# Enhancing Computation-to-Core Assignment with Physical Location Information

Orhan Kislal
The Pennsylvania State University
University Park, Pennsylvania, USA
kislal.orhan@gmail.com

Jagadish Kotra
The Pennsylvania State University
University Park, Pennsylvania, USA
jbk5155@cse.psu.edu

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

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

Myoungsoo Jung
Yonsei University
Seoul, South Korea
m.jung@yonsei.ac.kr

## Abstract

Going beyond a certain number of cores in modern architectures requires an on-chip network more scalable than conventional buses. However, employing an on-chip network in a manycore system (to improve scalability) makes the latencies of the data accesses issued by a core non-uniform. This non-uniformity can play a significant role in shaping the overall application performance. This work presents a novel compiler strategy which involves exposing architecture information to the compiler to enable an optimized computation-to-core mapping. Specifically, we propose a compiler-guided scheme that takes into account the relative positions of (and distances between) cores, last-level caches (LLCs) and memory controllers (MCs) in a manycore system, and generates a mapping of computations to cores with the goal of minimizing the on-chip network traffic. The experimental data collected using a set of 21 multi-threaded applications reveal that, on an average, our approach reduces the on-chip network latency in a 6×6 manycore system by 38.4% in the case of private LLCs, and 43.8% in the case of shared LLCs. These improvements translate to the corresponding execution time improvements of 10.9% and 12.7% for the private LLC and shared LLC based systems, respectively.

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

***Keywords*** Multicore Architectures, Compiler, Computation Mapping

**ACM Reference Format:**
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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192386

## 1 Introduction

With the increase in the number of cores in multicore systems, the traditional on-chip bus is becoming a bottleneck due to serialization and arbitration latencies. On-chip interconnects (also called network-on-chip – NoC) proposed by various researchers/vendors have been successful in mitigating the serialization and arbitration latencies, and have been widely accepted as a scalable alternative to on-chip buses. Various commercial products including Intel teraflops research chip [1], Intel Xeon Phi based processors [7], Tilera [4], and Intel Single-Cloud Chip [2] are examples of the NoC-based architectures in industry in the general purpose domain. In addition, on-chip networks have been widely deployed in domain-specific accelerators, including Anton [53] for molecular-dynamic code acceleration and NVIDIA GPUs for graphics acceleration.

However, adopting an on-chip network in a manycore system brings non-uniformity as far as data access latencies are concerned. This is primarily because both on-chip and off-chip data traffic use the same on-chip network and the network latency of a data access depends largely on the distance between the requesting core and data location; the latter can be a local cache (on-chip), a remote cache (on-chip), or main memory (off-chip) accessed via on-chip memory controllers (MCs). Prior architecture-centric research [31, 46–48, 50, 52] investigated different ways of constructing scalable networks. Prior software research [41, 45], on the other hand, studied various techniques to reduce network traffic and/or improve power efficiency. However, existing software-based solutions are limited to either certain classes

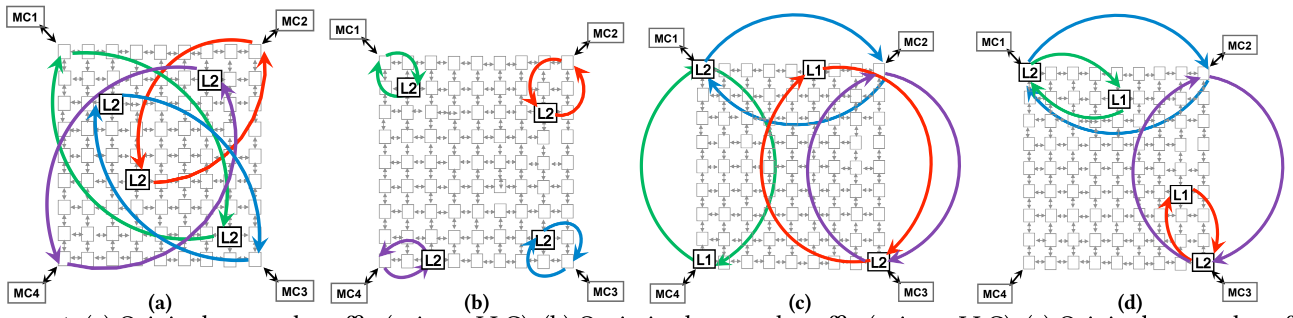


**Figure 1.** (a) Original network traffic (private LLC). (b) Optimized network traffic (private LLC). (c) Original network traffic (shared LLC). (d) Optimized network traffic (shared LLC). In the shared LLC case, a data request that misses in the L1 cache is directed first to an L2 bank (depending on the address of the data). Note that the target L2 bank can be far away from the L1 that misses the request. In both the private and shared LLC cases, an LLC miss travels over NoC to reach the target MC.

of applications or certain types of cache/memory hierarchies, and are not able to optimize both on-chip (LLC hits) and off-chip (LLC misses) data accesses in shared cache-based systems for both regular and irregular application programs.

We present novel compiler support that can be used to map multi-threaded application programs to on-chip network based manycores. The primary knob our compiler employs is computation-to-core mapping. At a high level, what our approach tries to achieve is illustrated in Figures 1a through 1d. (a) shows a typical (off-chip) data access scenario in the case of private LLC in an NoC-based manycore. One can observe that the (off-chip) data accesses are all over the 2D network space, possibly contending with each other, congesting network routers/buffers, and eventually leading to significant network latencies. In (b), on the other hand, the off-chip data accesses are localized, that is, cores access data via the nearest memory controller, reducing the on-chip network traffic significantly. It is important to note that, in doing so, not only the distances traveled by data accesses are reduced, but also the probability of congestion on the network. (c) shows a similar situation in the case of S-NUCA [30] type of shared LLCs. In this case, the network traffic is due to both LLC accesses and off-chip memory accesses, as the target LLC bank may be in a different node than the core that issues the request. (d) optimizes the scenario in (c), leading to significant reductions in both access distances and network congestion for both on-chip and off-chip data accesses.

Figure 2 quantifies the potential of optimizing the on-chip network performance in our multi-threaded applications for the private and shared LLCs. In this graph, we plot the execution time improvement, over the original execution (i.e., without any network optimization), when we work with an ideal network. In this context, an ideal network is one that incurs zero latency in transmitting messages between cores and last-level caches (on-chip data accesses) as well as between last-level caches and memory controllers (off-chip data accesses). One can observe that, a significant fraction of execution time (14% for private LLCs and 17.1%

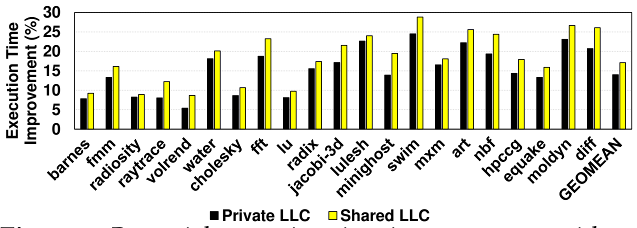


**Figure 2.** Potential execution time improvements with an ideal (zero-latency) on-chip network.

for shared LLCs, on average) can be saved if the on-chip network performance is maximized.

The rest of this paper presents a compiler-directed strategy that improves the on-chip network performance by optimizing computation-to-core mapping (assignment). Our main goal is to approach the maximum savings plotted in Figure 2 when using a realistic on-chip network. A unique characteristic of our approach is that, in performing this mapping, it considers both the distance between cores and LLCs as well as the distance between LLCs and MCs. In other words, our approach is location aware. Furthermore, our implementation can handle both regular applications (those with compile-time analyzable access patterns) and irregular applications (those with index array-based access patterns). It is to be noted that these two types of applications cover a large fraction of workloads running on high-performance manycore-based systems (e.g., those employing Intel Xeon Phi [7]). Our specific contributions in this work can be summarized as follows:

• We introduce four novel concepts to capture the (i) affinity of computations (loop iterations) to MCs, (ii) affinity of cores to MCs, (iii) affinity of computations to LLCs, and (iv) affinity of cores to LLCs.

• Using these four concepts, we present a compiler-based strategy to assign/map computations to cores for both private cache and shared cache systems. In the case of regular applications, our approach analyzes and transforms the code at compile time, and in the case of irregular ones, it employs a variant of the inspector-executor paradigm [18], developed originally in the context of application parallelism. The main goal of our approach is to *reduce* the distance-to-data by exploiting all four concepts mentioned in the item above.

• We describe our compiler implementation and present experimental data running a set of 21 multi-threaded applications on a cycle-accurate multicore simulator [10]. The collected data on a 6×6 manycore system indicate that, on average, our approach reduces the on-chip network latency by 38.4% in the case of private LLCs and 43.8% in the case of shared LLCs. These improvements translate to the corresponding execution time improvements of 10.9% and 12.7%. Note that, these savings are very close to those in Figure 2, which represents an ideal network (optimal case). Further, to demonstrate that our savings extend to commercial systems as well, we also present execution time improvements brought by our compiler on a commercial manycore system [55]. These results indicate that our approach brings performance improvements ranging between 12.2% and 29%.

## 2 Architecture Background: On-Chip Network Based Manycores

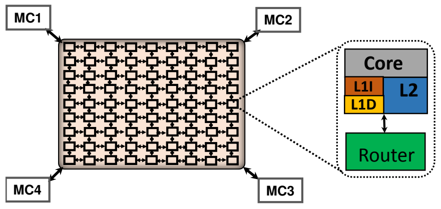


**Figure 3.** A manycore architecture and contents of a node.

**Manycore Architecture Overview:** Manycores are organized as nodes connected through an on-chip network (also called NoC). Figure 3 shows a 9x9 manycore with various hardware components comprising a node. Each node consists of a processing core, private L1 instruction and data caches, and an L2 cache bank along with a network router. Memory controllers (MCs) shown are connected to the off-chip memory through memory channels. These MCs manage the off-chip DRAM and are responsible for fetching/writing data from/to the off-chip main memory. The last-level cache (LLC) – in our case L2[1] – is a unified cache, which can cache both data and instructions. The network router shown in Figure 3 routes the packets between cores/LLC banks and LLC banks/main memory, and employs X-Y routing with wormhole switching to route the packets across the chip.

