this true for the shared L2 case as well.



(a) Customized layout for L2-to-MC in Figure 8b. In this example, $k = 2$.

(b) Customized layout for L2-to-MC in Figure 8a. In this example, $k = 1$.

Figure 11: The customized layouts for the L2-to-MC mappings shown in Figure 8.

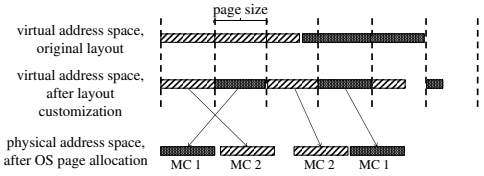


Figure 12: Changing Data-to-MC mapping through the OS support.

ple strip-minings on the v-th and the newly-generated dimensions. Specifically, assuming that we have $c_x \times c_y$ number of clusters and each cluster has $n_x \times n_y$ number of cores, then $R(r_v)$ can be expressed as $(((r_v/b)/(n_y * c_y * n_x))\%c_x, ((r_v/b)/n_y)\%c_y)$, where b is the data block size, and the subscript "x" and "y" indicate the number along the X-axis and Y-axis, respectively. Note that, here, $((r_v/b)/n_y)\%c_y)$ is the general form to calculate the index along the y-th dimension, and stripmining is performed three times, namely, in r_v/b, $(r_v/b)/n_y$, and $((r_v/b)/n_y)\%c_y$.

Next, based on this transformed layout, we perform permutation and strip-mining on the fastest-varying dimension of the target array (e.g., the last dimension in a row-major layout) to obtain an interleaved layout. Assuming that k is the number of MCs assigned to each cluster (see Figure 8 for sample assignments), and p is the cache block (line) size (in terms of the number of data elements), by transforming the array reference into the form of $(\cdots, r_n/(k * p), R(r_v), r_n\%(k*p))^T$, every consecutive $k*p$ data elements will be accessed by the cores in the same cluster in a round-robin fashion, and the off-chip accesses made for $k * p$ consecutive data elements will be sent to the memory controllers indicated by the L2-to-MC mapping.[5] An example is given in Figure 11.

Shared L2 Case: In a shared L2 based system [1], all the on-chip and off-chip requests for a data element are issued by its home (L2) bank, not by its local bank. The home bank for each data element is determined by the underlying Data-to-L2 Bank mapping mechanism, which is similar to the Data-to-MC mapping explained in Section 3. However, in the case of cache-line interleaving, this introduces an additional problem: if we only focus on improving the *on-chip access locality*, although the distance between the requester of the data and the L2 home bank that owns the data could be minimized, doing so could degrade the *off-chip access locality* on the network, and vice versa. However, taking both on-chip and off-chip data localization into account is not trivial. For example, assuming that we have 4 MCs, then the first p data elements will be mapped to MC1, and the next p data elements will be mapping to MC2, and so on. Given the desired L2-to-MC shown in Figure 8a, the off-chip accesses to the data elements whose home bank is connected to Core 2 will be sent to MC2 instead of the desired one – MC1.

An important question at this point is whether there always exists a layout such that both the on-chip and off-chip accesses can be localized. Our answer to this question is no. Let $addr(\vec{a})$ represent the virtual address of a data element $\vec{a}$. Then, the home bank that issues the off-chip accesses to $\vec{a}$, denoted as id_{HB}, can be expressed as:

$$id_{HB} = addr(\vec{a})/p\%N, \quad (4)$$

where p is the cache block size and N is the total number of cores on the network. Similarly, the MC to which request for $\vec{a}$ is sent, denoted as id_{MC}, can be expressed as:

$$id_{MC} = addr(\vec{a})/p\%N', \quad (5)$$

where N' is the total number of memory controllers. Based on Eqs. (4) and (5), we have $N * \delta_1 + id_{HB} = N' * \delta_2 + id_{MC} = addr(\vec{a})/p$. In the L2-to-MC mapping shown in Figure 8a, we have $N = 64$, $N' = 4$, which indicates that $4 * (16 * \delta_1 - \delta_2) = id_{MC} - id_{HB}$, where δ_1 and δ_2 are integers. To satisfy this equation, id_{MC} and id_{HB} must *always* be multiples of 4, which may not be the case. To solve this problem, our strategy is to generate a localized layout for the on-chip accesses first, and then try our best to localize the off-chip accesses. One could also first generate the layout localized for off-chip accesses and then try to localize the on-chip accesses as much as possible. Specifically, we first generate a layout with the property that most of the data accesses are local, i.e., home bank of each data block is the one connected to the core by which this data block is accessed. Similar to case of the private L2 based system, we transform an array reference $(\cdots, r_v, \cdots, r_n)^T$ into the form of $(\cdots, r_n/p, R'(r_v), r_n\%p)^T$, where $R'(r_v)$ is a set of transformed indices that can be used to identify the L2 bank, and can be expressed as $R'(r_v) = (r_v/b)\%N$. After this transformation, every consecutive p data elements will be accessed by the same core in a round-robin fashion.

