Data Movement Aware Computation Partitioning

Xulong Tang[1], Orhan Kislal[1], Mahmut Kandemir[1], Mustafa Karakoy[2]
[1]The Pennsylvania State University, USA, [2]TOBB University of Economics and Technology, TURKEY
{xzt102,omk103,kandemir}@cse.psu.edu,m.karakoy@yahoo.co.uk

ABSTRACT

Data access costs dominate the execution times of most parallel applications and they are expected to be even more important in the future. To address this, recent research has focused on Near Data Processing (NDP) as a new paradigm that tries to bring computation to data, instead of bringing data to computation (which is the norm in conventional computing). This paper explores the potential of compiler support in exploiting NDP in the context of emerging manycore systems. To that end, we propose a novel compiler algorithm that partitions the computations in a given loop nest into subcomputations and schedules the resulting subcomputations on different cores with the goal of reducing the distance-to-data on the on-chip network. An important characteristic of our approach is that it exploits NDP while taking advantage of data locality. Our experiments with 12 multithreaded applications running on a state-of-the-art commercial manycore system indicate that the proposed compiler-based approach significantly reduces data movements on the on-chip network by taking advantage of NDP, and these benefits lead to an average execution time improvement of 18.4%.

CCS CONCEPTS

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

KEYWORDS

Multicore Architectures, Near-Data Computing, Compiler

ACM Reference format:
Xulong Tang[1], Orhan Kislal[1], Mahmut Kandemir[1], Mustafa Karakoy[2]. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 15 pages.
https://doi.org/10.1145/3123939.3123954

1 INTRODUCTION

As many applications are increasingly being dominated by the cost of data movement (as opposed to the cost of arithmetic/logic operations), we witness a shift from the traditional "compute-centric" model to a more "data-centric" model, where the primary goal is to bring computation to data, instead of the other way around. This shift has led to a new computing paradigm, namely, "Near-Data Processing" (NDP), and we already see its various incarnations across different architectures and application domains [2, 9, 12, 22, 36, 40, 48, 52, 53, 58].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123954

To fully evaluate the potential of NDP, one needs to consider it from multiple angles, including but not limited to programming aspects, compiler support and system-level modifications. While most of the existing work on NDP has either focused on single-core systems [10, 20], storage subsystems [55, 56, 65] or systems that employ inter-chip parallelism [14, 48, 54, 61], a single manycore chip can also present significant opportunities for taking advantage of NDP. In particular, different cache hits on a large, on-chip network based manycore can have significantly different latencies depending on the distance between the requesting core and the cache that contains the requested data. Similarly, the distance between a last-level cache and a memory controller can be a major determinant of the latency of the last-level cache misses. In a sense, a different version of the NDP-related problems observed in large-scale systems is also observed in a single manycore chip.

Motivated by this, the main goal of this paper is to explore the potential of "compiler support" in exploiting NDP in the context of data-intensive and loop-dominated applications. We believe that, if the architectural details of the target manycore system could be exposed to it, a compiler can employ various computation and/or data reorganization techniques to exploit the opportunities presented by NDP. To that end, we propose and evaluate a novel compiler algorithm that "partitions" the computations (e.g., program statements) in a given loop nest into "subcomputations" and schedules the resulting subcomputations on different cores with the goal of reducing the "distance-to-data". As opposed to the conventional optimizations that try to bring data to computation, our approach tries to bring (by scheduling subcomputations close to where data are) computations to data. To do so, it employs "data movement" as a first-class optimization metric. However, minimizing data movement through subcomputation scheduling may lose the opportunity for reusing the data (from the L1 cache) in subsequent program statements. To address this, our approach also takes data locality into account when considering multiple statements together. That is, our approach targets both NDP and data locality. Our **contributions** can be summarized as follows:

- Focusing on multithreaded applications running on on-chip network based manycore systems, we explain the potential benefits of careful computation partitioning in terms of data movements on the network. We argue that computation partitioning can be an effective knob in realizing NDP.
- We propose a compiler algorithm that takes a loop nest to be executed on an on-chip network-based manycore, partitions it into subcomputations, and assigns each subcomputation to a core where NDP could be exploited. We formulate the problem of exploiting NDP as a Minimum Spanning Tree (MST) problem, and our compiler solves it using Kruskal's algorithm. Moreover, we explain how our approach takes advantage of L1 cache locality to further reduce data movements when considering multiple nearby program statements (e.g., those in a loop body) together.

- We present experimental evidence showing the effectiveness of the proposed compiler-based approach. Our experiments with 12 multithreaded application programs running on a state-of-the-art on-chip network based manycore indicate that the proposed compiler-based approach significantly reduces data movements on the on-chip network, and these reductions lead to an average execution time improvement of 18.4%. The results also indicate that our savings are consistent across the different values of the major configuration parameters of our target manycore platform.

We believe this is the first fully-automated compiler support targeting NDP/data movement in the context of data-intensive applications running on commercial manycores. The remainder of this paper is organized as follows. The next section gives background on on-chip network based manycore systems and physical address mappings they employ. Section 3 formulates data movement problem we target and Section 4 gives a compiler algorithm that employs data movement minimization as its main optimization metric. We go over several examples in Section 5 to explain how our compiler algorithm works in practice. Section 6 presents our experimental results, and Section 7 discusses the related work. We conclude the paper in Section 8 and briefly mention the planned future work.

2 BACKGROUND

Our target platform is an on-chip network based manycore architecture with $M \times N$ nodes, as shown in Figure 1. Each node of this architecture contains a core, a private L1 cache, and a bank of L2 cache (our last-level cache). The entire shared L2 cache is divided into banks that are distributed among nodes. We assume an SNUCA [33] type of cache organization in our approach, but our approach can be modified to work with other distributed on-chip cache management schemes as well. In SNUCA, each data block is statically mapped to an L2 cache bank (home bank) based on its physical address. A node that requests a specific data item brings it from its home bank. To facilitate our explanation, each node is labeled with (x, y), which indicates the location of the node in the on-chip network. When no confusion occurs, we use the term "n_v" to denote the home node for data "v". Memory controllers (MCs) are attached to the corner nodes.

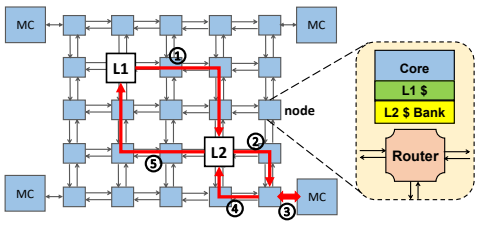


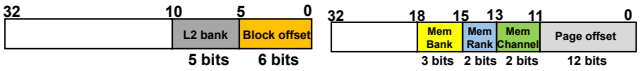
Figure 1: An example manycore architecture and memory access flow.

A typical data request in this architecture is handled as follows. When an L1 miss occurs, the request is forwarded to the node that has the home L2 bank (shown as ① in Figure 1). If hit in home L2 bank, it is sent back to the L1 cache (⑤). Otherwise, an L2 miss is forwarded to the target MC (②,③). The MC schedules the request and sends the corresponding data to the home L2 bank (④), and subsequently to the L1 cache of the requesting core.(note that, in general, the requesting core and the home L2 bank are in different nodes). There are two time-consuming periods involved in a memory access: 1) time spent on traveling over the on-chip network, and 2) time spent on accessing the off-chip memory. The former is a function of three factors: the number of network links to traverse, the data volume, and the congestion on the network. In this paper, our goal is to reduce the values of the first two factors by exploiting NDP.

We use Manhattan distance between two nodes in the on-chip network to measure the data movement distance. Given two nodes $n_{i,j}$ and $n_{x,y}$ with location labels (i,j) and (x,y), the data movement distance between $n_{i,j}$ and $n_{x,y}$ is defined as:

$$MD(n_{i,j}, n_{x,y}) = | i - x | + | j - y |.$$

This distance gives the "minimum" number of links that need to be traversed from $n_{i,j}$ to $n_{x,y}$. A long data movement distance can hurt performance because of two reasons. First, a longer distance increases the latency on the network, and second, it also increases chances for contention.



(a) Cacheline-level mapping at L2 cache. **(b) Page-level mapping at memory bank.**

Figure 2: Physical address mapping.