**LLC Management:** Typically, LLCs in a manycore are organized as bank based structures. A manycore LLC managed as a shared LLC incurs non-uniform latencies (even for cache hits) depending on where the data is allocated. Cache line (block) allocated in a local LLC bank incurs lower latency compared to accessing a cache line allocated in a remote LLC bank. Hence, these architectures are popularly referred to as Non-Uniform Cache Access (NUCA) architectures [30]. The reason for non-uniformity in cache access latencies is due to the additional on-chip traversal involved in accessing the cache lines in the remote LLC banks. Popular NUCA organizations include Static-NUCA (S-NUCA) and Dynamic-NUCA (D-NUCA) [14]. In S-NUCA, a cache line is statically mapped to a single LLC bank based on the address mapping, while cache lines in D-NUCA are migrated based on the frequency of cache line reuse. Since the overhead involved in tracking a cache line is highly coupled with complexity and power, D-NUCA is not preferred in practice.

In S-NUCA, the lower order bits in a physical address give the byte offset in a cache line. The next group of bits are typically used to map a physical address to an LLC bank. If the LLC bank is not shared, these cache bank bits are ignored and the cache line is allocated in the local node. However, if the LLC is shared, for a 9x9 manycore with, say, a total of 81 LLC banks, seven bits are used to map the cache line to an LLC bank. An L1 miss which results in an LLC hit in the S-NUCA architecture involves request/response messages to/from the LLC bank traversing on the NoC, while in the private LLC organization (since always the local LLC bank is accessed) no additional network traffic is incurred. Therefore, in the S-NUCA architecture, an LLC hit still incurs additional traffic on the network, unlike the private LLC. However, the private LLC bank organization can suffer from severe underutilization if there is a disparity across the memory intensities of the threads running on different cores.

**Handling LLC Misses:** Upon a miss in the L1 and LLC bank, a memory request is sent to the corresponding MC based on the memory address mapping. Physical addresses can be interleaved in main memory at various granularities. Typically, the interleaving is done at a page granularity. In this case, the lower order bits from 0-11 in a physical address represent the page offset in a 4KB page and, for a 4 channel system, the next (higher) two bits represent the memory controller this off-chip request will be routed to in case of an LLC miss. For a larger manycore, an off-chip access, irrespective of the LLC organization (be it private or S-NUCA), can incur significant on-chip traffic if the physical address is mapped to an MC which is far-off from its corresponding LLC bank. Consequently, optimizations that target reducing the on-chip traffic can directly translate to significant improvements in performance. One possible approach to improve application performance on network-based manycores is to reduce the distance between locations of the requester core and the data being requested. This paper proposes a practical computation mapping algorithm for that. Note that, one can also distribute physical addresses across available MCs using granularities smaller than a page [67]. For example, some architectures support cache line granularity, i.e., successive cache line-sized data are distributed across MCs in a round-robin fashion [67].

[1]Our approach can work with cache hierarchies of any depth.

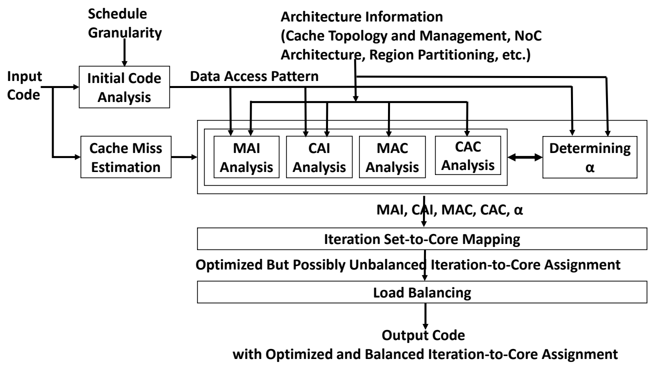

Schedule Granularity
Architecture Information (Cache Topology and Management, NoC Architecture, Region Partitioning, etc.)
Input Code
Initial Code Analysis
Data Access Pattern
Cache Miss Estimation
MAI Analysis
CAI Analysis
MAC Analysis
CAC Analysis
Determining α
MAI, CAI, MAC, CAC, α
Iteration Set-to-Core Mapping
Optimized But Possibly Unbalanced Iteration-to-Core Assignment
Load Balancing
Output Code with Optimized and Balanced Iteration-to-Core Assignment

**Figure 4.** High-level view of our approach.

## 3 Design Methodology

### 3.1 Outline of Our Strategy

We target multi-threaded applications parallelized over multiple cores. As discussed previously, our goal is to assign computations (iterations) to cores such that the network latency is reduced. Figure 4 illustrates the high-level view of our approach in the case of regular applications. Our compiler approach takes source code as input, and applies an initial code analysis which extracts data and control dependences as well as data access and reuse patterns. After this analysis phase, the derived data access patterns are forwarded to an affinity analysis.[2] Based on the affinities derived by the compiler, the loop iterations are then mapped to their most preferable cores and the corresponding output code is generated. Note that workload balance is also considered while we perform the computation-to-core mapping. Note also that loop bounds in our target programs do not necessarily need to be known at compile time as our approach performs a limited symbolic analysis as well. In the case of irregular applications, our compiler does not directly embed the determined iteration-to-core assignment into the application code; rather, it generates and embeds a code into the application code, which determines the final iteration-to-core assignment at runtime. Specifically, we exploit a variant of the inspector-executor paradigm [19]; the inspector phase captures the input-specific data access patterns to determine iteration mappings, and the executor phase uses those derived mappings.

To formulate the affinities which are the foundations of our approach, we define four new concepts: memory affinity of iterations (**MAI**), memory affinity of cores (**MAC**), cache affinity of iterations (**CAI**), and cache affinity of cores (**CAC**). While all these four concepts are used in the case of shared LLCs, only MAI and MAC are needed in the case of private LLCs. We first introduce MAI and MAC and present our approach in the context of private LLCs, and later move to the discussion of shared LLCs and explain CAI and CAC.

[2]In the case of regular-applications, our approach analyzes the data access patterns; so, it is uniformly applied, irrespective of the specific input-set being used.

### 3.2 MAI

**Problem 1:** *Given a loop nest, partition and map iterations to cores, such that the data elements accessed by loop iteration $\vec{i}$ are fetched from the nearest memory controllers.*

For a given loop iteration $\vec{i}$ [3], we define MAI($\vec{i}$) as a vector (also called *affinity vector*) that determines the affinity of $\vec{i}$ to the different memory controllers in the system. Specifically, MAI($\vec{i}$) in an $m$-entry vector ($m$ being the number of memory controllers), where the $k$th entry ($1 \leq k \leq m$) indicates the strength of the affinity (bond) between $\vec{i}$ and memory controller $k$. In our implementation, this strength is expressed as the fraction of the data requests of $\vec{i}$ destined to the $k$th memory controller in the system (if they miss in the LLC). To explain how MAI($\vec{i}$) is calculated, consider the manycore architecture in Figure 3 and the code fragment in Figure 5. Let us assume that, among the four data accesses issued by a particular iteration $\vec{i}$, [4] two accesses memory controller MC1, one accesses MC2 and one accesses MC3, as shown in the second column of Table 1. In this case, MAI($\vec{i}$) is set to $(0.5, 0.25, 0.25, 0)$, indicating that half of the requests go to the first memory controller (MC1), and each of the second and third memory controllers receives 25% of the data requests this iteration makes. Note that, higher the value (weight) in the $k$th position of MAI($\vec{i}$), the stronger the affinity between $\vec{i}$ and MC$k$. Furthermore, a stronger affinity between $\vec{i}$ and MC$k$ indicates that it may be preferable to assign $\vec{i}$ to a core that is close to MC$k$. This latter concept, that is, the proximity/closeness between a core and a memory controller, is captured by MAC (explained shortly).

**Table 1.** Locations of data accesses.

| | if misses (MC) | if hits (region) | Realistic Scenario |
|---|---|---|---|
| A(i) | MC3 | R2 | MC3 |
| B(i) | MC1 | R4 | R4 |
| C(i) | MC1 | R4 | R4 |
| D(i) | MC2 | R8 | MC2 |

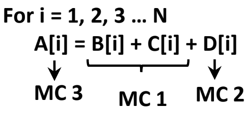


**Figure 5.** An example loop.

We use iteration set as the granularity of computation scheduling in our compiler. An iteration set consists of a set of consecutive loop iterations. We divide the iterations of a loop nest into iteration sets (of equal size, except perhaps for the last iteration set), and compute MAI($\vec{I}$) for each iteration set $\vec{I}$ considering *all* iterations $\vec{i}$ in $\vec{I}$. The reason that our compiler works with iteration sets rather than individual iterations is because consecutive individual loop iterations usually exhibit spatial locality (both row-buffer locality and cache-line locality) and have the similar MAI value. Using the concept of iteration set helps us maintain the spatial locality while applying our approach.

[3]Loop iterations are represented as vectors, i.e., $\vec{i} = (i_1 i_2 \cdots i_n)^T$, where $i_s$ is the value of the $s$th loop from top.

[4]Note that, in this case, the nest has only one loop, that is, $\vec{i}$ is one dimensional (i.e., $\vec{i} = i$).