We then optimize the off-chip accesses under this new layout as follows. If a data element resides on a memory address/location where the mapped MC is not adjacent to the desired MC, we skip this address and move that data element to the next closest memory address such that the mapped MC is adjacent to the desired MC and all the data elements originally at or beyond this address will be moved forward accordingly. This requires us to replace the original reference $\vec{a} = (\cdots, r_n/p, R'(r_v), r_n\%p)^T$ with $\vec{a'}$ where: $\vec{a}' = \vec{a} + (0, \cdots, 0, \delta * p)^T$. Here, δ is a counter that will be inserted at the loop level where the target array is accessed. It will be increased by 1 (i.e., $\delta = \delta + 1$) if $id_{MC}(\vec{a}) \in C$, where $id_{MC}(\vec{a})$ gives the desired MC for $\vec{a}$, and C is the set of memory controllers that are not adjacent to this MC. Figure 9(c) gives the code transformed using our layout customization.

Page Interleaving: If the page-interleaving (of physical address across MCs) is adopted, one can still apply the optimization scheme discussed so far by simply changing p to the page size. However, there is one additional problem that needs to be addressed: now the bits used for memory controller selection are modified by the OS. Therefore, the compiler-guided layout transformation scheme discussed so far cannot directly enforce the desired Data-to-MC mapping. Instead, we need the OS to be involved (by changing the existing page allocation policy). Figure 12 shows an example where, after layout transformation, with an assist from the OS, the data elements accessed by the same set of cores (marked using the same texture) are assigned to the physical pages mapped to the same memory controller.

In the modified page allocation policy, which can be implemented by using *madvise()* in Linux, when assigning a physical address space to a page with the given virtual address, we require the assigned physical address to belong to the desired memory controller. For example, in Figure 12, assuming that we only have two memory controllers, the data elements in the first memory chunk (page) should be mapped to the physical addresses whose Page-to-MC mapping bit (see Figure 5) will be set to 0, and the data elements in the second memory chunk (page) should be mapped to the physical addresses whose Page-to-MC mapping bit will be set to 1. In other words, the modified page allocation algorithm assigns

[5] We bind each thread to a core through a system call to ensure that the order of the cores is consistent with the order of memory controllers in the target two-dimensional grid (see Figure 2a).

Algorithm 1 Layout_Transformation

INPUT: Granularity: page/cache block, L2 cache attribute: shared/private, Desired L2-to-MC Mapping
OUTPUT: Data references with optimized data layout

```
1: //single reference function
2: function DATA-TO-CORE-MAPPING(access matrix A, u, v)
3:     B ← A − u_th column
4:     B^T g_v^T = 0
5:     B_lower ← Gaussian_Elimination(B^T)
6:     g_v^T ← Forward_Substitution(B_lower g_v^T = 0)
7:     (g_1^T, g_2^T, ..., g_n^T) ← Unimodular_Layout_Transformation(g_v^T)
8:     U ← (g_1^T, g_2^T, ..., g_n^T)^T
9:     if det(U) != ±1 then
10:        //Check if U is unimodular
11:        H ← Hermit_Normal_Form(U)
12:        U ← H^{-1} U
13:    end if
14:    return U
15: end function
16: for All arrays from the first to the last do
17:    Choose one array as A
18:    if multiple_references then
19:        //Multiple references(A_1, A_2, ..., A_k)
20:        for i from 1 to k do
21:            n_j ← Total access number of each reference
22:            s ← Number of references with same A_i
23:            W(A_i) ← Σ_{j=1}^{s} n_j
24:        end for
25:        find A_i with maximum W(A_i)
26:        U = DATA-TO-CORE-MAPPING(A_i)
27:    else
28:        U = DATA-TO-CORE-MAPPING(A)
29:    end if
30:    r'  ← U r
31:    r ← r'
32:    //r = (..., r_v, ..., r_n)^T
33:    //now we have the reference r optimized for on-chip access and we begin
       layout customization
34:    if cache block then
35:        p ← block_size
36:        k ← MCs per cluster
37:        //From user input L2-to-MC Mapping
38:        if private L2 then
39:            R(r_v) ← (((r_v/b)/(n_y * c_y * n_x))%c_x,
           ((r_v/b)/n_y)%c_y)
40:            r ← (..., R(r_v), ..., r_n)^T
41:            Permutation and Strip_mining(r)
42:            r ← (..., r_n/(k * p), R(r_v), r_n%(k * p))^T
43:        else if Shared L2 then
44:            R'(r_v) ← (r_v/b)%N
45:            r ← (..., R'(r_v), ..., r_n)^T
46:            Permutation and Strip_mining(r)
47:            r ← (..., r_n/p, R'(r_v), r_n%p)^T
48:            //optimize off-chip access
49:            for Each reference a = (..., r_n/p, R'(r_v), r_n%p)^T do
50:                δ ← 0
51:                while id_MC(a) ∈ C do
52:                    δ ← δ + 1
53:                end while
54:                a'  ← a + (0, ..., 0, δ * p)^T
55:            end for
56:        end if
57:    else if page then
58:        p ← page_size
59:        OS-assisted customization which is similar with cache line customization
60:    end if
61: end for
```

physical pages in a round-robin fashion to guarantee the desired distribution of pages across memory controllers.

At this point, there are two important issues that need to be clarified. First, generating virtual memory layout at the beginning simplifies the modification to the page coloring algorithm. This is because, after this transformation, we can easily obtain the desired Page-to-MC mapping bits for each data element based on the Data-to-MC mapping, which is indicated by the Data-to-Core mapping and L2-to- MC mapping. Second, if the memory space attached to the specified MC is full, an alternate MC is selected and the page is placed into the space managed by that controller. Consequently, our approach does not increase the number of page faults, i.e., the available physical memory space is fully utilized.