Physical address mapping defines how the physical address space is distributed across multiple shared components (e.g., L2s, memory channels, banks). There are two popular address mapping strategies: 1) cache line-granularity mapping, and 2) page-granularity mapping. Each of these mappings can be used for mapping addresses across different components such as cache banks and memory banks.[1]. Figure 2a depicts a cache line-granularity mapping of addresses over L2 caches, where a bank of the L2 cache is indexed using 5 bits (bits 6, 7, 8, 9 and 10). Figure 2b gives a page-granularity mapping of physical addresses over 4 memory controllers (channels), 4 memory ranks per channel, and 8 memory banks per rank. The least 12-bits of a physical address represent the offset, assuming a 4KB page. The next 2 bits (bits 12 and 13) represent the channel ids. In a specific channel, 2 bits (bits 14 and 15) are used to identify 4 individual ranks. Finally, 3 bits (bits 16, 17 and 18) represent the related memory banks.

Note that the manycore architecture defined above represents a "template" and our approach is still applicable when the values of the specific components of this template are varied. For instance, we can work with any type of on-chip cache hierarchy, any type of on-chip network topology, any type of last-level cache management scheme, and any type of off-chip memory request scheduling policy.

3 DATA MOVEMENT AS A MINIMUM SPANNING-TREE PROBLEM

3.1 Importance of Computation Partitioning

One of the important tasks in executing computations on any manycore system is to partition them across cores/nodes. Recent literature contains many papers [4, 18, 28, 35] that try to do this partitioning targeting various metrics such as latency, throughput, and power consumption. One of the other potential metrics is reducing

[1]For example, one can distribute addresses at a cache line or page granularity over caches, and similarly, at a cache line or page granularity over memory banks.

the distance (in terms of the number of links in the on-chip network) between the requesting core/node and the node (the L2 bank) that contains the requested data. However, all existing partitioners today targeting loop-based computations perform this assignment at an "iteration granularity", no matter what objective function they have. That is, the minimum unit of computation assigned to a core is a "loop iteration". The core that is assigned a loop iteration is responsible for bringing all the input data (which may involve a lot of data transfers over the on-chip network), computing the result, and storing the result in the store node (i.e., the home node for the output data in SNUCA). Clearly, this standard (default) way of assigning computations to cores can lead to 1) a high volume of data traveling over the network and 2) a large number of network links being traversed.

In this work, we consider a novel approach to reduce data transfer costs in emerging manycore processors. Our approach tries to *minimize* both the data volume and the number of links by performing computation partitioning (assignment) at a "subcomputation" granularity. More specifically, our scheme partitions a program statement (which may be in a loop body) into subcomputations and performs each subcomputation in a core/node that results in the "minimum" data movement. Also, whenever possible (i.e., permissible by data dependences), subcomputations are performed in *parallel* and the correctness of execution is guaranteed via carefully-placed synchronizations. To capture both data volume and distance, we formally define *data movement* as follows:

$$Data\ Movement = \sum_{i=1}^{m} sizeof(V_i) \times MD(n_{V_{req}}, n_{V_i}), \qquad (1)$$

where m denotes the total number of input data, and n_V denotes the node id in the on-chip network. For each input data V_i in a statement, data movement, our primary target metric in this work, is computed as the data size of V_i multiplied by the Manhattan distance between $n_{V_{req}}$ (the node requesting V_i) and n_{V_i} (the node contains V_i). As long as the size of V_i is constant, we omit it in our discussion, and use the expressions "data movement", "network footprint", and "number of links traversed" interchangeably.

In the default execution (i.e., when not using our approach), the requesting node $n_{V_{req}}$ must be the node where the original computation is performed. We relax this constraint in our optimization where the target node can be an *intermediate node* performing a "subcomputation".

To better explain the concept of data movement, let us consider a statement in one loop iteration (e.g. i^{th} iteration):

$$A(i) = B(i) + C(i) + D(i) + E(i).$$

As shown in Figure 3, the input data (which are all of the same size) are assumed to be present in the L2 caches of the corresponding home nodes (if not, the data would be present in memory banks but a similar analysis applies to that case too). In the default execution, all of the input data ($B(i)$, $C(i)$, $D(i)$, and $E(i)$) are fetched into $n_{A(i)}$ for computation and the final result is stored in $n_{A(i)}$ (i.e., the node $n_{A(i)}$ is also assumed to be the node to which computation of $A(i) = B(i) + C(i) + D(i) + E(i)$ is assigned). As a result, we have 13 data movements based on Equation (1) (i.e., the number of links visited is 13). However, one can observe that there are overlapping network paths while fetching each input data to $n_{A(i)}$. Specifically, $B(i)$ and $E(i)$ traverse the same two links from $n_{B(i)}$ to

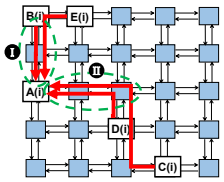
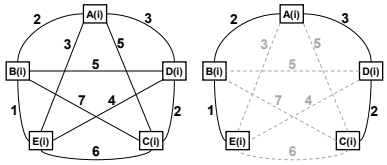


Figure 3: Data access for $A(i) = B(i) + C(i) + D(i) + E(i)$.



(a) Graph representing the network links of a program statement. **(b) Corresponding MST.**

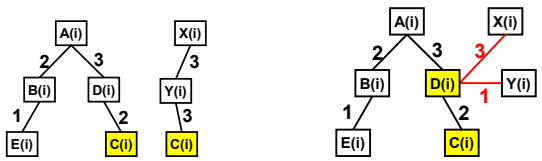
Figure 4: Formulation of NDP as MST.

$n_{A(i)}$ (❶). Therefore, instead of loading $B(i)$ and $E(i)$ individually, we can compute $B(i) + E(i)$ in $n_{B(i)}$ and send the result to $n_{A(i)}$. Clearly, doing so reduces the data movements by 2. Similarly, we can perform $C(i) + D(i)$ in $n_{D(i)}$, and consequently reduce 3 data movements (⓫). As a result, the total amount of data movement is reduced to 8.

3.2 Minimizing Data Movement

We formulate the problem of minimizing data movement on the on-chip network as a Minimum Spanning Tree (MST) problem and solve it using Kruskal's algorithm. It is important to emphasize that, for now, we apply our MST solution to each single statement in isolation. Later, we will discuss how multiple statements are considered together to take advantage of both NDP and data reuse. We believe that MST is a very good fit for our data movement problem because minimizing data movement essentially means minimizing the number of on-chip network links that need to be traversed by data elements. We build a complete graph for each program statement in a given loop iteration. In such a graph, the "vertices" represent "nodes", the "edges" represents "network links", and the "weight costs of edges" represent "network distances" between two nodes. We use Kruskal's algorithm to find the MST that connects all the vertices together with a minimum cumulative edge weight, which gives us the minimum data movements on the network. For example, Figure 4a shows the generated complete graph for same statement used in Figure 3. The weight costs labeled on edges are the Manhattan distances between nodes where the data locates. The resulting MST is illustrated in Figure 4b, with the minimum edge costs, which also means the minimum data movement.

3.3 Exploiting Data Reuse



(a) Single-statement MSTs of 2 nearby statements. **(b) Multiple-statement MSTs with data locality.**

Figure 5: Data reuse aware MST formulation.

An important observation is that, considering multiple statements (statement instances[2]) at the same time allows us to 1) exploit *data reuse* across nearby statements, and 2) further reduce the number of on-chip network links to be visited by taking advantage of L1 cache hits. For example, Figure 5a shows two MSTs that belong to two neighboring statements. Note that $C(i)$ is reused by statement-2 ($X(i) = C(i)+Y(i)$) after statement-1 ($A(i) = B(i)+C(i)+D(i)+E(i)$). If $C(i) + D(i)$ in statement-1 is scheduled on node n_{Di}. That is, $C(i)$ is fetched to n_{Di} and placed into the L1 cache of n_{Di}. Therefore, n_{Di} is also considered having $C(i)$ in its L1 cache when we generate the MST for statement-2. Figure 5b depicts the MST when using n_{Di} for statement-2. The total data movement of the two statements is reduced compared to the case where $C(i)$ is fetched from n_{Ci} to n_{Yi} for computation.