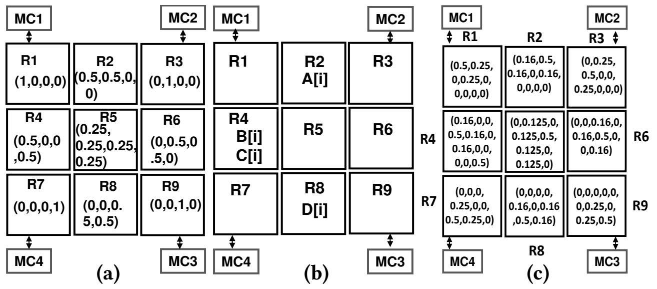

MC1
MC2
R1 (1,0,0,0)
R2 (0.5,0.5,0,0)
R3 (0,1,0,0)
R4 (0.5,0,0,0.5)
R5 (0.25,0.25,0.25,0.25)
R6 (0,0.5,0,5,0)
R7 (0,0,0,1)
R8 (0,0,0.5,0.5)
R9 (0,0,1,0)
MC4
MC3
(a)
MC1
MC2
R1
R2 A[i]
R3
R4 B[i] C[i]
R5
R6
R7
R8 D[i]
R9
MC4
MC3
(b)
MC1
MC2
R1 (0.5,0.25,0,0.25,0,0,0,0,0)
R2 (0.16,0.5,0.16,0,0.16,0,0,0,0)
R3 (0,0.25,0.5,0,0,0.25,0,0,0)
R4 (0.16,0,0,0.5,0.16,0,0.16,0,0,0,0,0.5)
R5 (0,0.125,0,0.125,0.5,0.125,0,0.125,0)
R6 (0,0,0.16,0,0.16,0.5,0,0,0.16)
R7 (0,0,0,0.25,0,0,0.5,0.25,0)
R8 (0,0,0,0,0.16,0,0.16,0.5,0.16)
R9 (0,0,0,0,0,0,0.25,0,0.25,0.5)
MC4
MC3
(c)

**Figure 6.** (a) MAC vectors. (b) Cache (L2) hits in the 2D space. (c) CAC vectors.

## 3.3 MAC

**Problem 2:** *For a specific MAI of a given iteration set $\vec{I}$, map it to core k, such that the total amount of data movement due to the LLC misses originating from the execution of $\vec{I}$ is minimum.*

We use Manhattan Distance to measure the distance between a core/private LLC and an MC. Therefore, the affinities between a core and the memory controllers in the system can be expressed using a set of Manhattan Distances. However, since two nearby cores would, most of the time, have similar preferences for a given memory controller in the system, in this work, we adopt a simpler measure of affinity. We divide the 2D space (of Figure 3) into 9 regions ($R_1$ through $R_9$), and the cores that fall in the same region are assumed to have the same affinity to a given MC. The benefits of dividing cores into regions are two-folds. First, it allow us to to map iteration sets with respect to MAI vectors. Second, within a region, more core candidates allow us to achieve better workload balance. Figure 6a shows MAC vectors for (cores in) different regions ($R$). For example, the reason MAC is $(0.5, 0.5, 0, 0)$ in the second region ($R_2$) is because the cores in this region are much closer to the first two MCs (MC1 and MC2) compared to the remaining two controllers. On the other hand, the cores in $R_5$ have (more or less) equal-distances to all four MCs in the system, which is indicated by setting their MAC to $(0.25, 0.25, 0.25, 0.25)$.

## 3.4 Putting MAI and MAC Together

Recall that our goal is to assign a given set of loop iterations to available cores such that each core satisfies most of its off-chip data accesses from the nearest memory controller(s), as illustrated in Figure 1b. MAI and MAC, both represented as affinity vectors, provide us with the capability to measure the affinity of loop iteration-to-core. Specifically, let us first define the similarity (or difference) between two affinity vectors: Given two affinity vectors, $\vec{\delta} = (\delta_1, \delta_2, \cdots, \delta_m)$ and $\vec{\delta\prime} = (\delta\prime_1, \delta\prime_2, \cdots, \delta\prime_m)$, the difference (opposite of similarity) between them is defined as

$$\eta(\vec{\delta}, \vec{\delta\prime}) = \sum_k |\delta_k - \delta\prime_k|/m.$$

**Algorithm 1** Algorithm to assign iteration sets to cores (private LLC).

```
Input: Number of cores, regions and iteration sets
Output: Iteration sets to core assignment
 1: Divide cores into regions $R_a$
 2: for each $R_a$ do
 3:     compute MAC($R_a$);
 4: divide iterations into iteration sets $\vec{I}_k$
 5: for each iteration set $\vec{I}_k$ do
 6:     compute MAI($\vec{I}_k$);
 7: /* assigning iteration sets to regions */
 8: for each $\vec{I}_k$ do
 9:     $\eta_m = \infty$
10:     for each $R_a$ do
11:         $\eta_l = \eta$(MAI($\vec{I}_k$),MAC($R_a$))
12:         if $\eta_l < \eta_m$ then
13:             $\eta_m = \eta_l$
14:             ASSIGNED($\vec{I}_k$) = $R_a$
15: /* load balancing across regions */
16: target_avg = number of iteration sets / number of regions
17: for each $R_a$ in donor $X_c$ region do
18:     for each $R_b$ in receiver $Y_d$ region do
19:         compute NBGH($R_a$,$R_b$)
20: sort NBGH into SORTED_NBGH
21: for each ($R_a$, $R_b$) in SORTED_NBGH do
22:     if $R_a$ can donate and $R_b$ can receive then
23:         transfer iteration sets from $R_a$ to $R_b$
24:         update the surplus and need for $R_a$ and $R_b$ respectively
```

The lower the difference (also called error) between two affinity vectors, the higher the similarity between them. Using the similarity concept on specific value of MAI and MAC, we second calculate $\eta_l$(MAI, MAC). We prefer the core region which gives the smaller value of $\eta_l$(MAI, MAC) such that the total amount of date movement due to that iteration set is minimized.

## 3.5 Iteration Mapping for Private Last-Level Caches

Algorithm 1 gives the pseudo-code of our compiler algorithm, which assigns loop iterations (actually, iteration sets) to cores in the case of private LLC. This algorithm is invoked once for each parallel loop nest in the input code, and has two main parts. The first part assigns each iteration set ($\vec{I}$) to a region using the concepts of MAI and MAC. The second part balances the workloads across different cores, if it is not already balanced at the end of the first step, by moving select iteration sets among regions, in a "location-aware" manner. The first part of our algorithm (defined between lines 1 and 14) goes over all iteration sets of the loop nest being optimized one-by-one, and assigns each iteration set $\vec{I}_k$ to a region $R_a$ such that $\eta_l$(MAI($\vec{I}_k$), MAC($R_a$)) – difference between MAI($\vec{I}_k$) and MAC($R_a$) as defined above – is smaller or equal than $\eta_m$(MAI($\vec{I}_k$), MAC($R_b$)), for any $a \neq b$. That is, for each $\vec{I}_k$, we identify the most suitable region – one with an affinity vector most similar to MAI($\vec{I}$) considering all 9 regions. As an example, consider the MAI $(0.5, 0.25, 0.25, 0)$. The second column of Table 2 gives the error values between this MAI and each of our 9 regions (with the MAC vectors shown in Figure 6a). We see that, for the iteration set with this MAI, region $R_5$ would be the most preferable one since it incurs the minimum error value (0.125). On the other hand, for an iteration set with MAI $(0, 0, 0.5, 0.5)$, the most preferable region would be $R_8$ (the error values in this case are listed in the third column of Table 2).

After this step, we run a load-balancing step (captured by lines 15-24 in Algorithm 1) whose goal is to balance the loads (measured in terms of the number of iteration sets assigned) across the different regions. We achieve this by first calculating the (target) average number of iteration sets per region, and then, identifying the *donor* and *receiver* regions, and finally, transferring some iteration sets from donors to receivers. In this context, donor is a region whose assigned number of iteration sets is larger than the target average, and receiver is a region whose assigned number of iteration sets is smaller than the average. After identifying the donor and receiver regions, we build an ordered set of region pairs (denoted as SORTED_NBGH in Algorithm 1), each consisting of one donor and one receiver. The order used is based on proximity, that is, if a donor and a receiver are neighbors in the 2D space, they take the first place; and the larger the distance between a donor and a receiver is (in the 2D space), the lower the location of the corresponding pair in the order. Then, starting with the top of the list, our approach transfers iteration sets between the corresponding donor and receiver to bring them as close to the average as possible. Let us assume, for instance, that regions $R_1$, $R_5$, and $R_9$ are donors with the surpluses of 2, 8, and 2 iteration sets, respectively, and regions $R_3$ and $R_8$ are receivers with the needs of 3 and 9 iteration sets. In this case, we first start with either $(R_5, R_8)$ or $(R_9, R_8)$ and perform iteration set transitions between $R_5$ and $R_8$ (or between $R_9$ and $R_8$). Supposing that we start with $(R_5, R_8)$, we transfer all surplus of $R_5$ (8) to $R_8$; now, $R_5$ has no surplus left, so it is not considered anymore; and, the need of $R_8$ is reduced to 1. We then consider the pair $(R_9, R_8)$ , and assign 1 iteration set from $R_9$ to $R_8$. At this point, $R_8$ is also out of the picture, and the surplus of $R_9$ is reduced to 1. We next move to pair $(R_9, R_3)$ as, although they (e.g., $R_9$ and $R_3$) are not neighbors, there is only 1 region between them. The remaining pairs are processed similarly. Note that this load-balancing strategy tries to limit the load (iteration set) transfers between close-by regions, that is, it is *location aware*.

Algorithm 1 works well with private LLC since, in such architectures, data from off-chip memory are brought directly into the LLC of the requester core. However, as discussed earlier in Section 2, in the case of shared LLC (S-NUCA), each data block has a home cache/node and the distance between the requester core and home node can also play a significant role in shaping performance. So, to address shared caches, we augment our affinity vectors with two new ones: cache affinity of iterations (CAI) and cache affinity of cores (CAC). In computing these affinities, we again logically partition the 2D on-chip network space into 9 regions.

## 3.6 CAI

**Problem 3:** *Given a loop iteration set $\vec{I}$, map it to a core such that the data elements involved in that iteration set are fetched from the nearest last-level cache (LLC) banks in the shared LLC system.*

For a given iteration $\vec{i}$, we define its CAI($\vec{i}$) as a 9-entry vector[5], such that the $j$th entry indicates the strength of affinity between this iteration and region $j$. As in the case of MAI($\vec{i}$), we use an example to better explain its calculation. Consider the same computation in Figure 5, assuming this time all data accesses hit in some LLC in the system, as illustrated in Figure 6b and indicated in the third column of Table 1. In this case, CAI($\vec{i}$) can be computed as $(0, 0.25, 0, 0.5, 0, 0, 0, 0.25, 0)$, indicating that two of the data references are satisfied from the caches in the 4th region, whereas the remaining two references are satisfied from regions 2 and 8.