5.4 Handling Indexed Array Accesses

As stated earlier, our approach mainly targets application programs with affine references. However, in many data-intensive applications, there are data array accesses made through index arrays. For example, in *hpccg*, the core computation includes a sparse matrix-vector multiplication (SpMV), and similarly, in *minimd*, we have indexed array accesses. Our current implementation can handle such codes as follows. The idea is first to use profiling and extract the "dense access patterns" of the indexed references (e.g., in SpMV, the access pattern to the array in two-dimensional representation). In the case of CRS representation for example, this is done by determining the contents of the row-pointer an column-index arrays. Then, we generate an affine function (with enclosing loops) that approximates the addresses generated by the reference. This approach, whose details are beyond the scope of this work, can result in over-approximation (the resulting affine expression generates more addresses than the original – indexed array – reference) or under-approximation (and in fact increasing the number of enclosing loops can increase the accuracy of the approximation). Still, we can use the result of this approximation to perform our data layout restructuring, as over- or under-approximation does *not* create a correctness issue but can only lead to a performance issue. We want to make it clear however that, for some references, the inaccuracy resulting from approximation can be very bad (e.g., more than 30%), in which case our implementation simply does not optimize those references.

5.5 Algorithm

The formal algorithm for our data layout transformation is given as Algorithm 1, which changes the data layouts of arrays and modifies the array references in the code. In optimizing the layout of each array, we consider *all* the references to it in all loops in the program. The outermost loop (line 16) iterates over all arrays in the program. Lines 17-32 capture the algorithm for determining Data-to-Core Mapping, which calls the function DATA-TO-CORE-MAPPING (defined in lines 1-15). In this phase, we try to find a set of parallel hyperplanes (on the data space of the arrays) orthogonal to the data partitioning dimension (defined in Section 5.2 or specified by the user) such that, most of the data elements in a partitioned data block (formed by those hyperplanes) are accessed by the same thread/core. In this way, the data elements touched by different threads/cores are isolated in the transformed data space.

Lines 34-60 illustrate our algorithm of layout customization. In this phase, permutation and strip-mining can be used multiple times. In the case of private L2 cache (lines 38-42), we change the transformed array indices (by using permutation and strip-mining) such that the data elements accessed by each thread/core are mapped to the cores in the same cluster in a round-robin fashion (recall that each cluster corresponds to a memory controller). In other words, these data elements are local to the threads/cores (in the same cluster) that uses them for computation. In case of shared L2 cache (lines 43-56), in addition to the transformation applied to the private-L2 case, we further change the layout to ensure that the data elements are NOT mapped to the cluster that is far from the threads/cores that use these data for computation (note that, in shared-L2 case, mapping all the data to the threads/cores in the same cluster that uses them for computation is impossible).

It is important to note that, in our implementation, there is no reason why the case of references (to the same array) from different nests should be treated any different than the case of references in the same nest. This is because our approach simply attaches a weight to each layout preference, and this weight is a function of the iteration count of the nests that enclose the corresponding references. As a result, if, say, two references to the same array in different nests prefer the same data layout, we increase the weight of the corresponding "layout preference" accordingly. Since the weights are accumulated (for each layout preference), our algorithm does not care which nest(s) they are originating from.

6. Experiments

6.1 Experimental Setup and Applications

The proposed layout optimization is implemented using the Open64 infrastructure [12] version 4.2.4, as a source-to-source translator. During the compilation process, a loop transformation-guided by array dependence analysis restructures the intermediate code for improving both parallelism and data locality (cache performance). Our proposed scheme is inserted as an additional pass after this transformation and function inlining.

Apart from the user input that specified L2-to-MC mapping, the entire approach is fully automated within the compiler. If the user does not specify an L2-to-MC mapping, we use, by default, the mapping given in Figure 8(a). All the experiments are carried out using the GEM5 simulator [13], with our modified Linux kernel that implements the customized page allocation policy explained in Section 5.3.

The important simulation parameters and their default values are given in Table 1. We present results from a set of multithreaded programs from SPECOMP [14] and Mantevo suites [15]. We used all applications from SPECOMP except *equake*, we could not execute in our simulator due to a memory error. For the SPECOMP applications, we used the large input sets. The input sizes of the remaining applications ranged between 124.1 MB and 1.9 GB. In the default configuration (Table 1), all these applications run one thread per core and, in each experiment, we run only one multithreaded application (later we present results with different combinations of our applications). Also, our default L2-to-MC mapping is the one shown in Figure 8a, with private L2s (later we present results with shared L2 as well).

Table 2 gives for each benchmark the percentage of arrays that have been optimized using our approach, as well as the percentage of array references that have been satisfied by the layout transformation chosen for the corresponding array.[6] It should also be mentioned that, while the fraction of data shared by two or more threads is not very high in these applications (averaging on 14%), accesses to the shared data constitute a much larger fraction (around 31%) of the total data accesses. *fma3d* and *minighost* are two applications with the highest inter-core data sharing. As will be discussed later, this is why they prefer mapping M2 (in the Figure 8b) over the default one (M1).

It is important to emphasize that both the original application programs as well as their optimized versions (using our approach) are compiled with the same node compiler using the highest level of optimization, enabling basically all major loop restructurings for maximizing cache performance such as loop permutation and iteration space tiling. Further, we observed in our experiments that, the impact of our approach on the last-level cache misses was within 1%. Also, the results presented in the rest of this section include all the overheads brought by our layout transformation (e.g., those due to strip-mining and padding). We observed that these overheads constitute about 4% of the total execution time.