4 COMPILER ALGORITHM

In this section, we discuss the details of our proposed compiler framework The **goal** of our approach is to 1) minimize the data movement of on-chip network for each single statement in a loop nest, 2) exploit L1 cache data locality across multiple statements, and 3) balance the computation across all the cores. There are three major steps of our approach (Algorithm 1): 1) data location detection, 2) single statement splitting, and 3) subcomputation scheduling.

4.1 Data Location Detection

We build a graph for each program statement to facilitate our subsequent optimizations. To build that graph, the very first task is to determine the nodes holding the data involved in a statement. We derive the on-chip network location of each program data by calling a *GetNode* function (line 11). This function takes two parameters: 1) a variable (data) and 2) a variable-to-node mapping structure. The latter is used in the multi-statement case (discussed later) to indicate the nodes that contain the requested data in their respective L1 caches (i.e., the data have already been fetched into the node due to the previously-scheduled subcomputations). In SNUCA, the L2 bank bits and memory channel bits in a physical address (as illustrated in Figure 2) determine which on-chip node (L2 cache bank) and memory controller hold the requested data. To prevent the memory channel bits from changing during the virtual address-to-physical address (VA-to-PA) translation, we need help from the OS. Normally, physical page allocation is performed by the OS APIs using a page-coloring based algorithm. We modify this API to preserve the cache bank and memory channel bits from the virtual address while allocating the physical pages. In other words, the OS support guarantees that our compiler infers the on-chip data location from its virtual address. Obviously, the requested data may not always be in the on-chip cache. We use a cache miss predictor [11] for L2 cache. If the predictor detects that the requested data is an L2 miss, the related MC is used as the location of that data.

4.2 Single Statement Splitting

With this knowledge of the location of data, we use Kruskal's algorithm (between lines 20 and 29 in Algorithm 1) to generate an Minimum Spanning Tree (MST), which gives the minimum possible total edge weight. Recall that edge weights indicate data movements (Section 3.2). Thus, the minimized total edge weight gives us the minimized total data movement for a program statement. Note that a program statement may contain different (arithmetic and logical) operators and/or parentheses which force certain priorities on computations. To support such computation priority, we employ a "level-based" optimization strategy. Specifically, all data accessed by a program statement are classified into nested sets (line 5). The computation priority degrades from the "innermost" set to the "outermost" set. Starting from innermost set to outermost set, we generate MST in each level and consider the already-processed set as a "single component" in the next level. For example, let us consider the following statement: $x = a * (b + c) + d * (e + f + g)$. The nest sets after data classification is $(a, (b, c), d, (e, f, g))$, based

[2]We use the term "statement instance" to refer to the execution of a statement in a given loop iteration. Consequently, a given (static) program statement can have as many instances as the number of iterations that enclose it.

Algorithm 1 NDP-aware subcomputation scheduling

```
INPUT: Number of nodes (N); Statement (S_k(j))
OUTPUT: Subcomputation-to-node mapping
 1: function SINGLE_STATEMENT_SPLITTING(N, S_k(j), variable2node_map)
 2:     //Initialization
 3:     MSTedges ← ∅
 4:     //Parsing the variables in S_k(j) and form the nested set considering op priority and parentheses
 5:     VariableSets ← variable_parsing(S_k(j))
 6:     while from innermost set Vset_x in VariableSets do
 7:         NodeSet ← ∅
 8:         EdgeSet ← ∅
 9:         for each element V_i in Vset_x do
10:             //node_{V_i} can be a single node or a set of nodes.
11:             node_{V_i} ← GetNode(V_i, variable2node_map)
12:             if node_{V_i} not in NodeSet then
13:                 NodeSet ← NodeSet ∪ node_{V_i}
14:                 for node_{V_k} in NodeSet do
15:                     MD = Manhattan_distance(node_{V_k}, node_{V_i})
16:                     EdgeSet = EdgeSet ∪ (node_{V_k}, node_{V_i}, MD)
17:                 end for
18:             end if
19:         end for
20:         //Solve the MST problem using Kruskal's algorithm.
21:         Sort EdgeSet in increase order based on MD.
22:         while each (node_{V_i}, node_{V_j}, MD) in EdgeSet do
23:             set_i = getSet(node_{V_i})
24:             set_j = getSet(node_{V_j})
25:             if set_i ≠ set_j then
26:                 MSTedges = MSTedges ∪ (node_{V_i}, node_{V_j}, MD)
27:                 set_i ∪ set_j
28:             end if
29:         end while
30:     end while
31:     return MSTedges
32: end function
33: //Scheduling
34: variable2node_map ← ∅
35: for j from 0 to the size of statement_window do
36:     MSTedges_k(j) = Single_Statement_Splitting(N, S_k(j),
37:                                 variable2node_map)
38:     variable2node_map ∪ (S_k(j), NodeSet(S_k(j))
39: end for
40: for j from 0 to the size of statement_window do
41:     while MSTedges_k(j) ≢ ∅ do
42:         pick up node_s only has one edge in MSTedges_k(j)
43:         while (node_s, node_n, MD) in MSTedges_k(j) do
44:             remove (node_s, node_n, MD) from MSTedges_k(j)
45:             if workload_balance(node_n) then
46:                 do V_s op V_n on node_n
47:             end if
48:             if node_n is store node then
49:                 break
50:             end if
51:             mark node_s as visited child node
52:             if node_n has child nodes not visited then
53:                 break
54:             end if
55:             node_s ← node_n
56:         end while
57:     end while
58: end for
```

on the computation priority and parentheses. First, an MST is generated for (e, f, g). Second, an MST is generated for (b, c). Next, an MST if generated for $(a, (b, c), d, (e, f, g))$, considering both (b, c) and (e, f, g) as two "single" components. This level-base optimization guarantees program correctness.

4.3 Subcomputation Scheduling

Now we answer the question of where to schedule the subcomputations so that the data movement is minimized as specified in the related MST. We discuss our scheduling strategy for the single statement and multi-statement cases separately, as the latter case presents opportunities for both NDP and data locality.

Single statement: The rule of subcomputation scheduling for a single statement is that each edge in the MST should be traversed only once so that the total data movement is minimized. We start by picking a leaf node in the MST, as our start node ($node_s$). We follow the edge to its parent node ($node_n$) and schedule the subcomputation on $node_n$. We then move from $node_n$ to its parent node until we reach a node that has more than one child nodes. Such a node with 2 or more child nodes indicates that it needs a "synchronization" to wait for the results of the subcomputations from all its child nodes, before it can compute. Once all subcomputation results are collected on that node, our algorithm moves forward and finally reaches the last node where the final result is stored. Note that subcomputation scheduling at a single-statement granularity *maximizes* NDP by *minimizing* the number of links traversed by all the data accessed by the statement.

Multiple statements: While single statement optimization minimizes data movement, considering multiple statements together allows us to take advantage of "inter-statement" data locality. To support the optimization of multiple statements, we employ a *window-based* algorithm for subcomputation scheduling. We leverage our algorithm to take into account all the statements in a window by exploiting a $variable2core\_map$ (line 34) structure. This compiler-defined data structure tells which nodes have already-scheduled subcomputations that require the same data block. That is, the requested data blocks are most likely presented in the L1 cache of those nodes. This information is then used for constructing the subsequent MSTs, and as a result, the constructed MSTs benefit from *both* NDP and data reuse.

While different "window" definitions are possible, in this work, a window contains a number of "consecutive program statements", which may or may not belong to the same loop iteration. For instance, in a loop nest with 4 statements in its body, a window of size 4 means that each window contains one iteration. Hence, all 4 statements that belong to the same loop iteration are *considered together* when we perform our computation partitioning and subcomputation placement. On the other hand, for the same loop nest, a window size of 2 means that the first window will contain the first two statements to be executed by the first loop iteration, the second window will contain the remaining two statements to be executed by the first loop iteration, and the third window will contain the first two statements to be executed by the second loop iteration, and so on. As another example, a window size of 8 indicates that each window will contain all the statements that belong to 2 consecutive loop iterations.