## 3.7 CAC

**Problem 4:** *For a specific CAI of a given iteration set $\vec{I}$, map it to core k, such that the total amount of data movement due to fetching data blocks from their home nodes (LLC home banks) is minimum.*

Using the same logical partitioning of the 2D on-chip network space, CAC is calculated in a way similar to MAC. We use a 9-entry vector (as in the case of CAI). For all the cores in a region $R_a$, the $a$th entry of its CAC is set to 0.5, and the remaining weight (affinity) is *equally distributed* across the immediate neighbors of that region. In practice, what this means for a core is that, half of the preference the core has regarding cache accesses is given to the LLCs in the region where that core resides, and the remaining preference is distributed across its neighboring regions. As a result, as shown in Figure 6c, the CAC vector for the cores in region $R_1$ is set to $(0.5, 0.25, 0, 0.25, 0, 0, 0, 0, 0)$, indicating that, as far as the cores in this region are concerned, the most preferable LLCs are those in the same region ($R_1$), and the next preferable LLCs are those in regions $R_2$ and $R_4$, which are immediate neighbors of $R_1$. Similarly, the CAC vectors of the cores in regions $R_2$ and $R_5$ are set to $(0.16, 0.5, 0.16, 0, 0.16, 0, 0, 0, 0)$ and $(0, 0.125, 0, 0.125, 0.5, 0.125, 0, 0.125, 0, 0)$, respectively. Note that, like MAC, CAC is also application independent.

## 3.8 Iteration Mapping for Shared Last-Level Caches

Now that we have affinity vectors MAI, MAC, CAI and CAC, we are ready to present our iteration (set) mapping algorithm for the shared (S-NUCA) LLC case. Recall that $\eta_m$ computed above captures the error (dissimilarity) between a MAI and a MAC, which can be used to determine the "ideal" location (core) for a given iteration set considering its off-chip memory accesses. Similarly, one can define $\eta_c$, as a measure of the error between a CAI and a CAC. In mathematical terms,

[5]In general, in an $m$-region case, it is an $m$-dimensional vector.

**Table 2.** Error calculation for three different MAI vectors.

| | **MAI = (0.5,0.25,0.25,0)** | **MAI = (0,0,0.5,0.5)** | **MAI = (0,0.25,0.25,0)** |
|---|---|---|---|
| **R1** | (0.5+0.25+0.25+0)/4 = 0.25 | (1+0+0.5+0.5)/4 = 0.5 | (1+0.25+0.25+0)/4 = 0.325 |
| **R2** | (0+0.25+0.75+0)/4 = 0.25 | (0.5+0.5+0.5+0.5)/4 = 0.5 | (0.5+0.25+0+0)/4 = 0.1875 |
| **R3** | (0.5+0.75+0.25+0)/4 = 0.375 | (0+1+0.5+0.5)/4 = 0.5 | (0+0.75+0.75+0)/4 = 0.25 |
| **R4** | (0+0.25+0.25+0.5)/4 = 0.25 | (0.5+0+0.5+0)/4 = 0.25 | (0.5+0.25+0.25+0.5)/4 = 0.325 |
| **R5** | (0.25+0+0+0.25)/4 = **0.125** | (0.25+0.25+0.25+0.25)/4 = 0.25 | (0.25+0+0+0.25)/4 = **0.125** |
| **R6** | (0.5+0.25+0.25+0)/4 = 0.25 | (0+0.5+0.5+0)/4 = 0.25 | (0+0.25+0.25+0)/4 = **0.125** |
| **R7** | (0.5+0.25+0.25+1)/4 = 0.5 | (05+0+0.5+0.5)/4 = 0.25 | (0+0.25+0.25+1)/4 = 0.325 |
| **R8** | (0.5+0.25+0+0.25+0.5)/4 = 0.325 | (0+0+0+0)/4 = **0** | (0+0.25+0.25+0.5)/4 = 0.25 |
| **R9** | (0.5+0.25+0.75+0)/4 = 0.325 | (0+0+0.5+0.5)/4 = 0.25 | (0+0.25+0.75+0)/4 = 0.25 |

if CAI = $(\delta_1, \delta_2, \cdots, \delta_9)$ and CAC = $(\delta\prime_1, \delta\prime_2, \cdots, \delta\prime_9)$, then the difference between them is defined as

$$\eta_c(\vec{\delta}, \vec{\delta\prime}) = \sum_k |\delta_k - \delta\prime_k|/9.$$

At this point, it is important to note that, for an iteration set $\vec{I}$, both $\eta_m$ and $\eta_c$ can be important since one of them captures memory affinity while the other captures cache affinity. It is also to be noted that, if we knew that all the accesses made by $\vec{I}$ will be satisfied from the on-chip last-level cache, memory affinity would not be important. On the other hand, if we knew that none of the accesses made by $\vec{I}$ will be satisfied from the on-chip last-level cache (i.e., all will go to the off-chip memory), cache affinity would be less important. But, in practice, only some of the accesses in $\vec{I}$ can be satisfied from the LLCs whereas others go to the off-chip memory, so it is important to consider both $\eta_m$ and $\eta_c$. Motivated by this, we use a weighted sum of them, that is, we define the overall error in the assignment of $\vec{I}$ as

$$\eta = \alpha.\eta_c + (1 - \alpha).\eta_m,$$

where $0 \leq \alpha < 1$. We later discuss how to determine $\alpha$. However, we need to make one important point clear. In an S-NUCA based system, after an L1 miss, first the L2 cache (depending on the address of the data being requested) is checked (where $\eta_c$ plays a role), and in the case of an L2 (our LLC) miss, a request is made to the target MC (based on its physical address) from the missing L2, *not* from the requesting core. Therefore, the definition of MAI in the formulation above needs to be slightly modified in the case of S-NUCA. Specifically, instead of capturing the affinity between a core and an MC, we need to capture the affinity between an LLC and an MC. This is achieved by considering (in computing MAI) the locations of the LLC caches, instead of cores. The mapping part of the compiler algorithm for the case of shared LLC (S-NUCA) is given as Algorithm 2. Note that the load-balancing part is same as that of Algorithm 1, and hence, it is omitted in Algorithm 2.

## 3.9 Discussion

As discussed above, our approach divides the 2D network space into regions, and the concept of "region" is used in the calculations of MAC, CAI, and CAC. Clearly, the region concept is an abstraction to make our formulation of affinity simpler. If desired, one may want to reduce the region size (and thus have more regions) in an attempt to achieve a model with higher accuracy – having a finer-grained/more reliable affinity between cores, LLCs and MCs. In the extreme case, each region can contain only one core. The downside of this approach would be the requirement of working with longer CAI and CAC vectors. In our evaluations, we report results from a sensitivity experiment where we tested the potential benefits that could be brought by smaller regions.

**Algorithm 2.** Core part of the algorithm that assigns iteration sets to cores (shared LLC).

```
1: for each iteration set $\vec{I}_k$ do
2:     $\eta_m = \infty$
3:     for each $R_a$ do
4:         $\eta_1 = \eta(\text{MAI}(\text{LLC}(\vec{I}_k)),\text{MAC}(R_a))$
5:         $\eta_2 = \eta(\text{CAI}(\vec{I}_k),\text{CAC}(R_a))$
6:         if $\eta_1 + \eta_2 < \eta_m$ then
7:             $\eta_m = \eta_1 + \eta_2$
8:             $\text{ASSIGNED}(\vec{I}_k) = R_a$
```

Our approach can also be modified to work with network topologies other than 2D meshes. The only requirement is that the positions of the cores, last-level caches, and memory controllers with respect to one another should be exposed to our compiler. For example, we can handle different placements of MCs in the network space, and in fact, in Section 5, we report results with an alternate MC placement. Similarly, we can work with more sophisticated/non-standard LLC management schemes such as [13] as long as they are exposed to our compiler.

Another point is that, if desired, certain affinity vectors can be computed differently without changing the high-level structure of our optimization strategy. For example, in computing CAI, we give, for a given core, half of the preference to the LLCs of the region where that core resides. Clearly, one may opt to assign different weights to the entries of CAI, in an attempt to better reflect the specifics of a given architecture. Also, in calculating MAC vectors, we can encode memory controller preferences in a finer-granular fashion. In addition, our approach works at a region granularity, that is, each iteration set is assigned to a region. There is also the question of how the iterations assigned to a region are to be distributed over the cores in that region. In our implementation, we make this finer-granularity assignment randomly. In other words, once, for a given iteration set, the most suitable region is identified, the iteration set is assigned to a core in that region randomly, with the only constraint that the loads of the cores in the region should be more or less balanced. Again, if desired, one may want to employ different strategies for this fine-granular mapping. One option would be letting the OS to do the scheduling within each region[6]. Another option would be working with smaller regions (with fewer cores), to have more control over where (to which core) an iteration set is mapped within its assigned region.

[6]Our experiments with this OS option generated around 2% better results (on average) than the random selection option.

The next point of discussion is that is that two different paths, each having the same number of links, can (in principle) experience different latencies due to contention. However, note that our approach tries to reduce this contention at the first place, by reducing the distance a data request travels on the network. This brings two important advantages. First, a message visits minimum number of links, and second, this also makes contention less likely, as the probability of contention increases with the number of links a message traverses.