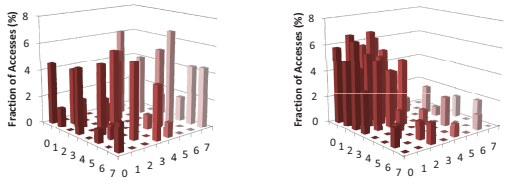
[6] The reason why we could not transform some arrays is because they use pointer accesses or index array accesses which could not be approximated by our approach.

Table 1: The simulated configuration.

Parameter	Default Value
Cores and Caches	
Processor	two-issue, SPARC processor
Data/Instr. L1	16 KB (per node), 64 byte lines, 2 ways
L2	256KB (per node), 256 byte lines, 16 ways
L1, L2 and per hop latencies	2, 10, 4 respectively
NoC	
Size	8×8 two-dimensional mesh
Delays and Routing	16B links, 2-cycle pipeline, XY-routing
Memory System	
Number of Memory Controllers	4 [placed into 4 corners of the mesh]
Scheduling Policy	FR-FCFS [16]
Capacity	4GB
Device Parameters	Micron MT47H64M8 DDR3-1600 timing parameters [17], 4 banks/device, 16384 rows/bank, 512 columns/row
Row Buffer Size	4KB [same as page size] 4 active row buffers per DIMM
Optimization Parameters	
Interleaving Unit	4KB [same as page size] or 256 Bytes [same as L2 cache line size]
L2-to-MC mapping	as shown in Figure 8a

Table 2: Percentage of arrays optimized (second column) and array references satisfied (third column).

wupwise	95%	76%
swim	92%	79%
mgrid	88%	80%
applu	95%	81%
galgel	83%	76%
art	91%	83%
ammp	55%	69%
fma3d	91%	78%
apsi	96%	82%
gafort	85%	80%
minighost	76%	83%
hpccg	68%	76%
minimd	88%	84%



(a) Before optimization. **(b)** After optimization.

Figure 13: Impact of off-chip access localization. The bars capture the accesses to MC1 in Figure 8a.

6.2 Results with Proposed Schemes

Impact of Off-Chip Access Localization: We first present two representative maps to illustrate how our approach changes the off-chip traffic destined to a memory controller. Our objective is to see how much off-chip access localization our approach achieves in practice. For this experiment, we focus on one of our applications, *apsi*. We present, in Figures 13a and 13b, the distribution of off-chip accesses, for the controller MC1 shown in Figure 8a. In both the graphs, the vertical axis plots the fraction of off-chip accesses made to this controller from each of our 64 nodes over the entire execution. The coordinates of the nodes in our two-dimensional grid are given by the remaining two axes in Figure 13. The first graph is for the original case and the second one is for the case when using our proposed strategy. The most important observation from these results is that, while in the original case the off-chip access requests to a memory controller come from all over the chip, in the optimized case, the accesses to the same memory controller are highly skewed towards the nearby cores. More details on requested distribution can be found in our technical report [18].

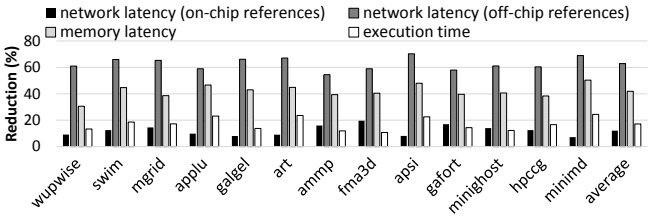


Figure 14: Results with page interleaving of physical addresses.

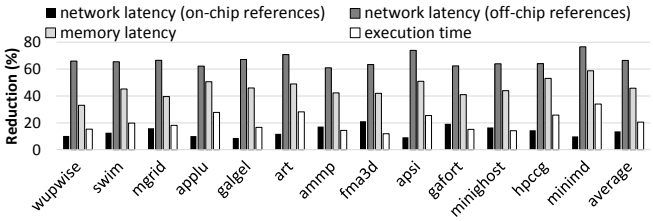


Figure 16: Results with cache-line interleaving of physical addresses.

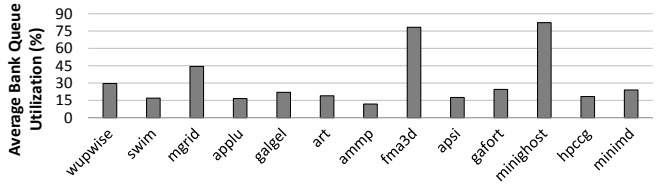


Figure 18: The bank queue utilization for our applications when using mapping M1 (Figure 8a).

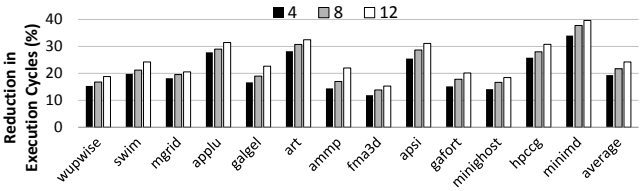


Figure 20: Sensitivity to the number of MCs.