It is to be observed that our approach in a sense considers the impact of a potential L1 miss as a "data movement cost". More specifically, if we miss in L1, we will go to the home bank/node of the requested (missed) data and fetch it, which means visiting some network links (data movement). Consequently, in any window size of more than 1 statement, when we place a subcomputation of a statement into a node by considering a potential L1 hit in that node (due to the fact that a previous subcomputation has accessed the same data), what we actually do is to *reduce* the data movement (network footprint). This is because, if the subcomputation would be placed into another node (instead of the one that would generate an L1 hit), we would incur extra data movements to bring the data from its L2 home to another L1. Thus, we can conclude that, in our approach, the costs of all on-chip cache activities are measured in terms of data movements.

4.4 Impact of Window Size

The window size in our algorithm determines the scope of scheduling. The *minimum* window size is 1, corresponding to the single statement optimization. Although the single statement optimization is certainly ideal from an NDP perspective for each single statement, it does *not* capture the data reuse opportunities with the nearby statements. On the other extreme, a very large window size[3] considers a large number of statements together and can thus potentially exploit more data locality (L1 hit) opportunities, but it has several drawbacks. First, the compilation complexity increases as the number of statements in a window increases. Second, an ideal scheduling may not exist with a large window size. For example, let us assume that there are 10 statements sharing a particular data element x. It is possible that the first five statements prefer moving x to n_1, whereas the remaining five statements prefer moving x to node n_2. If we consider all 10 statements together, we cannot find an ideal location (node) to accommodate both constraints. Third, a very large window size may bring negative effects while exploiting NDP and data reuse due to the L1 cache pollution. For example, a data block might be evicted from a node's L1 cache, due to the fact that the subcomputation requesting that data block is executed too late in a large window case. Therefore, identifying a proper window size is non-trivial and requires a careful consideration of the *tradeoff* between NDP and data locality.

To determine a proper window size, we implement a "preprocessing step" in our compiler before the scheduling of any subcomputation is finalized. More specifically, we initialize the window size to 1 statement as our initial configuration, and compute the resulting data movements. We then increase the window size to 2 statements, repeat the same process and determine the number of data movements. As discussed above, we take into account L1 locality as well (since now we consider two statements together). We continue in this fashion until we compute the number of data movements that would be experienced if we use 8 statements as our window size.[4] Finally, among all the window sizes checked, we

[3] In the extreme case, the loop nest being optimized can be completely unrolled and the resulting loop body – with all the statement instances in it – can be considered as one gigantic "window".

[4] We could not find any loop nest in our application programs, for which a window size of more than 8 program statements generated the best result.

prefer the one that minimizes the total data movements and finalize our subcomputation scheduling accordingly.

It also needs to be noted that, each window size means a different way of grouping program statements in a loop. For example, in a loop body with 1 statement, a window size of 2 assigns statement instances into groups where each group has 2 statement instances (e.g., the first group/window will have the statements that will be executed by the first two iterations), and similarly, a window size of 3 means that the statement instances are divided into groups of 3 (statement instances). Consequently, two statement instances that would be in the same group (window) when using a smaller window size can map to different windows when using a larger window size, and as a result, we may miss the opportunity of exploiting the data reuse between them (if there exists one). The last example of Section 5 illustrates this scenario for a loop whose body contains 4 statements. This is why, instead of working arbitrarily with a very large window size, our algorithm considers different window sizes before finalizing the subcomputation scheduling.

4.5 Load Balancing, Parallelism, Synchronization, and Code Generation

Load balancing: Our optimization partitions a program statement into subcomputations and assigns each subcomputation to a node with the goal of minimizing data movements. If we perform this assignment without considering the loads of the individual nodes, it is possible that some nodes would take a large amount of subcomputations, whereas some other nodes would be lightly-loaded, leading to workload imbalance across nodes. To address this potential problem, our approach works in a load-balancing manner. Recall that subcomputations are assigned to different nodes while going from the leaf nodes to the parent nodes on an MST. The compiler considers the number of subcomputations already assigned to each node when deciding where to assign new subcomputations[5]. Our scheduler assigns a subcomputation to a node only if the target node satisfies: 1) the minimum data movement requirement and 2) the resulting workload is balanced. Specifically, if assigning a subcomputation to a node would cause that node taking more than 10% (a configurable parameter in our implementation) extra load than the next highly-loaded node, our scheduler skips this node and moves to the next one.

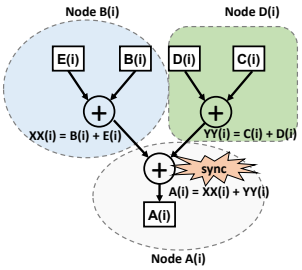


Figure 6: Subcomputation scheduling and synchronization.

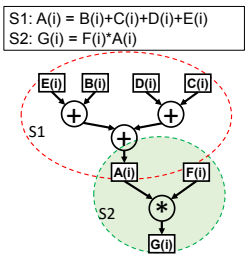


Figure 7: A case with two statements having an inter-statement dependency.

[5]The cost of subcomputations is measured as the number of operations, except that division is considered 10x costlier compared to addition/multiplication.

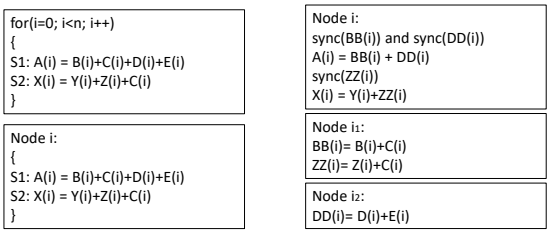
Intra-statement parallelism and synchronizations:

We now discuss our approach from the perspective of "intra-statement parallelism". Given an example statement $A(i) = B(i) + C(i) + D(i) + E(i)$, as shown in Figure 6, our approach breaks it into subcomputations $XX(i) = B(i) + E(i)$, $YY(i) = C(i) + D(i)$, and $A(i) = XX(i) + YY(i)$. As a result, $XX(i) = B(i) + E(i)$ and $YY(i) = C(i)+D(i)$ are scheduled on different nodes and they are executed in parallel. In other words, our approach introduces more parallelism compared to the original execution of $A(i) = B(i)+C(i)+D(i)+E(i)$.

To ensure the correctness of program execution, "synchronizations" are needed among subcomputations. As illustrated in Figure 6, $n_{A(i)}$ needs a synchronization barrier in order to get the updated values of $XX(i)$ and $YY(i)$ before computing $A(i) = XX(i) + YY(i)$. While these (point-to-point) synchronizations can have a negative impact on performance, we handle synchronizations using two strategies that help us minimize their penalties on performance. First, our approach compensates the synchronization overhead by taking advantage of subcomputation-level parallelism whenever it is possible to do so. Second, we implemented in our compiler a "transitive closure" based *synchronization minimization* strategy to remove redundant synchronizations. Our strategy is, at a high level, based on the scheme discussed in [51]. However, while the strategy in [51] targets synchronizations due to shared data accesses, our implementation targets synchronizations due to subcomputations. Specifically, our compiler builds a "synchronization graph" where each node corresponds to a subcomputation instance (i.e., a subcomputation parameterized by loop variables) and there is an edge (called "synchronization arc") between two subcomputation instances, sub_a and sub_b, if a synchronization is needed between them (say, from sub_a and sub_b). Let us now assume that there is a chain of synchronizations involving subcomputations sub_1, sub_2, $\cdots$ sub_{r-1}, sub_r. Assume further that there is also another chain that includes only sub_1 and sub_r. In this case, this latter synchronization is redundant as it is already captured by the first chain. Our compiler detects and drops such redundant synchronizations. In our experiments, we explicitly quantify both subcomputation-level parallelism and synchronization overheads. Further, the performance improvements reported in this paper include all the synchronization overheads brought by our approach.

Inter-statement dependencies: There can be inter-statement dependencies (flow/anti/output dependencies) among consecutive statements, and such dependencies can limit the potential instruction level parallelism (ILP). However, inter-statement dependencies pose no problem for our optimization because of two reasons. First, we do not migrate the final result to another node. The final output data is stored on the same node where it was supposed to be without our optimization. This guarantees that, in a flow dependency for example, the subsequent statement will get the updated value. Second, our scheduler captures the inter-statement dependencies and inserts necessary synchronizations. Figure 7 gives an example of an MST having a window size of two statements. A flow dependency due to $A(i)$ exists between the two statements. As our scheduling traverses the MST, $A(i)$ is computed before the computation of $G(i)$.