## 4 Implementation Details

We implemented our scheme using the PLUTO compilation framework [11], and tested its effectiveness using both regular multi-threaded applications (those with mostly compiler-analyzable data accesses patterns) and irregular multi-threaded applications (those with mostly indirect/index-array based data access patterns). For the first group, the input code is modified by the compiler to generate optimized mapping of computations to cores. For the second group on the other hand, we employed the inspector-executor paradigm [18] (more on this shortly). [7] In both these implementations, one of the crucial issues is to determine an appropriate value for the $\alpha$ parameter when targeting the shared cache systems. In the case of regular applications, we determine the value of $\alpha$ by using a cache miss estimation strategy. Let us consider the computation shown in Figure 5 once more, this time assuming a cache miss estimating strategy (we use CME [23] in our implementation[8]) indicates that, for a specific $i$, $B(i)$ and $C(i)$ will hit in the last-level cache, whereas $A(i)$ and $D(i)$ will miss in the cache and access the off-chip memory (see the last column of Table 1), i.e., $A(i)$ accesses MC3 and $D(i)$ accesses MC2 (let as assume, for simplicity, that each iteration set has only 1 iteration, i.e., $\vec{I} = i$). In this case, we compute MAI$(i)$ as $(0, 0.25, 0.25, 0)$ and CAI$(i)$ as $(0, 0, 0, 0.5, 0, 0, 0, 0, 0)$. The last column of Table 2 lists the error values obtained with this refined MAI, which indicate that regions $R5$ and $R6$ are the most suitable regions for this iteration. In other words, cache miss estimation can be used to refine the computations of MAI and CAI. Furthermore, since we now know that two of the accesses are hits and the remaining two are misses, we can use this information to set $\alpha$ to 0.5. If on the other hand only one of these four requests were estimated to be cache hit, the $\alpha$ parameter would be set to 0.25, giving more weight, for this iteration, to the off-chip memory accesses (as opposed to cache accesses). To summarize, we use cache miss estimation [23] to (1) produce more precise MAI and CAI vectors and (2) determine the value of the $\alpha$ parameter. Note that cache miss estimation in our work is useful for both private cache and shared cache systems. In the former case, it is used to refine MAI, while in the latter case, it is used for refining MAI and CAI as well as determining the value of $\alpha$. We want to emphasize that, our proposed approach does *not* affect the degree/amount of parallelism in the loop nest being optimized; rather, it just assigns iterations to cores with the goal of reducing the footprint of data accesses on the on-chip network.

Let us now discuss the inspector/executor paradigm employed for the case of irregular applications. In most irregular applications there is an outer (timing) loop that iterates either a fixed number of times or until a convergence criterion is met. Typically, either before this timing loop is entered or after its first iteration, the contents of index arrays are determined.[9] In our implementation of the inspector/executor paradigm, the compiler inserts an "inspector code" into the application code after the first iteration of the timing loop[10] that (1) determines for each data access (and also for each iteration) the LLC hits, misses, caches that satisfy hits, and the MCs that handle misses; (2) constructs MAI and CAI; (3) determines the value of $\alpha$, and (4) fills a data structure to indicate computation set-to-core assignment. Then, the timing loop (now it is called executor) executes using this scheduling information. All these four steps mentioned above are executed at runtime, and consequently, they incur execution time overheads. In the results reported later, these overheads are fully captured. On the other hand, compared to the static compiler analysis, the inspector/executor paradigm has the advantage of working with more accurate information. Specifically, the hit/miss behavior in this case is observed at runtime (instead of being estimated at compile time) using CME, and similarly the calculations of MAI, CAI and $\alpha$ are performed considering the actual cache and memory accesses. Assuming that, for a given iteration set, the hit/miss behavior and ids of the caches and MCs accessed do not significantly change from one iteration of the timing loop to the other, we can expect the values (of MAI, CAI and $\alpha$) calculated this way to be more accurate than those values calculated in the case of regular applications based on compiler-estimated values. In our experiments, we carefully

[7]Note that the distinction between "regular" and "irregular" applications can be blurry at times. In particular, many large array-based application programs can have both regular and irregular data accesses. For the sake of simplicity, in our experiments, we classify an application "regular", if a large majority of its data accesses are via regular references, and similarly, an application is termed as "irregular" if a large majority of its array accesses are through index arrays. Each of our 21 applications fall into one of these two categories. Note however that, even when optimizing an irregular application using the inspector-executor paradigm, its regular loop nests (if any) are completely optimized at compile time.

[8]We changed the original implementation in [23] to employ statistical methods when computing the number of solutions. Such methods helped us maintain the accuracy of the original implementation while making it run much faster in practice. We observed that the accuracy of our CME implementation varied between 76% and 93%, depending on the application program.

[9]In some applications, index arrays may be assigned in multiple places in the program code. Our approach handles such cases as well, by executing the inspector code multiple times.

[10]Timing loops are marked in the code by the user.

quantify the overheads brought by our approach in the case of irregular applications. Once our compiler computes MAI, MAC, CAI, and CAC, they are stored in a data structure – essentially, a table indexed by the iteration set id. Our approach then uses this table as well as the architecture description to assign iterations to cores.

Finally, it is important to note that, while the compiler/programmer works with virtual addresses, the hardware works with physical addresses. More specifically, the ids of the LLC or MC to be accessed by a given data reference (if it misses in L1) is a function of the physical address. In the case of irregular applications, our approach employs the inspector-executor paradigm, and consequently, learns – in the inspector phase at runtime – the LLC and MC location(s) for a given iteration set. In the case of regular applications on the other hand, the compiler needs to figure out these LLC and MC ids at compile time. To do this, in addition to exposing the numbers and locations of LLCs and MCs to the compiler, we also need to expose the physical address. In our approach, this is achieved by using an OS call during data allocation which ensures that the locations in the virtual address that correspond to the MC and LLC bits are *not* modified during the virtual address-to-physical address translation. As a result, by looking at the virtual address of a data element, we will be able to tell the ids of the target LLC and MC (if it misses in LLC) that will be accessed. Consequently, for a given iteration set $\vec{I}$, we are able to build the MAI and CAI vectors. We want to emphasize that, while our approach puts some restriction on the OS during virtual-to-physical address translation, we do not expect this restriction to be a significant problem. This is because the OS already has a lot of flexibility in address mapping and removing a small set of mappings from its consideration may not be very important. Our experiments confirmed that; that is, with this restriction, there was no increase in the number of page faults. We expect this to be the case even in the future systems, as the main memory capacities keep increasing.

## 5 Evaluation

We evaluated the effectiveness of our approach using a simulator as well as a real state-of-the-art manycore system. The reason why we used a simulator (gem5 [10] in the full-system mode) is because we wanted to collect detailed on-chip network statistics (e.g., network latency) as well as experiments with both private and shared cache topologies. The important parameters of the target manycore architecture and their default values are given in Table 4. We later modify some of these parameters to conduct sensitivity experiments. To our knowledge, in on-chip networks, static-routing is the norm, and is widely adapted in the current commercial processors like Tilera and Xeon Phi.

[11]Fraction of iteration sets moved due to load-balancing.

**Table 3.** Benchmark properties.

| Benchmark | Number of Loop Nests | Number of Arrays | Number of Iteration Groups | Frac.[11] |
|---|---|---|---|---|
| barnes | 110 | 2 | 88,624 | 14.3% |
| fmm | 86 | 5 | 237,904 | 9.9% |
| radiosity | 164 | 19 | 189,353 | 11.2% |
| raytrace | 134 | 12 | 521,089 | 6.8% |
| volrend | 75 | 36 | 381,157 | 12.9% |
| water | 30 | 16 | 698,012 | 7.1% |
| cholesky | 128 | 51 | 411,882 | 12.2% |
| fft | 4 | 19 | 420,914 | 15.1% |
| jacobi-3d | 4 | 3 | 219,437 | 8.3% |
| lulesh | 6 | 1 | 109,086 | 8.2% |
| minighost | 4 | 1 | 97,132 | 11.7% |
| swim | 4 | 12 | 327,136 | 13.6% |
| mxm | 2 | 3 | 278,008 | 11.0% |
| art | 12 | 16 | 411,876 | 9.4% |
| nbf | 44 | 12 | 289,990 | 18.5% |
| hpccg | 4 | 4 | 78,032 | 10.4% |
| equake | 12 | 8 | 309,528 | 7.7% |
| moldyn | 2 | 6 | 220,354 | 13.9% |
| diff | 8 | 12 | 361,151 | 12.8% |

**Table 4.** System setup.

| | |
|---|---|
| Manycore Size, Frequency | 36 cores (6 × 6), 1 GHz, 2-issue |
| # of Regions, Region Size | 9(2 × 2) |
| L1 Cache | 16 KB; 8-way; 32 bytes/line |
| L2 Cache | 512 KB/core; 16-way; 64 bytes/line |
| Coherence Protocol | MOESI |
| Router Overhead | 3 cycles |
| Page Size | 2 KB |
| On-Chip Network Frequency | 1 GHz |
| Routing Policy | X-Y routing |
| DRAM | DDR3-1333; 250 request buffer entries; 4 MCs 1 rank/channel; 8 banks/rank |
| Row-Buffer Size | 2 KB |
| Data Distribution | page granularity round robin for banks cache line granularity round robin for LLC |
| Iteration Set Size | 0.25% of iterations |
| Operating System | Linux 4.7 |

We use a set of 21 multi-threaded benchmarks in this study including both regular and irregular programs: all Splash-2 applications except one [64], jacobi-3d [3D Jacobi computation], lulesh [5], minighost [3], swim [6], mxm [dense matrix multiplication], and art [6]), nbf [25], hppccg [3], equake [6], moldyn [25], and diff, a differential equation solver. Table 3 lists important characteristics of these benchmarks. The second and third columns give the number of loops and number of arrays in each code; the fourth column shows the total number of computation sets; and the last column gives the fraction of computation sets that are moved (from one core to another) as a result of load balancing.

The input sizes of these applications range between 451 MB and 1.4 GB, and their LLC (L2) misses range between 13.3% and 37.2%, averaging on 23.3%. After the warm-up phase we simulated each application either to completion or until 1 billion-instructions are executed. To determine the completion of a benchmark in gem5, we registered for an exit event from the program and dumped the stats in the callback handler of that event.

**Default Computation Mapping.** Below, we compare our approach to a default iteration set-to-core mapping. In this default (baseline) mapping, iterations of a parallel loop nest are divided into (iteration) sets and these set are assigned to cores in a round-robin fashion (note that this is essentially the same iteration set definition used in our approach). For example, if a loop nest has $Q$ iteration sets (recall from Table 4 that in our default setting, iteration set size is set to 0.25% of the total iterations in the nest) and we have $P$ cores, each core is assigned $Q/P$ iteration sets, without taking into account any location information. Note that both the default and our optimized codes use all available conventional data locality (e.g., tiling) and SIMD optimizations; they differ only in how they assign iterations to cores. In other words, the default mapping is locality aware but *not* location aware, whereas our approach is both locality and location aware.