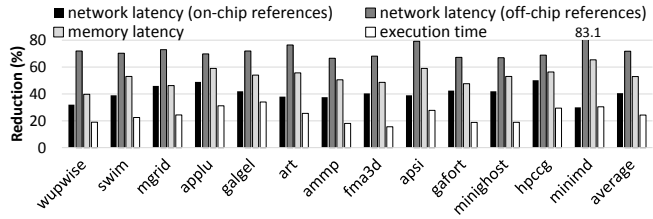


Figure 22: Evaluation of shared L2.

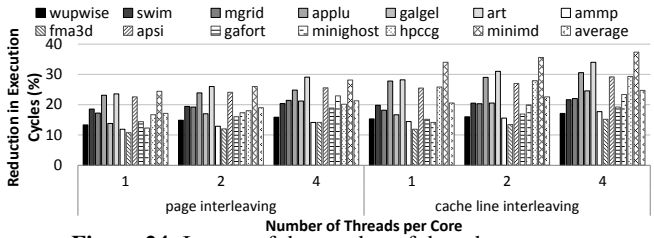


Figure 24: Impact of the number of threads per core.

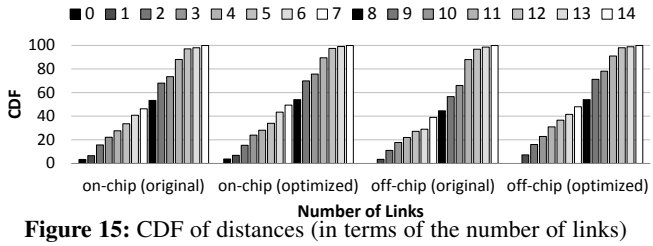


Figure 15: CDF of distances (in terms of the number of links) for both on-chip and off-chip accesses.

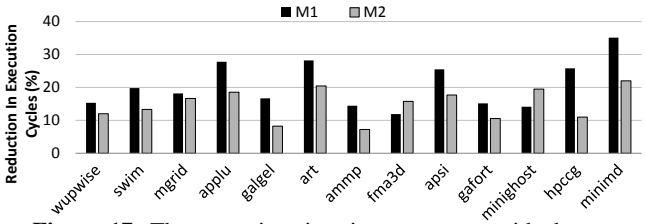


Figure 17: The execution time improvements with the mappings shown in Figure 8, under cache-line interleaving.

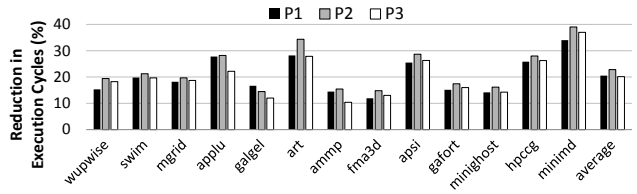


Figure 19: Results with different MC placements.

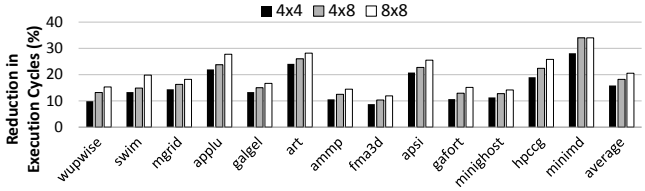


Figure 21: Sensitivity to the core count.

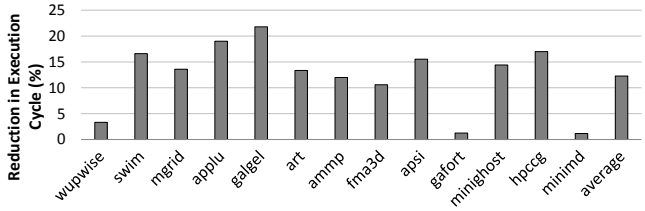


Figure 23: The results comparing the first-touch policy to our approach.

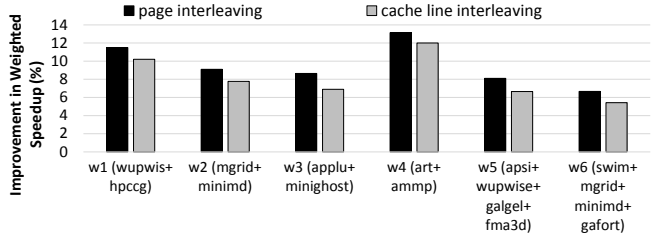


Figure 25: Evaluation of multiprogrammed workloads.

Performance Improvement: The impact of these localized accesses on application performance is quantified in Figure 14 (in the case of page-level interleaving). In this bar-chart, we present four results for each application, normalized to the original execution. The first bar gives the reduction in the network latency of on-chip accesses; the second bar captures the reduction in the network latency of off-chip accesses; the third bar gives the memory latency reduction for off-chip accesses (as a result of the reduction in queuing latency); and the last bar shows the reduction in execution time. The average improvements in these four metrics are 12.1%, 62.8%, 41.9% and 17.1%, in that order. While these savings are lower than the savings achieved by the optimal scheme described in Section 2, they still represent significant improvements, clearly showing that optimizing for off-chip accesses can be very important in NoC-based manycore systems.