Let us also briefly discuss how our approach handles "may-dependences". In our target applications, may-dependences can occur mainly in two scenarios. First, if there is a dependence across



(a) Sample loop and default node assignment. **(b) Code after our optimization.**

Figure 8: High-level view of code generation.

a conditional (i.e., the statement after conditional depends on the statement before that conditional), we copy and schedule the conditional together with the subcomputations to preferable nodes. Although doing so may introduce additional data fetches due to the duplicated conditionals[6], conditionals in nested loop-bodies are rare in our target applications. But, these extra overheads are nevertheless included in our experimental results. Second, indirect array accesses (which are common in irregular applications) might cause may dependences as well. For example, statement-A writes to array $X[i]$ and statement-B reads array $X[Y[i]]$. In this situation, statement-B may depend on statement-A due to the value of $Y[i]$, whose value cannot be determined at compile time. To target this type of may dependences, we apply inspector-executor paradigm [15]. Specifically, in most loop-dominated applications, there is an outer (timing) loop that iterates either a fixed number of times or until a convergence criterion is met. In our implementation, the compiler inserts an inspector-code for the beginning iterations of the timing loop. The inspector is invoked at runtime to gather the may dependence information which are being used later in the executor phase (i.e., the remaining iterations of the timing loop). Our subcomputation scheduling scheme is only invoked in the executor stage and uses the may dependence information collected by inspector to make decisions on subcomputation scheduling. It is to be mentioned however that, this problem brought by may dependences is not unique to our proposed scheme. More specifically, even a scheme that maps computations at an iteration granularity (i.e., does not break up an interation) would also need a mechanism to ensure correct execution in the existence of may dependences.

Code generation: In the default computation assignment, a loop is distributed over available nodes in the target manycore at an "iteration granularity". That is, each node is assigned one (or several) "entire loop" iteration(s). As explained above, our approach splits the statements and performs scheduling at a "subcomputation" granularity. Figure 8 shows the high-level code generated by our approach. Let us assume a loop body with two statements per iteration, as shown in Figure 8a. Suppose that iteration i is assigned to node i in the default execution (also shown in Figure 8a). After our optimization (window size is 2 statements), the statements (S1 and S2) are broken into subcomputations and scheduled on different nodes (i.e., node i_1 and node i_2), as illustrated in Figure 8b. Note that the subcomputations in node i_1 and node i_2 can potentially execute in parallel, as we have previously discussed. Synchronizations are inserted in the code segment scheduled to be executed in node i, to ensure the correctness of execution. It is important to emphasize that node i_1 and node i_2 also have their own assigned iterations and those iterations are optimized individually in the same way as those of node i, while ensuring load-balance. (i.e., some of the iterations originally assigned to nodes i_1 and i_2 will also be broken into subcomputations that will be mapped to other nodes).

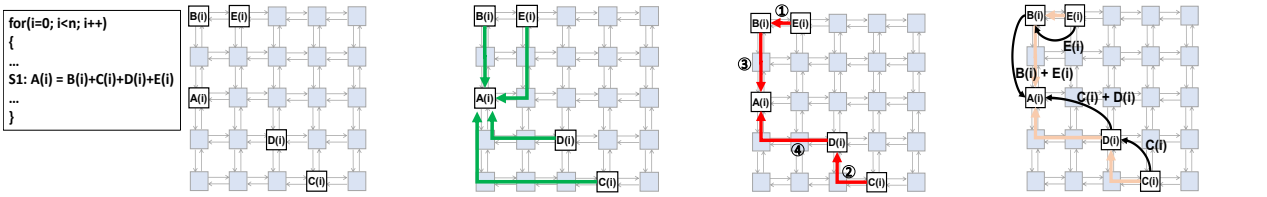
[6]The conditional itself may contain computations. Since it is being duplicated and copied to different nodes, each copy needs to fetch the data and perform the computation to evaluate the conditional.

5 EXAMPLES

Single statement: Let us first consider a simple case, shown in Figure 9: one statement with equal operator priorities and no parentheses. Let us assume that, $A(i)$, $B(i)$, $C(i)$, $D(i)$, and $E(i)$ are located in nodes $n_{A(i)}$, $n_{B(i)}$, $n_{C(i)}$, $n_{D(i)}$, and $n_{E(i)}$, respectively (as depicted in Figure 9a). In the default execution (shown in Figure 9b), $B(i)$, $E(i)$, $D(i)$, and $C(i)$ are fetched into $n_{A(i)}$ and the computation is performed in $n_{A(i)}$. The final result is also stored in the L1 cache of $n_{A(i)}$. According to our definition of data movement, the total number of data movements is 13. Our approach first generates an MST for this statement. We first pick up edge ① in Figure 9c since it has the smallest edge weight. We next pick up edge ②. Note that, if we have multiple edges with the same weight, we randomly pick one of them. We then select edge ③. Finally, we connect $n_{A(i)}$ and $n_{D(i)}$ using edge ④. The resulting MST is given in Figure 9c. Figure 9d shows how subcomputations are scheduled with respect to MST. We start with the leaf nodes and assign the subcomputations to their parent nodes. Suppose that we choose $n_{E(i)}$ (randomly selected from $n_{C(i)}$ and $n_{E(i)}$). We then move to $n_{B(i)}$ and decide to perform the subcomputation $B(i) + E(i)$ in $n_{B(i)}$. Next, we move to $n_{A(i)}$ and, since $n_{A(i)}$ is a parent node, it has to wait until all of the results expected from its child nodes are computed. Therefore, we pick the remaining leaf node ($n_{C(i)}$) and perform $C(i) + D(i)$ in $n_{D(i)}$. Finally, the last computation is performed, and the output data is stored in $n_{A(i)}$. Compared to default execution which involves 13 data movements, our optimization reduces the total data movements to 8.

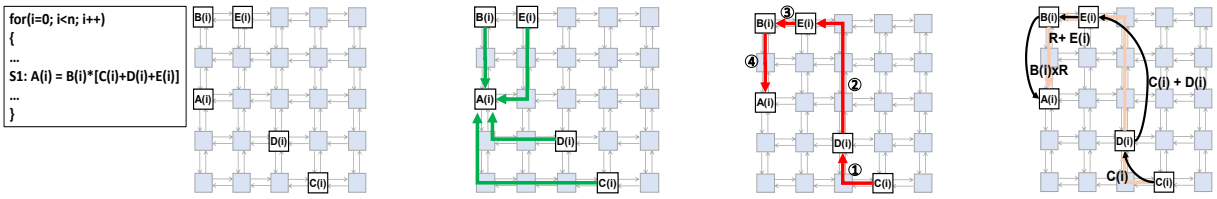
Single statement with parentheses: In our second example, as shown in Figure 10, we discuss the case of single statement consisting of priorities and/or parentheses. We assume that $A(i)$, $B(i)$, $C(i)$, $D(i)$ and $E(i)$ are located in the nodes shown in Figure 10a, and the default (unoptimized) execution fetches all data to $n_{A(i)}$ (Figure 10b). Since $C(i) + D(i) + E(i)$ needs to be computed before being multiplied by $B(i)$, our approach applies the "level-based" optimization discussed earlier. Recall from Section 4.2 that, data in a statement are parsed into nested sets. Statement splitting happens from the innermost set towards the outermost set, and a partial MST graph is formed for each nested set. In this example, the generated nested set is $\{A(i), B(i), \{C(i), D(i), E(i)\}\}$. We start by constructing an MST graph on $\{C(i), D(i), E(i)\}$. Our algorithm picks edge ① and then ②, as shown in Figure 10c. After that, $\{C(i), D(i), E(i)\}$ is considered as a "single component" in $\{A(i), B(i), \{C(i), D(i), E(i)\}\}$. Therefore, in the second step, we pick ③, as it has the shortest edge (distance 1) from $n_{B(i)}$ to "node"$\{C(i), D(i), E(i)\}$. The final MST is illustrated in Figure 10c. The scheduling strategy for this case is the same as in the previous example (Figure 10d), and as a result, we reduce the total data movements from 13 to 9.