In our gem5 experiments (with the full system mode), with the openMP codes, we use OMP_NUM_THREADS() and GOMP_CPU_AFFINITY() to set the number of threads and to pin threads to cores, respectively. With the pthreads codes, we use pthread_setaffinity_np() (numactl binary can also be used for that). Similarly, in our KNL experiments, we use KMP_HW_SUBSET (a runtime environment variable) to specify the number of sockets, number of cores, and number of threads/core. For example, KMP_HW_SUBSET=1s,6c,4t indicates 24 threads on 6 cores with 4 threads/core inside a single socket. For pinning threads to cores in pthreads applications, we use pthread_setaffinity_np() in the application code. This call maps a thread (or a set of threads) to a core (or to a set of cores). The corresponding call for OpenMP applications is GOMP_CPU_AFFINITY().

**Default Data Mapping.** As our default data mapping, we use different strategies for memory banks and last-level cache (LLC) banks. Specifically, we distribute physical addresses across memory banks in a round robin fashion at a page (memory row) granularity, which is 2KB in our case. In contrast, in an attempt to maximize parallelism across cache accesses, we distribute addresses across cache banks in a round robin fashion at a cache line size (64 bytes) granularity. Note that both these distributions are quite popular and were used by prior research [28, 57, 58]. Note also that, in gem5, depending on the address mechanism defined (e.g., cache, bank, rank bits), an LLC Miss (physical address) is routed to corresponding banks and ranks. decodeAddr() in gem5 is used to set up the rank and bank information and create a DRAM packet out of the memory request packet coming from LLC.

We later also compare our computation mapping scheme to a recent data layout optimization scheme [22] as well as a state-of-the-art computation mapping scheme [16]. To implement our compiler analysis, we used PLUTO [11]. Basically, we augmented PLUTO with three main components: CME, symbolic-analysis support, and our computation-mapping module. As mentioned earlier in Section 4, in our implementation, CME is used (as a sub-module) to (i) produce more precise MAI and CAI vectors and (ii) determine the value of the $\alpha$ parameter.

**Private Cache Results:** We first present the accuracy results in our MAI estimation. Recall that, in the case of regular applications, we use cache miss estimation to determine MAI which may not exactly match the MAI observed in execution, due to the inaccuracy in cache miss estimation. The results in Figure 7a for the regular applications indicate the error (as defined in Section 3) between the MAI vector estimated at compile time and the MAI vector observed at runtime. This error is calculated for each iteration set, and the result in Figure 7a represents the average value over all iteration sets. In the case of irregular applications on the other hand, our approach calculates MAI after the inspector code and uses them in the executor code (the remainder of execution). The results in Figure 7a for irregular applications show the average error (across all iterations) between the MAI computed at the end of the inspector code and the MAI observed at the executor phase. As can be expected, the MAI accuracy is high across all applications, with an average error value of 7.9%, indicating that the compile-time estimation of MAI is also reasonably accurate.

The first bar for each application in Figure 7b shows the average reduction in on-chip network latency when using our approach. Our first observation from these results is that our approach reduces network latency across all regular and irregular applications, with an average of 38.4%. In few programs (equake, volrend, barnes), the savings are not significant, primarily because in those applications the default mapping of iteration sets to cores perform quite well. On the other hand, in applications such as lulesh, swim and moldyn, where the default mapping of iterations performs poorly, our approach achieves very significant savings in on-chip network latencies. For each application, the second bar in Figure 7b plots the impact of these improvements of network latency on execution times. Each bar in this graph represents the percentage reduction in execution time over the default mapping strategy. We want to emphasize that, in the case of irregular applications, these results include all the runtime overheads incurred by our approach. The average performance improvement is 10.9% for all applications. At this point, we can compare these results to those presented in Figure 2 which represents the maximum performance improvement with an ideal network. The gap between the two plots (the second bar in Figure 7b and the first bar in Figure 2) is more pronounced in the case of irregular applications compared to the regular ones, due to the runtime overheads incurred in the former. The runtime overheads of our approach are quantified in Figure 7c. Each bar gives the overhead as a fraction of the overall execution time of the application. We see that these overheads range between 0.7% and 19.5%, averaging on 2.9%.

**Shared Cache Results:** We next present the results collected with a shared LLC. Figure 8a gives the average MAI accuracy and CAI accuracy results for our applications. We observe an average MAI (CAI) error value of 11% (14%). The

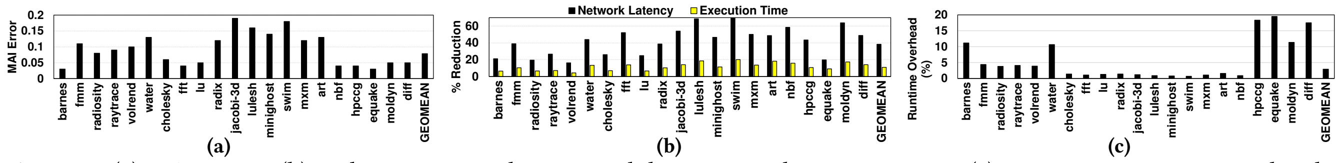


**Figure 7.** (a) MAI errors. (b) Reduction in on-chip network latencies and execution time. (c) Percentage runtime overheads.

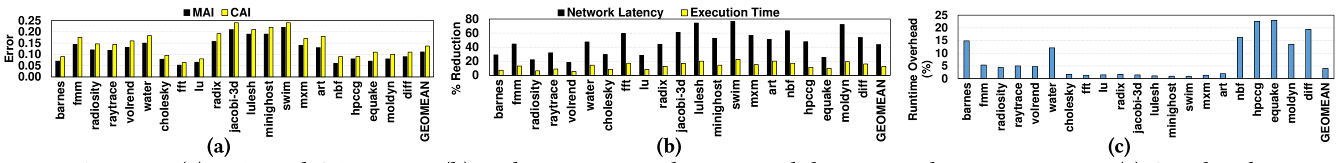


**Figure 8.** (a) MAI and CAI errors. (b) Reduction in on-chip network latency and execution time. (c) Overheads.

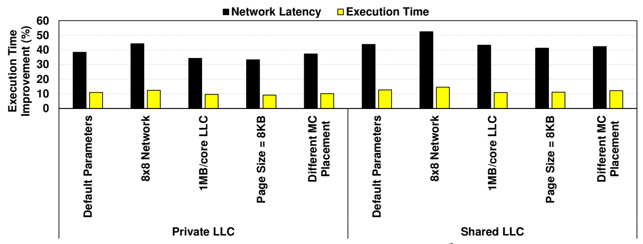
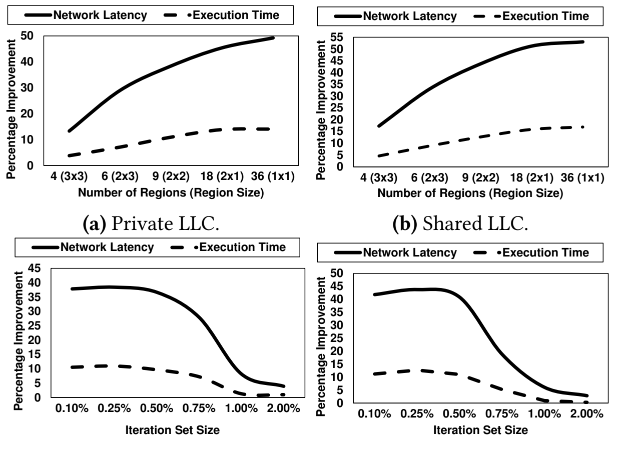


**Figure 9.** Sensitivity results.



**(a)** Private LLC. **(b)** Shared LLC.

**(c)** Private LLC. **(d)** Shared LLC.

**Figure 10.** Results showing sensitivity to the number of regions and iteration sets.

savings in on-chip network latencies are plotted as the first bar for each application in Figure 8b, and indicate an average network latency improvement of 43.8% across all applications. The corresponding execution time saving – plotted as the second bar for each application in Figure 8b – is 12.7%, on average. We see that these improvements (both in network latencies and execution times) are higher than the corresponding average improvements in the case of private LLCs. The main reason for this is the fact that, in the case of shared caches, the on-chip network experiences much more traffic due to remote L2 accesses in S-NUCA. Consequently, the default iteration mapping, which does not do anything specific for optimizing network traffic, performs quite poorly, and consequently, our approach generates higher savings compared to the default mapping. Figure 8c gives the dynamic overheads of our approach, as a fraction of the overall execution time. These values are similar to those presented earlier for the private cache case.

**Varying Simulation Parameters:** In the interest of space, for each experiment we only report the geometric mean of the network latency and execution time improvements for the private and shared cache cases. We present the results when key hardware parameters are modified in Figure 9 (in each experiment, only one parameter is modified). It can be observed from these results that (i) increasing the network size (and the number of cores) increases the performance improvements brought by our approach. This is mainly because a larger network makes network optimization more important as a result of the increase in distance between the requester core and data location (main memory or last-level cache, in the shared LLC case); (ii) as can be expected, increasing the LLC capacity tends to reduce our savings since doing so reduces the volume of the message traffic in the network; (iii) a larger page size (8KB) brings lower benefits since a larger page size causes the data accesses to use a smaller fraction of the on-chip network and this tends to reduce opportunities for our approach; (iv) changing the location of the memory controllers does not have a significant impact on our savings. In this experiment, we placed the four memory controllers in the middle of each side of the 2D space, instead of the intersection of the sides (corners), and the savings achieved in both the cases are quite similar.