To explain the improvements in on-chip network latencies, Figure 15 plots the CDF of the number of links traversed by on-chip and off-chip requests in both the original and optimized cases when considering all applications. Specifically, a bar corresponding to (x, y) in this graph indicates that $y\%$ of the requests (messages) traverses x or fewer links. One can make two important observations from this bar-chart. First, our approach reduces the number of links traversed by off-chip memory accesses significantly. For example, while in the original execution, only 22% of the requests use 4 links or less, in the optimized case the corresponding figure is 31% (i.e., more messages use fewer links). Clearly, one can expect this reduced number of links (per request) to reduce both off-chip and on-chip access latencies. However, when we look at the on-chip requests, the picture we see is different. More specifically, our approach does not have significant impact on the distance traveled by on-chip requests. Therefore, one can conclude that the improvements on on-chip access latencies (plotted in Figure 14) are mostly due to the reduction in network contention.

Evaluation of Cache-Line Interleaving: We next present results collected when the physical addresses are interleaved across cores at a cache line granularity. In Figure 16, all bars are normalized with respect to the case where the same granularity physical memory distribution is used without our approach. As in the case of Figure 14, the first bar in this plot gives the reduction in the network latency of on-chip accesses; the second bar captures the reduction in the network latency of off-chip accesses; the third bar gives the memory latency reduction for off-chip accesses; and the last bar shows the reduction in execution time, and the average improvements are 13.6%, 66.4%, 45.8% and 20.5%, in that order. In the rest of our experiments, unless stated otherwise, we use cache-line interleaving. Note that, while our percentage savings are better with cache-line interleaving, in the original codes, page interleaving generates better results than cache-line interleaving (3% on average).

Results with an Alternate L2-to-MC Mapping: Figure 17 gives the execution time improvement results with the mappings shown in Figure 8, where M1 and M2 in Figure 17 correspond to the mappings illustrated in Figure 8a and Figure 8b, respectively. It can be seen that, in most of our applications, going from M1 to M2 results in a reduction in our savings. This is because in these applications, off-chip access localization is more important than high levels of memory-level parallelism (MLP), and the banks connected to an MC are able to satisfy the memory level parallelism demand of a cluster (of 16 cores). In *fma3d* and *minighost* however, going from M1 to M2 generated better results. This is because these two applications exhibit much higher memory parallelism demand compared to the remaining ones. To explain this better, we present in Figure 18, the bank queue utilization (occupancy) for our applications when using M1 mapping (a higher value indicates a larger number of request waiting in the queue to be serviced). As can be observed, the utilization numbers are much higher with these two applications, compared to the remaining ones. Consequently, in these two applications, allowing a core to access more memory controllers (more banks) helps, though doing so increases the distance-to-memory controller for some requests. As a result, we can say that, assuming that the number of channels per memory controller is sufficiently large, M1 mapping, which associates each core with a single memory controller, should perform well in most cases.

Results with Different MC Placements: As pointed out in [19], modern flip-chip packaging allows sufficient escape paths from almost anywhere on the chip. This enables designers to explore different placements of memory controllers within the on-chip network. So far, we used the MC placement depicted in Figure 8a. Figure 26a and Figure 26b show two alternate MC placements (all with four MCs). The results with these placements are plotted in Figure 19 (P1, P2 and P3 correspond to the placements in Figure 8a, Figure 26a and Figure 26b, respectively). One can observe from these results that placement P2 generates slightly better results than others (an average improvement of about 20.7%), mainly because the average distance-to-controller is expected to be lower under this placement.

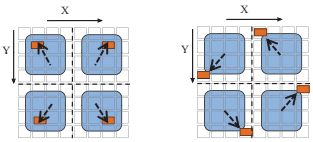
Results with Different MC Counts: To eliminate the potential effects that could come from the different memory controller placements, we increase the number of memory controllers based on the configuration depicted in Figure 8a. The configurations tested are shown in Figure 27a and Figure 27b, and have L2-to-MC mappings similar to the one shown in Figure 8a. The results with these new configurations are presented in Figure 20. We see that our approach generates higher savings with larger MC counts. This is mainly because a larger number of controllers lead to better memory parallelism within each cluster, which in turn increases the effectiveness of our approach (as we are not hurt by a degradation in memory level parallelism when off-chip accesses are localized).

Results with Different Core Counts: In Figure 21, we present results with 4×4 and 4×8 manycore systems with four controllers (one corner each, with the L2-to-MC mapping similar to Figure 8a). The results with our default configuration (8×8) are reproduced here for ease of comparison. We see an average improvement of 14% and 18% for the 4×4 and 4×8 configurations, respectively.

Results with Shared L2: Figure 22 shows the results collected when the available L2 space is managed as a shared SNUCA cache (with cache-line interleaving for both L2 and main memory). In general, the results shown here are similar to those given earlier in the case of private L2s (Figure 16). However, our approach achieves better improvements with shared cache, except in two benchmarks (*fma3d* and *minighost*). The average execution time improvement in the shared L2 case is about 24.3%. Overall, these results along with those presented earlier show that our approach works very well under both private and shared last-level caches.

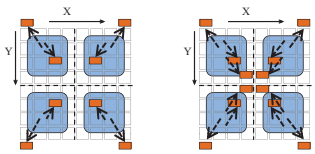
6.3 Comparison against the First-Touch Policy

So far, we evaluated our memory layout optimization approach under two hardware-based physical address interleaving schemes (page based and cache line based). We now compare our compiler-based approach to an OS-based first-touch policy [20]. In this policy, we still adopt the cluster concept but a page is allocated from an MCx if the first access to the page is from a node in cluster x. Figure 23 plots the average improvements our scheme brings (used along with page interleaving) over this first-touch policy. We see that, on average, our scheme generates 12.3% improvement over the first-touch policy. This is because the first-touch policy is essentially a greedy approach and assumes that most of the accesses to a given page in the entire execution would be from the same cluster where the first access to that page is made. According to



(a) (b)

Figure 26: Different MC placements with the same L2-to-MC mapping.