Multiple statements: Next, using Figure 11, we discuss a multiple statements scenario. Assuming two statements: S1 ($A(i)$ =



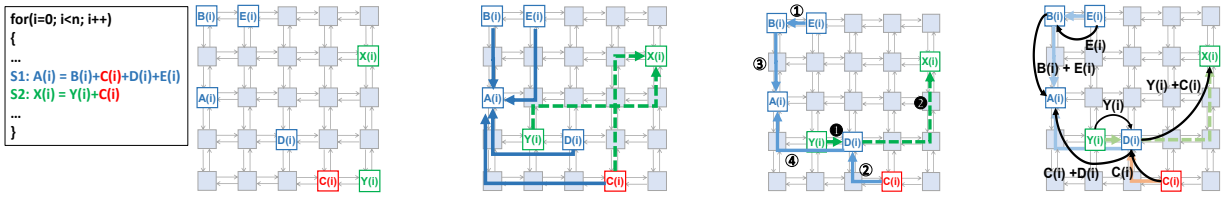
(a) Sample code and data placement. (b) Default execution. (c) MST. (d) Scheduling.

Figure 9: A single statement in a loop iteration.



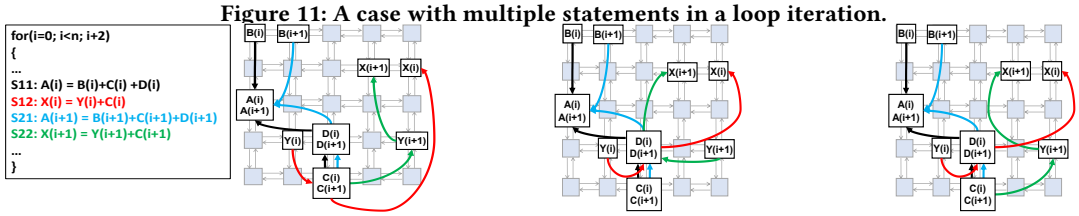
(a) Sample code and data placement. (b) Default execution. (c) MST. (d) Scheduling.

Figure 10: A single statement with parentheses in a loop iteration.



(a) Sample code and data placement. (b) Default execution. (c) MST. (d) Scheduling.

Figure 11: A case with multiple statements in a loop iteration.



(a) Sample code with window size=1. (b) Window size=2. (c) Window size=3.

Figure 12: Scheduling with different window sizes.

$B(i) + C(i) + D(i) + E(i)$), and S2 ($X(i) = Y(i) + C(i)$). Originally, data are in the nodes highlighted in Figure 11a and, in the default execution (Figure 11b), data are brought into $n_{A(i)}$ and $n_{X(i)}$ for the computations of S1 and S2, respectively. This results in 13 data movements for executing S1 and 9 data movements for executing S2. If we separately apply single statement optimization to S1 and S2, the total number of data movements of S1 is reduced to 8, whereas the total number of data movements of S2 is 7. However, this optimization strategy does not take into account the fact that $C(i)$ is also in the L1 cache of $n_{D(i)}$, because subcomputation $C(i) + D(i)$ in S1 is scheduled to $n_{D(i)}$.. To capture this opportunity, we construct the MST of S2 as shown in Figure 11c. We schedule $Y(i) + C(i)$ in $n_{D(i)}$ instead of $n_{C(i)}$ (Figure 11d). The benefits of doing so are two-fold. First, $C(i) + Y(i)$ in S2 can generate an L1 cache hit for $C(i)$ in the L1 cache of $n_{D(i)}$. Second, we reduce the total data movements caused by S2 to 5. As a result, the total number of data movements is reduced to 13 (compared to 22 in default execution, and 15 in the single statement at-a-time optimization).

Different window sizes: We now give an example illustrating the impact of having different window sizes when scheduling subcomputations. As shown in Figure 12, we assume that the maximum window size is 3 statements, and the loop body is unrolled by one iteration to have enough statements filling the window (labels as S11, S12, S21, and S22). Let us further assume that $D(i)$ and $D(i + 1)$ are in the same data block, indicating spatial locality between $D(i)$ and $D(i + 1)$. Similarly, $A(i)$ and $A(i + 1)$ as well as $C(i)$ and $C(i + 1)$ share the same data block and are represented by using a single square in Figure 12. Note that $B(i)$ and $B(i + 1)$ are not in the same data block and are mapped to different nodes. The remaining variables are in separate data blocks residing in the cache banks that belong to the different nodes. In the default execution, the total number of data movements for S11, S12, S21, and S22 are 9, 10, 3, and 7, respectively (29 in total). Note that the S21 only needs to fetch $B(i + 1)$ and hits its L1 cache on accessing $C(i + 1)$ and $D(i + 1)$ due to spatial locality. Figure 12a shows the scenario of a single statement (window size is 1) optimization. In this case, the total number of data movements for S11, S12, S21, and S22 are

Barnes	68.3%	Cholesky	97.2%	FFT	92.3%	FMM	74.4%
LU	90.7%	Ocean	77.3%	Radiosity	77.3%	Radix	84.2%
Raytrace	85.1%	Water	82.1%	MiniMD	95.5%	MiniXyce	93.8%

Table 1: The fraction of compile time-analyzable data references.

Barnes	63.1%	Cholesky	91.8%	FFT	84.5%	FMM	70.6%
LU	85.7%	Ocean	72.7%	Radiosity	69.9%	Radix	89.1%
Raytrace	80.2%	Water	77.6%	MiniMD	87.4%	MiniXyce	86.5%

Table 2: Cache hit/miss predictor accuracy.

App	add/sub	mul/div	others (shift, logical, etc.)	App	add/sub	mul/div	others (shift, logical, etc.)
Barnes	51.4%	26.2%	22.4%	Cholesky	39.4%	47.6%	13%
FFT	33.1%	46.5%	20.4%	FMM	47.2%	45.3%	7.5%
LU	41.8%	51.6%	6.6%	Ocean	52.2%	41.4%	6.4%
Radiosity	46.2%	33.4%	20.4%	Radix	39%	38.7%	22.3%
Raytrace	43.4%	49.7%	6.9%	Water	58.1%	28.2%	13.7%
MiniMD	44.4%	37.2%	18.4%	MiniXyce	46.3%	36.7%	17%

Table 3: The fraction of computation types offloaded.

6, 7, 6, and 6, respectively (25 in total). Note that S21 has 6 data movements because $C(i+1)$ is found in the L1 cache of $n_{D(i)}$ while computing $C(i+1) + D(i+1)$ in node $n_{D(i)}$. Therefore, there is no need to fetch $C(i+1)$ from $n_{C(i)}$. In the second scenario, shown in Figure 12b, two statements are considered together (i.e., a window size of 2) in an attempt to capture data reuses in L1. Subcomputations $Y(i) + C(i)$ and $Y(i+1) + C(i+1)$ are performed in $n_{D(i)}$. As a result, the total number of data movements for S11, S12, S21, and S22 are 6, 5, 6, and 5, respectively (22 in total). Figure 12c depicts the case where window size is 3 statements. Note that, in this case, the last statement (S22) is separated from the window. Therefore, S22 have no information of $C(i+1)$ is fetched into $n_{D(i)}$. Consequently, S22 cannot take advantage of the spatial locality between $C(i)$ and $C(i+1)$. In this scenario, the data movements for S11, S12, S21, and S22 are 6, 5, 6, and 6, respectively (23 in total). Therefore, in this example, our approach prefers a window size of 2 statements.

6 EXPERIMENTAL EVALUATION

6.1 Setup

All the experiments reported in this section are performed on a system that uses Intel Knight's Landing (KNL). KNL [57] is a state-of-the-art manycore system designed to deliver massive data-level and thread-level parallelism using high bandwidth memory. It is the newest incarnation of Intel Xeon Phi architecture and can be connected to a host machine using various interconnects such as PCIe, Ethernet, and Infiniband (in our experiments though, we used the native mode in which applications are booted from the KNL itself without any help from the host). KNL consists of 36 "tiles" interconnected by a 2D mesh on-chip network. Each tile contains 2 cores (a total of 72 cores), 4 vector processing units, and a 1 MB L2 bank. It has two types of memories: a conventional DDR4 (6 channels, 2400Mhz, can be expanded up to 384GB) and an on-package high bandwidth MCDRAM.