The results with different region sizes are plotted in Figures 10a and 10b. We see that for both the private LLCs and shared LLCs, going beyond a certain number of regions does not bring much additional benefits. On the other hand, as expected, working with a very few (but large) regions performs quite poorly as doing so causes some loss in location awareness. In Figures 10c and 10d, we plot the impact of changing the iteration set size. Recall from Table 4 that the iteration set size we employed so far in our experiments was 0.25% of the total number of iterations (in each loop nest). We see that, it is important to work with small iteration set sizes, as increasing the set size smoothes the differences in the access patterns of different iteration sets, and this in turn

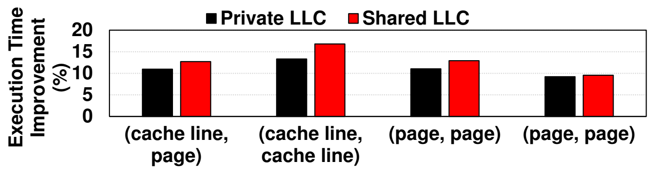


**Figure 11.** Results with different combinations of physical address distribution over (memory banks, cache banks).

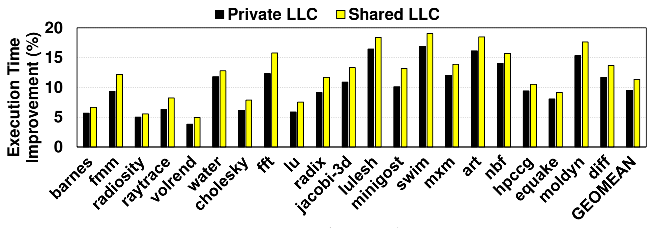


**Figure 12.** Results with DDR-4.

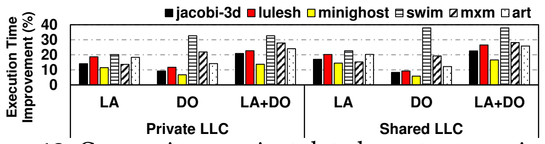


**Figure 13.** Comparison against data layout reorganization. LA denotes our approach and DO denote the scheme in [22].

makes iteration scheduling less critical. However, working with a very small value (e.g., 0.1%) is not a good idea either, because in that case the dynamic overheads seen in irregular applications start to play a larger role in shaping the application performance.

The next parameter we change is the distribution of data (physical addresses) across memory banks and cache banks. Recall from Table 4 that, our default distribution of physical addresses across memory banks is round robin with page size granularity, and that across caches is round robin with cache line size granularity. In principle, other combinations are also possible, and the corresponding results are plotted in Figure 11. Due to space concerns, for each combination, we only report geometric mean across all 21 applications, and (cache line, page) corresponds to our default distribution. It can be observed from these results that our approach performs quite well in all combinations.

It is to be emphasized that, while we expect our approach to generate better results than the default computation mapping under any give distribution of data (addresses) across memory banks, it is still an open question how one should *co-optimize* computation and data distributions together to maximize overall performance. Since computation and data distributions are tightly coupled, a co-optimization approach can be promising and will be part of our future research.

Note that, so far, we reported results with DDR-3. In Figure 12, the execution time improvements with DDR-4 are presented. Although these savings are a bit lower compared to the DDR-3 results, they still represent significant improvements over the default mapping (when it uses DDR-4), resulting in 9.5% and 11.4% average improvements, for private and shared LLCs, respectively.

We also collected results using a SIMPOINT [51] like approach that considers phase behavior of the application being simulated. The collected results are similar to those obtained via our default (1 billion instruction) approach, hence, we do not report the detailed results here.

Overall, these sensitivity experiments demonstrate that our approach performs very well across the different values of the major simulation parameters.

**Comparison against Alternate Approaches:** In this part, we first quantify the benefits of our computation mapping strategy when it is used along with a data layout optimization strategy, and then compare our approach to a previously published computation mapping scheme.

We are not aware of any data layout optimization strategy that targets at reducing on-chip network traffic in irregular applications. A recent work [22] tries to reduce network traffic by determining a suitable memory layout of data for regular application programs. Such layout optimizations are inherently limited in the sense that, while different loop nests accessing the same data structure may demand different memory layouts, a practical scheme needs to select a single layout (for each data structure) to be used for the entire program. Consequently, the selected layout may not be very desirable (let alone being optimal) for every loop nest in the code. We were able to run only 6 of our applications with the scheme in [22]. The results given in Figure 13 show that, in 4 of the applications tested, our approach (denoted LA) outperforms the scheme in [22] (denoted DO in the graph), while the latter performs better in the remaining two (swim and mxm). The LA+DO version in the graph represents an optimization strategy in which first DO is applied and then our scheme. We see that doing so brings additional benefits in all but one benchmark. In swim, data layout optimization performs extremely well which leaves no further scope for our scheme to bring any additional benefits.

We also compared our approach to a recent OS-/hardware-based computation mapping scheme [16] that targets reducing distance-to-data. While the scheme in [16] is for multiprogrammed-workloads, one can treat each thread of a multithreaded-application as if it is a separate application and use [16] for thread/application-to-core assignment. The results are presented in Figure 14 as the last two bars for each application (the first two bars reproduce the corresponding results our compiler-based approach achieves). We can see that [16] does not perform well in the shared LLC case, mainly because, in an S-NUCA system, for an application with significant miss-rate, it is the L2-to-MC distance that plays the biggest role (not the core-to-MC distance which is the main target of [16]). On the other hand, [16] performs, as expected, better with private LLCs. Still, our approach generates better results than [16]. This is because, [16] works best when different applications in the workload have different

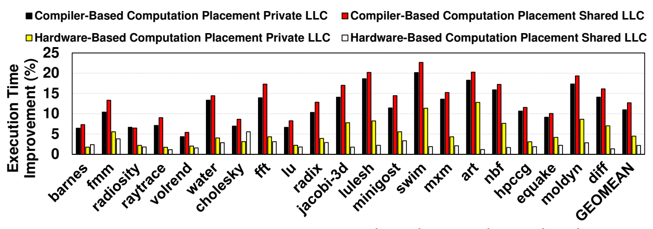


**Figure 14.** Comparison against a hardware-based scheme.

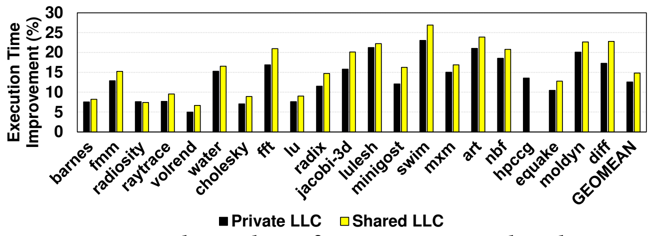


**Figure 15.** Results with perfect MAI, CAI and cache miss estimation.

characteristics/memory intensities/network intensities. But, different threads of a multi-threaded application generally have similar network/memory intensities, diminishing the performance differences across different thread-to-core assignments. Our approach, on the other hand, collocates the iteration-sets that heavily share data in between in the same region, thereby leading to better performance.

**Optimality:** While in general it is difficult to define "optimal computation mapping" in the context of manycore environments running regular and irregular applications, one way of mimicking it using the simulator is to assume perfect MAI and CAI estimations. Assuming 100% accuracy in MAI- and CAI-estimation as well as perfect cache miss estimation, we collected execution time savings under both the private and shared cache configurations, and the results are plotted in Figure 15. We see that these results are not much better than the corresponding savings when we do not have perfect MAI, CAI and cache miss estimation, indicating that our approach performs quite well in practice.

While not presented here, we also tested the effectiveness of our approach when running multiple multi-threaded applications (each optimized using our approach) at the same time. Our experiments revealed that the proposed approach brings, on average, around 18.1% improvement in the case of private caches and 26.7% in the case of shared caches.

**Results with Intel KNL:** While this work is a primarily simulation-based study as it is hard to get detailed on-chip network statistics from real manycore systems and we wanted to change some hardware parameters, we also evaluated our approach on an Intel KNL (Knight's Landing) manycore and measured the execution times. KNL is based on a 2D mesh network. This architecture is organized into 36 tiles, each containing two cores (allowing concurrent execution of up to 240 threads), featuring 32KB data cache and a pair of custom 512-bit AVX vector units. Each tile also has an L2 bank of 1MB. KNL also features a high-bandwidth near memory (more details can be found in [7]). KNL has

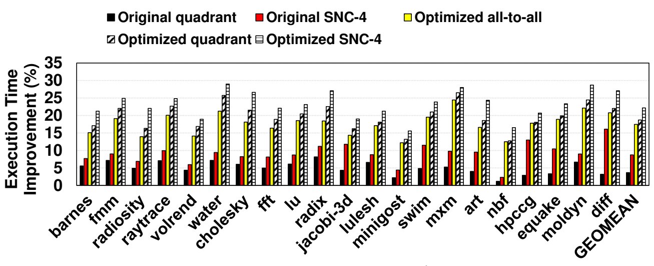


**Figure 16.** KNL results.

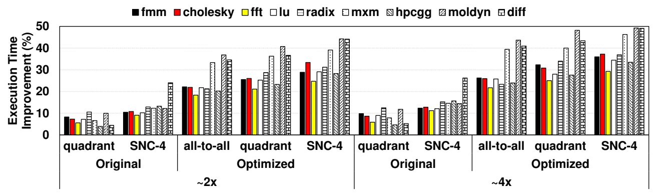


**Figure 17.** Results with different input sizes.

three different cluster modes, which are essentially ways of dividing the chip into separate virtual regions, with the goal of keeping the on-chip communication as local as possible. In the all-to-all mode, addresses are uniformly hashed over all memories and caches independently. In the quadrant mode, the chip is divided into 4 (virtual) quadrants and a memory access travels within the same quadrant. Note that this mainly optimizes for main memory accesses. Finally, in the SNC-4 mode, each quadrant is exposed as a separate cache-coherent NUMA cluster.

The graph in Figure 16 gives the reduction in execution cycles with and without our approach, and all values are with respect to the execution times in the all-to-all mode without our approach.[12] One can see that, our approach improves the performance of the all-to-all mode significantly, and actually makes it even better than the quadrant mode without our approach. This is because the quadrant mode mainly optimizes for main memory accesses whereas our approach optimizes cache accesses as well. Our approach also improves over the SNC-4 mode (22.2% on average). It is even more interesting to note that the all-to-all mode with our approach performs better than the SNC-4 without our approach (by 8.8% on average). This is mainly because the SNC-4 mode can lead to significant contention on some links while aggressively enforcing data locality. In any case, the best results are obtained when our compiler-based approach is combined with SNC-4.