(a) (b)

Figure 27: Configurations with 8 and 12 MCs.

our results, this assumption does not hold except in three of our applications (*wupwise*, *gafort* and *minimd*).

6.4 Results with Varying Thread Counts per Core and Multiprogrammed Workloads

Recall that, in the results presented so far, we used one (multithreaded) application at a time, and ran always one thread per core. In this section, we report results with other options. Figure 24 gives the results when using more than 1 thread per core in executing a multithreaded application. The results shown clearly indicate that our approach brings higher improvements when the number of threads per core is increased. This is primarily because the degree of on-chip network contention increases dramatically, in the original case (without our optimization), when we have more threads, on the same core, of an application with long NoC distances. Our optimization is able to reduce the duration of this intensive contention by reducing the distance to data. So, for example, in *minighost* – under cache-line interleaving – our execution time saving reached around 20% when using two threads per-core. Figure 25 plots the results when we execute a multiprogrammed workload of multithreaded applications. It is important to make it clear that our approach is compiler based and does not do anything specific for multiprogrammed workloads. The goal in this section is just to quantify the impact of our approach when co-executing multiple applications in the same NoC-based manycore system. The x-axis in Figure 25 gives the workloads we formed and ran, and the y-axis shows the weighted speedup [21], a frequently-used metric for evaluating the performance of multiprogrammed workloads. The main observation from these results is that the improvements we achieve change between 5.4% and 13.1%, depending on the applications in the workload. Clearly, integrating our scheme with an OS-based approach could increase these savings further.

7. Related Work

Most of the prior compiler-directed layout transformation are oriented towards improving cache performance. Leung and Zahorjan [8] propose array restructuring to improve the spatial locality in nested loops. O'Boyle and Knijnenburg [22] describe a unifying framework for non-singular layout transformations. Both these works and our work employ data transformation matrices. However, while these prior works focus on improving data locality (cache performance), we focus on improving off-chip accesses (cache misses). Therefore, our work is also orthogonal to these works. Further, since our desired transformation matrix U is only characterized by its v-th (data partitioning dimension) row vector, one can further take the data locality into account by adding additional constraints on U.

Franz and Kistler [23] increase cache utilization of pointer-based programs by splitting data objects in memory. Rivera and Tseng [11] introduce padding to eliminate conflict misses. Anderson et al [10] propose an integrated computation parallelization and layout transformation framework, and use strip-mining and permutation to improve data locality. We employ these layout transformation techniques to address different problems. In our context, padding is used to align the based address of inter-array and intra-array to ensure each data element has the desired virtual address in the transformed layout; and strip-mining and permutation are used to ensure that the off-chip accesses can be sent to the desired memory controller. Since our goal is different from these works, we generate different transformation formulas.

Bugnion et al [24] present a compiler-directed page coloring strategy to eliminate conflict misses in parallel application. Jin et al [25] propose a distributed L2 cache management approach through page-level data to cache slice mapping in a future processor chip comprising many cores. Cho and Jin [26] considered an OS-level page allocation approach to managing on-chip L2 caches. Ros et al [27] propose a distance-aware round-robin mapping policy, an OS-managed policy which addresses the trade-off between cache access latency and number of off-chip accesses. Our page allocation algorithm is simpler than these works since we take advantage of the layout transformation to simplify the job for the OS. We also want to emphasize that, the page allocation scheme in our second component is only a complement to the first component. Lu et al [5] develop a layout transformation for enhancing NUCA-based chip multiprocessors by reducing non-local L2 accesses for localizable computations. There are two differences between our work and this work. First, they focus on on-chip accesses, while our goal is to investigate the impact of off-chip accesses on on-chip network. Second, we handle both shared L2 and private L2 caches while the their work targets only shared L2. Consequently, the theory we developed is entirely different from theirs.

There has been a large body of prior work addressing the problem of data distribution on large-scale NUMA systems [28–30]. Marathe et al [28] present a profile-based memory placement methodology. Navarro et al [29] discuss a compiler-based scheme to find the iteration and data decompositions that minimize communication and load imbalance overheads in parallel programs targeted at NUMA architectures. Majo and Gross [30] study the data distribution of a program to the individual and multiple data access patterns on NUCA. As opposed to the prior NUMA-oriented work, our approach handles the case with shared on-chip caches, a unique characteristic of multicores. Also, in prior NUMA-based work, there are typically two locality domains: local versus global. In contrast, in our target NoC-based system, there are "degrees of locality", depending on the distance between the requesting core and the target memory controller, which we exploit. Finally, prior NUMA based work targets "inter-node optimization", whereas our work targets "intra-node optimization", where a "node" in this context is an NoC based multicore.

In contemporary NUMA systems, congestion on memory controllers and interconnects are the main performance bottlenecks. Observing this, Dashti et al [31] develop an OS-based memory placement algorithm that mainly targets traffic congestion. The proposed algorithm uses data replication and data migration, and goes beyond conventional locality optimizations normally employed in traditional NUMA systems. Such OS-based techniques are complementary to compiler-based approaches (such as ours), and in principle, best improvements can be obtained by coordinating the compiler and OS based approaches.