The underlying 2D-mesh in KNL provides three modes of clustered operation, each supporting "address affinity" at different levels. These modes differ primarily in how an L2 cache (last-level cache) miss is serviced. More specifically, these modes differ in relative positions of 1) the core (tile) that generates the L2 miss; 2) the tag directory that owns the missing address; and 3) the memory which supplies the requested data block. In *all-to-all mode*, the addresses are uniformly hashed over all memory, and a core can request data in any memory component. This mode certainly uses more of the on-chip network than the remaining two (explained shortly), but an off-chip access can travel long distances on the network. In *quadrant mode*, the 2D-mesh is assumed to be divided into 4 sections and memory addresses are assigned such that the location of the tag directory and the target memory are in the same section. In *SNC-4 mode*, the core that makes the memory request, the tag directory and the target memory are all in the same quadrant. The quadrant mode is the default mode (unless memory capacities in different channels are different from each other, in which case the all-to-all mode is the default) and most of the results reported in this section are collected with the quadrant mode. However, we also report a set of results when our approach is used in conjunction with the other two cluster modes as well.

In addition to the cluster modes, KNL supports three memory modes. In *cache mode*, MCDRAM is configured as a direct-mapped cache; in *flat mode*, both MCDRAM and DDR4 are presented as regular memory mapped in the same address space; and in *hybrid mode*, a portion of MCDRAM is configured as a cache and the remaining is configured as flat memory (this partitioning can be controlled at the system boot-up time). In most of our experiments, we use the flat mode (as it generates better results, when used without our approach, than the other two modes). But, we later report results with the other two memory modes as well.

Unless stated otherwise, all results are collected using this KNL system. A few experiments that delve into more detailed statistics use a simulation platform that models KNL.

We used 12 *multithreaded* applications as listed in Table 1 (taken from Splash-2 [63] and Mantevo [23] suites), and each application is run to completion. Table 1 gives the fraction of program data references that can be statically analyzable. That is, this gives the fraction of references for which we can identify the location of data (both in caches and in memory). Note that our scheme is implemented on top of data dependency analysis with static disambiguation [50] to detect potential memory aliasing. However, we believe that, with proper modifications, our approach can also be made to work with dynamic/speculative disambiguation [25] to better detect memory aliasing and further improve performance. The accuracy of the cache hit-miss predictor used in this work is given in Table 2. The dataset sizes manipulated by these applications range between 661MB and 3.3GB, and the L2 misses of the original applications (i.e., without our optimization) change between 16.4% and 37.2%. Table 3 gives, for each application in our experimental suite, the fraction of the type of computations that are re-mapped under our compiler scheme. For example, in Radiosity, 46.2% of the re-mapped computations are additions/subtractions, and 33.4% of them are multiplications/divisions. The remaining re-mapped operations constitute 20.4%.

The proposed compiler support is implemented using LLVM [38] as a source-to-source translator. Both the original version and optimized version of each application is then compiled using the same native Intel (node) compiler with the highest optimization level turned on.

In this work, we compare our compiler approach to a "default" computation placement strategy. It is to be emphasized that this default strategy against which we compare is also *highly optimized* from a data locality angle. More specifically, in this default strategy, iteration space is divided into chunks and each chunk is assigned to the most beneficial core using "profile data". This profile data

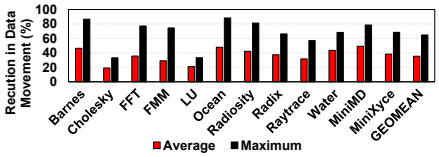


Figure 13: Data movement reduction over the original applications.

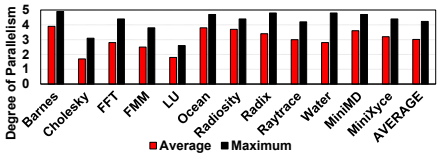


Figure 14: Degree of parallelism achieved through subcomputation scheduling.

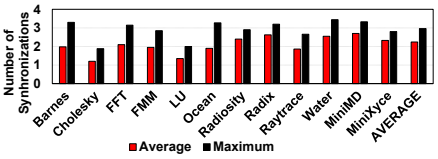


Figure 15: Number of synchronizations per statement due to subcomputation scheduling.

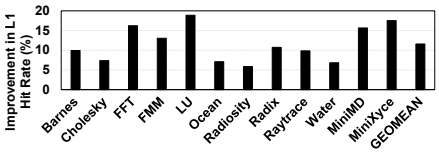


Figure 16: Improvement in L1 hit rate.

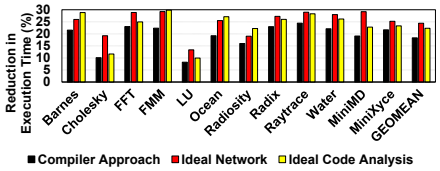


Figure 17: Reduction in execution times of our compiler-based approach and two ideal cases.

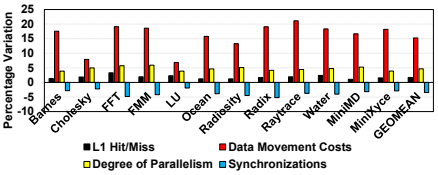


Figure 18: Contribution of different metrics to the execution time (simulation results).

captures which last-level caches or MCs a given chunk of iterations use, indicating subsequently which node would be the ideal one (from an LLC/MC locality viewpoint) for executing that chunk. This default strategy actually generates better results than some prior techniques which we also tested using KNL including [49] and [17]. That is, we also implemented the data locality optimization schemes proposed in [49] and [17] for our target architecture (both these techniques target last-level caches), and the collected experimental results revealed that, our profile-based default strategy generated around 8.3% and 12.6% performance improvement, on average, over them. In a sense, our default computation placement strategy represents the limit in "pure data locality (LLC performance) optimization" for our target architecture (assuming that iterations are not broken into subcomputations). Note however that, while [49], [17], and the default strategy are very good at optimizing data locality, *none of them specifically targets reducing data movement explicitly*. Therefore, our results presented below indicate the importance of minimizing on-chip data movements through computation partitioning and location aware computation placement and doing so in a data locality-aware fashion. All the performance improvement results brought by our compiler approach and reported in the rest of this section are *on top of* this optimized default computation placement strategy.

Both the default version and our optimized version are also fully optimized using the Intel node compiler with the highest optimization level, which basically implements all loop-level optimizations (e.g., unrolling) as well as all fine-grain parallelism optimizations such as SIMDization.

We want to re-iterate that, unless otherwise stated, the results presented below are collected assuming the quadrant cluster mode and flat memory mode. When using the flat mode, we need to decide which data structures are placed into MCDRAM. To do that, we used Intel Vtune (as advised in [27]) to determine candidate data structures for MCDRAM. This approach involves three steps, whose details are omitted due to lack of space: 1) profiling different data structures; 2) profiling data analysis; and 3) code modification. The first two steps identify the candidate data structures to place into MCDRAM, whereas the last step inserts pragmas in the code to enforce this placement.

6.2 Results

We start by presenting, in Figure 13, the reduction in data movement as a result of our optimization (over the default computation placement defined above). The first and second bars in Figure 13 represent the average (per statement) and maximum reductions (observed across all statements in the code) in data movement, compared to the default version. The geometric mean of the average movement reduction on the on-chip network is about 35.3%. We also observe that, in some applications such as Barnes, Ocean and MiniMD, this reduction is quite high, whereas in some others (e.g., Cholesky and LU) the benefits are not as significant. In this latter group, the original network footprint (the total number of links) is small, which makes our approach less effective.

As discussed earlier, in addition to reducing the data movements on the on-chip network, our approach also takes advantage of parallelism across subcomputations. Figure 14 plots the "degree of parallelism", that is, the average (first bar) and maximum (second bar) number of subcomputations executed in parallel per program statement. When averaged over all the application programs we tested, the degree of parallelism per statement is around 3. Applications such as Ocean and Barnes exhibit larger values of this metric since they have longer/more complex statements (compared to the others), and our compiler was able to extract more subcomputations that could be run in parallel. Recall that the parallel execution of subcomputations can bring synchronization overheads as well, which are plotted in Figure 15. Comparing this graph with that in Figure 14, we see that a higher degree of subcomputation parallelism usually corresponds to a larger number of synchronizations (as we stated earlier, a large fraction of synchronizations are eliminated thanks to our transitive closure based synchronization minimization).