Figure 17 gives the KNL results when the input sizes of certain applications are increased. We were able to increase the input sizes of only 9 of our applications without creating any correctness issue. The left half of Figure 17 reports the results when the input size is approximately 2x of the original, and the right half gives the same results with a 4x increase. As can be expected, our approach performs better with larger input sizes. This is primarily due to the fact that

[12] All the (original/optimized) versions tested on KNL use all conventional data locality (e.g., loop permutation, tiling, unrolling) as well as low-level parallelism (e.g., SIMD) optimizations.

the unoptimized codes perform worse with larger inputs, and consequently, the relative improvements brought by our approach increase. We also tested our approach when running four multi-threaded applications at the same time on Intel KNL. Our approach brought about 22% performance improvement (on average) over the SNC-4 mode.

## 6 Related Work

**Hardware and OS Based Optimizations:** Prior works including [29, 34–37, 43, 54, 59, 65, 66] apply hardware/OS based optimizations. Beckmann et al. [9] proposed a mechanism to co-schedule data and threads on a 36-tile system such that the on-chip data movement is minimized. In S-NUCA, higher LLC hit rates may not necessarily translate to higher performance because of the on-chip traffic incurred. In fact, as shown by our results, the proximity between a thread and its data can play a major role in shaping the overall application performance. Mishra et al. [47] suggested that having separate bandwidth-optimized and latency-optimized physical networks can improve performance by reducing end-to-end memory access latencies. Since the on-chip interconnect latencies are reduced in this approach, overall on-chip throughput is improved thereby improving system performance. Das et al. [17] proposed router privatization policies to accelerate performance-critical packages on network and eventually improving performance. Zhuravlev et al. [68] developed a classification scheme to address the contention of shared resources on NoC. The focus of most existing hardware-based studies is multi-programmed workloads. In contrast, we focus on multi-threaded workloads. Das et al. [16] proposed an approach that maps applications to cores such that the inter-application interference on shared resources is minimized. Focusing on multi-programmed workloads, their idea is to map the applications which are highly memory intensive and network-sensitive to cores which are close to the memory controller. As our experiments indicated, this approach does not work well for S-NUCA systems and for applications with high LLC hit rates, since the threads of such applications can still make lots of remote (on-chip) cache accesses. Dashti et al. [20] proposed a memory placement algorithm that reduces network congestion in NUMA architectures. We believe that the OS-based and compiler-based solutions are complementary and can be used together.
**Software-Based Proposals:** A number of previous compiler works [12, 15, 26, 27, 32, 33, 38, 39, 49, 56, 60, 62] explored the ways to improve data locality (cache performance). The main goal of these works is to achieve either temporal reuse or unit-stride (or small-stride) accesses for as many array references as possible so that the access pattern of data can be aligned with its memory layout. These works mainly try to minimize cache misses, and implicitly assume that all cache hits take the same amount of time to service and, similarly, all cache misses have the same latency (miss latency is typically assumed be much higher than hit latency though). Note that this specific latency assumption does not hold in current manycores where there are many caches and memory controllers/memory banks distributed all over the chip. In these architectures, both caches and memory controllers are accessed using an on-chip network and thus, from a given core's perspective, different cache hits (and similarly different misses) can have very different access latencies, dictated primarily by the core's proximity to the cache bank or memory bank that hold the requested data.

The other group of software based work [8, 21, 24, 40, 42, 44, 61, 63] focus on NUMA (non-uniform memory access) systems. These works clearly have some "vicinity" concept built in them; however, this concept is mostly built upon a local-remote dichotomy, without any precise location information, that is, different degrees of remoteness are not considered. If a data access cannot be performed locally, NUMA-centric works do not care from which remote node the requested data access will be satisfied. In comparison, our approach takes advantage of relative locations, in the 2D NoC space, of cores, caches and memory controllers/banks. Note also that NUMA strategies and our approach are in a sense complementary; in a shared memory multiprocessor, we can use NUMA techniques for inter-node optimization and our approach for intra-node (inter-core) optimization.

## 7 Concluding Remarks

In many previously-proposed compiler works that target multicores or manycores, relative positions of cores, last-level caches and memory controllers are not explicitly taken into account, and consequently, a lot of optimization opportunities are lost. This paper shows that, by being location aware, i.e., considering the positions of cores, caches and memory controllers, a compiler can achieve significant improvements in parallel application execution times. Specifically, we propose a strategy to assign/map parallel computations to cores in on-chip network based manycore systems. The results collected using 21 multi-threaded applications (including both regular and irregular codes) reveal that, our approach reduces execution times by 10.9% and 12.7%, on average, for private and shared LLCs, respectively. Our future work includes co-optimizing computation and data mapping together.

## Acknowledgment

We thank Kathryn S. McKinley for shepherding our paper. We also thank the anonymous reviewers for their constructive feedback. Jung is supported in part by NRF 2016R1C1B2015312, DOE DEAC02-05CH 11231, IITP-2017-2017-0-01015, NRF2015M3C4A7065645, and MemRay grant (2015-11-1731). This research is supported in part by NSF grants #1439021, #1629915, #1626251, #1409095, #1629129, #1526750, #1439057, and a grant from Intel.

## References

[1] 2007. Intel teraflops research chip. goo.gl/lewCk7.

[2] 2009. Intel Single-cloud chip. goo.gl/RSJjfg.

[3] 2012. minighost. https://mantevo.org/default.php.

[4] 2012. The Architecture and Performance of the TILE-Gx Processor Family. http://www.tilera.com/products/processors/TILE-Gx_Family.

[5] 2013. CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks/

[6] 2013. SPEC OMP 2001. https://www.spec.org/omp2001/

[7] 2014. Intel Xeon Phi processor. goo.gl/3DVc9T.

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

[9] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling. In *Proceedings of the 21st international symposium on High Performance Computer Architecture (HPCA)*.

[10] 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 Comput. Archit. News* (2011).

[11] 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)*.

[12] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *ASPLOS*.

[13] M. Chaudhuri. 2009. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *Proceedings of High Performance Computer Architecture (HPCA)*.

[14] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2003. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO)*.

[15] MichałCierniak and Wei Li. 1995. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *PLDI*.

[16] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*.

[17] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2010. Aergia: Exploiting Packet Latency Slack in On-chip Networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*.

[18] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. 1994. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *J. Parallel Distrib. Comput.* (1994).

[19] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. 1994. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *J. Parallel Distrib. Comput.* (1994).

[20] 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 *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] 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*.

[22] 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*.

[23] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. Program. Lang. Syst. (TOPLAS)* (1999).

[24] 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*.

[25] Hwansoo Han and C.-W. Tseng. 2006. Exploiting locality for irregular scientific codes. *Parallel and Distributed Systems, IEEE Transactions on* (2006).

[26] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prith Banerjee. 1999. A matrix-based approach to global locality optimization. *J. Parallel and Distrib. Comput.* (1999).

[27] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. 2001. A layout-conscious iteration space transformation technique. *IEEE Trans. Comput.* (2001).

[28] 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*.

[29] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2016. uC-States: Fine-grained GPU Datapath Power Management. In *PACT*.

[30] Changkyu Kim, Doug Burger, and Stephen W Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ACM SIGPLAN Notices*.

[31] John Kim, James Balfour, and William Dally. 2007. Flattened Butterfly Topology for On-Chip Networks. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.

[32] 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*.

[33] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-centric Multi-level Blocking. In *PLDI*.

[34] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das. 2016. Re-NUCA: A Practical NUCA Architecture for ReRAM Based Last-Level Caches. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[35] J. B. Kotra, D. Guttman, N. C. N., M. T. Kandemir, and C. R. Das. 2017. Quantifying the Potential Benefits of On-chip Near-Data Computing in Manycore Processors. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[36] J. B. Kotra, S. Kim, K. Madduri, and M. T. Kandemir. 2017. Congestion-aware memory management on NUMA platforms: A VMware ESXi case study. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*.

[37] Jagadish B. Kotra, Narges Shahidi, Zeshan A. Chishti, and Mahmut T. Kandemir. 2017. Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[38] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS*.

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

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

[41] Yong Li, A. Abousamra, R. Melhem, and A. K. Jones. 2012. Compiler-Assisted Data Distribution and Network Configuration for Chip Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* (2012).

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

[43] Jun Liu, Jagadish Kotra, Wei Ding, and Mahmut Kandemir. 2015. Network Footprint Reduction Through Data Access and Computation Placement in NoC-based Manycores. In *Proceedings of the 52Nd Annual Design Automation Conference*.

[44] Henrik Löf and Sverker Holmgren. 2005. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *ICS*.

[45] Qingda Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 2009. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *Proceedings of the Parallel Architectures and Compilation Techniques (PACT)*.

[46] Amirhossein Mirhosseini, Mohammad Sadrosadati, Ali Fakhrzadehgan, Mehdi Modarressi, and Hamid Sarbazi-Azad. 2015. An Energy-efficient Virtual Channel Power-gating Mechanism for On-chip Networks. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[47] Asit K. Mishra, Onur Mutlu, and Chita R. Das. 2013. A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*.

[48] Asit K. Mishra, N. Vijaykrishnan, and Chita R. Das. 2011. A Case for Heterogeneous On-chip Interconnects for CMPs. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*.

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

[50] 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 *PACT*.

[51] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

[52] A. Sharifi, E. Kultursay, M. Kandemir, and C.R. Das. 2012. Addressing End-to-End Memory Access Latency in NoC-Based Multicores. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.

[53] David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. 2009. Millisecond-scale Molecular Dynamics Simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*.

[54] 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*.

[55] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* (2016).

[56] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI*.

[57] 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)*.

[58] 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*.

[59] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture*.

[60] S. Verdoolaege, M. Bruynooghe, G. Janssens, and P. Catthoor. 2003. Multi-dimensional incremental loop fusion for data locality. In *ASAP*.

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

[62] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *PLDI*.

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

[64] 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)*.

[65] Haibo Zhang, Prasanna Venkatesh Rengasamy, Nachiappan Chidambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2018. FLOSS: FLOw Sensitive Scheduling on Mobile Platforms. In *In Proceedings of The Design Automation Conference (DAC)*.

[66] 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*.

[67] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2002. Breaking Address Mapping Symmetry at Multi-levels of Memory Hierarchy to Reduce DRAM Row-buffer Conflicts. In *The Journal of Instruction-Level Parallelism (JILP)*.

[68] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.