Recent architecture-based works involve memory scheduling of off-chip accesses (e.g. [7, 16, 32]), which are orthogonal to our compiler based approach. Das et al [33] propose new application-to-core mapping policies that reduce the inter-application inter-

ference in the on-chip network and memory controllers. However, they focus on multiprogrammed workloads with only private caches, and their approach is purely architectural. Studies on memory controller placement for NoC-based multicores include the works from [19] and [34]. Abts et al [19] show that intelligent memory controller placement and routing strategy can help reduce the contention (hot spots) in the on-chip network and lower the variance in reference latency. Xu et al [34] investigate the optimal placement of multiple memory controllers in an 8×8 NoC and propose a generic "divide and conquer" method for solving the placement of memory controllers in large NoCs. These studies are largely orthogonal to the main focus of our work.

8. Conclusions

To our knowledge, this is the first compiler-based work that targets optimizing on-chip network behavior of off-chip accesses in both private L2s and shared L2 based manycores. Specifically, it minimizes the distance between the requester core/node and the target memory controller. Doing so brings two benefits in NoC: first, off-chip data accesses are expedited as they travel shorter distances over the on-chip network, and second, on-chip data access latencies are also reduced as, after the optimization, they are less affected by off-chip accesses. Expediting off-chip requests in the on-chip network also reduces their wait latencies in the memory system. Our results with private L2s (Shared L2), indicate an average of 20.5% (24.3%) saving in execution time. The corresponding execution time improvement, in the case of shared L2 is about 24.3%.

Acknowledgments

We thank Prof. Alexandra Fedorova for her feedback and comments on the paper. This work is supported in part by NSF grants 1213052, 1205618, 1439021, 0963839, and 1017882, as well as a grant from Intel.

References

[1] L. Benini and G. D. Micheli, *Networks on Chips: Technology and Tools*. Elsevier Inc., 2006.

[2] J. Lira, C. Molina, R. N. Rakvic, and A. González, "Replacement techniques for dynamic NUCA cache designs on CMPs," *J. Supercomput.*, 2013.

[3] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," *Proc. of HPCA*, 2009.

[4] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," *Proc. of MICRO*, 2004.

[5] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai, "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," *Proc. of PACT*, 2009.

[6] M. T. Kandemir, Y. Zhang, J. Liu, and T. Yemliha, "Neighborhood-aware data locality optimization for NoC-based multicores," *Proc. of CGO*, 2010.

[7] Y. Kim, D. Han, O. Mutlu, and M. Harchol-balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," *Proc. of HPCA*, 2010.

[8] S.-T. Leung and J. Zahorjan, "Optimizing data locality by array restructuring," *Technical Report, Dept. of Computer Science and Eng., Univ. of Washington*, 1995.

[9] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[10] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and computation transformations for multiprocessors," *Proc. of PPOPP*, 1995.

[11] G. Rivera and C. Tseng, "Data transformations for eliminating conflict misses," *Proc. of PLDI*, 1998.

[12] "Open64," *http://www.open64.net*.

[13] "Gem5," *http://gem5.org*.

[14] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," *OpenMP Shared Memory Parallel Programming*, 2001.

[15] "Mantevo," *http://mantevo.org/*.

[16] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," *Proc. of MICRO*, 2010.

[17] "Micron DDR3 SDRAM Part MT41J128M8," *Micron Technology Inc.*, 2007.

[18] W. Ding, X. Tang, M. T. Kandemir, Y. Zhang, and E. Kultursay, "Optimizing off-chip accesses in manycores,"

[19] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core CMPs," *Proc. of ISCA*, 2009.

[20] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on cc-numa compute servers," *Proc. of ASPLOS*, 1996.

[21] T. Snavely, "Symbiotic jobscheduling for a simultaneous multithreaded processor," *Proc. of ASPLOS*, 2000.

[22] M. O'Boyle and P. Knijnenburg, "Non-singular data transformations: definition, validity and applications," *Proc. of ICS*, 1997.

[23] M. Franz and T. Kistler, "Splitting data objects to increase cache utilization," tech. rep., University of California, Department of Information and Computer Science, 1998.

[24] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," *Proc. of ASPLOS*, 1996.

[25] L. Jin, H. Lee, and S. Cho, "A flexible data to L2 cache mapping approach for future multicore processors," *Proc. of MSPC*, 2006.

[26] S. Cho and L. Jin, "Managing distributed, shared L2 caches through os-level page allocation," *Proc. of MICRO*, 2006.

[27] A. Ros, M. Cintra, M. E. Acacio, and J. M. Garcia, "Distance-aware round-robin mapping for large NUCA caches," *Proc. of HiPC*, 2009.

[28] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-directed page placement for CC-NUMA via hardware-generated memory traces," *JPDC.*, 2010.

[29] A. Navarro, E. Zapata, and D. Padua, "Compiler techniques for the distribution of data and computation," *JPDS*, 2003.

[30] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for numa systems," *Proc. of CGO*, 2012.

[31] F. G. L. L. Q. R. Dashti, Fedorova, "Traffic management: A holistic approach to memory placement on numa systems," *Proc. of ASPLOS*, 2013.

[32] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: Memory subsystem control with a unified predictor," *Proc. of ICS*, 2012.

[33] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," *Proc. of HPCA*, 2013.

[34] T. Xu *et al.*, "Optimal memory controller placement for chip multiprocessor," *Proc. of CODES+ISSS*, 2011.