Figure 16 gives the improvement of our approach brings over the L1 hit rate of the default strategy. As stated earlier, the default computation placement strategy is already highly optimized from a data locality angle, but it mainly targets the last-level cache (L2 in our case) performance. Our approach complements it by improving the L1 performance as well (in addition to taking advantage of NDP) when optimizing multiple statements within the same loop body. The results in Figure 16 indicate that our approach improves the L1 performance by 11.6% on average.

The cumulative impact of the changes in all these four metrics (magnitude of data movement, degree of parallelism, number of synchronizations, and L1 performance) is plotted in Figure 17 (first bar for each application) as the percentage reduction in execution time brought by our approach over the default case. When averaged over all 12 applications, our approach achieves an execution time improvement of 18.4%. In other words, careful placement of subcomputations on the 2D-mesh can lead to significant performance improvements, even over a locality-optimized computation placement.

To gain more detailed insight into the behavior and impact of our proposed approach, we also used a simulator (built upon GEM5[7]) that can give specific architecture-level statistics which cannot be easily collected from the real hardware. Specifically, we simulated a configuration that mimics KNL in detail and collected statistics to isolate the impact of the four metrics our approach affects: data movement, degree of parallelism, number of synchronizations, and L1 hit rate. We tested the following four schemes:

- S1: This version runs the original (default) code except that it has exactly the same L1 hit/miss profile as our optimized one[7]. This is achieved by enforcing, in simulation, the L1 hit/miss pattern of the optimized code on the default code[8]. In experimenting with this version, our goal is to answer the question of what would happened if we could somehow improve the L1 behavior but not affect the other three metrics.
- S2: This version also runs the default version but this time only the data movement costs are changed – they are exactly the same as those incurred in our optimized version.
- S3: In this case, we run the default version, the only change is that the degree of parallelism is the same as our optimized code, that is, exactly the same set of computations are executed in parallel. No other metric is affected.
- S4: In this last version, the default version is run with additional costs of the synchronizations incurred in the optimized version.

Figure 18 gives the execution time impact of these versions, normalized with respect to the default execution (higher the better). One can see that, the biggest improvements come from data movement reduction followed by the subcomputation parallelism. In fact, the geometric mean of the execution time improvement brought by the reduction in data movement alone is about 15.2%. We want to mention that our complete approach, when tested using the simulator, brought 19.7% performance improvement, meaning that data movement reduction alone is responsible for 77% of the total execution time improvement.

To illustrate that our approach does not lead to any network bottleneck, we present in Figure 19 the reduction in average and maximum on-chip network latencies brought by our optimized scheme over the default execution. This data is collected using our simulator. It can be observed that, our approach significantly reduces both the average network latency and the maximum network latency (which can be considered as a measure of congestion) for all applications tested. As a result, it does not lead to any additional congestion on the on-chip network.

[7]If a reference in our optimized version hits (resp. misses) in L1, in the default version it also hits (resp. misses).
[8]We first ran the optimized code and recorded its L1 hit/miss pattern. This pattern is then enforced on the execution of the default version.

6.3 Impact of Window Size

To determine the optimal window size to use in each loop nest, our approach considers all window sizes between 1 statement and 8 statements, and chooses the one that gives the minimum data movement. In Figure 20, we quantify the importance of this loop nest based strategy by comparing it against a "fixed window size" for *all* nests in the application. The first eight bars in Figure 20 gives the execution time improvements when using a fixed window size (varied from 1 statement to 8 statements) for all nests of an application. The last bar on the other hand is the result of our approach which customizes the window size to each loop nest in an application. Three important observations can be made from these results. First, in general, as expected, increasing the window size initially seems to be helping for each application. This is due to the fact that a larger window can better capture L1 locality, as discussed earlier. However, beyond a window size, the savings start to get reduced, primarily because L1 cache pollution/contention (i.e. the requested data block is no longer in the L1 cache when we execute the subcomputation on that node). To show this is indeed the case, we give in Figure 21 the variation in L1 hit rates as the window size is changed. Basically, comparing this graph with the one in Figure 20 clearly indicates that the execution time results follow the L1 hit rate trend. Second, the ideal (fixed) window size can change from application to application, and as such, it is difficult to adopt a globally acceptable value for it. Third, our compiler approach (last bar for each application) generates, for each application, a better result than the one that could be obtained when using the best-performing "fixed" window size. This means that, adapting the window size to the loop nest being optimized (that is, using different window sizes for different loop nests instead of fixing it for all loop nests) can be critical.

We want to emphasize that, the experiments above with a fixed window size (the first 8 bars for each application in Figure 20) take into account the impact of L1 reuse (like in our approach) as well. We also performed a set of experiments where, when we use a fixed window size, no potential L1 reuse is taken into account. This "reuse-agnostic" fixed window size approach generated, on average, 11% worse results than the "reuse-aware" fixed window size approach, whose results are plotted in Figure 20.

6.4 Comparison to Ideal Scenarios

In this section, we consider two different ideal scenarios: *ideal network* and *ideal data analysis*. To mimic the ideal network scenario. In the ideal network scenario, all messages in the network are assumed to take 0 cycle to complete. To mimic this ideal network (2D mesh) scenario, we collected network latency values (using code instrumentation) and deducted them from the overall execution times. The second bar for each application in Figure 17 gives the reduction in execution times with this ideal network scenario. As far as the geometric means are concerned, our approach (first bar for each application) and the ideal network scenario result in 18.4% and 24.4% improvements. We also note that our approach comes close to ideal case in applications such as Radix and Radiosity.

In the second experiment, we first profiled the application code and determined the precise data access pattern and location (which also implies 100% hit/miss prediction accuracy for cache). Then, in the second run, we used this profile data (in our optimized version)

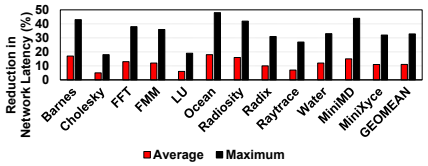


Figure 19: Reduction in on-chip network latencies (simulation results).

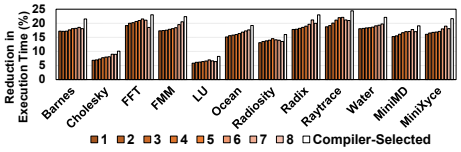


Figure 20: Results with different statement window sizes. 1-8 correspond to fixed window sizes for entire application.

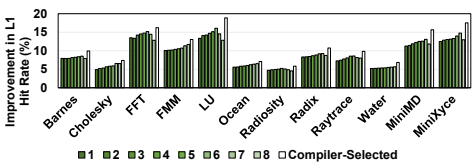


Figure 21: Improvements on L1 hit rates with different window sizes.

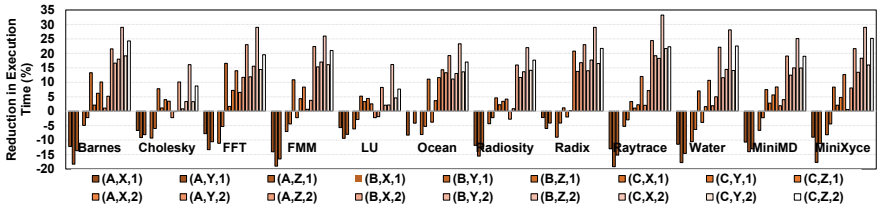


Figure 22: Results with the different KNL configurations (A: ALL-TO-ALL; B: QUADRANT; C: SNC-4; X: FLAT; Y: CACHE; Z: HYBRID; 1: original code; 2: optimized code). Note that (B,X,1) corresponds to the default KNL configuration running the original applications, and (B,X,2) corresponds to the same KNL configuration running the optimized codes.

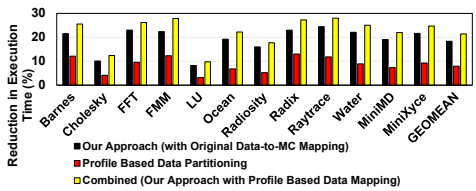


Figure 23: Comparison against a profile-based data mapping scheme.

to perform ideal data allocation and subcomputation assignment for each and every program statement. The third bar for each application in Figure 17 gives the reduction in execution time with this ideal data analysis (which is not implementable in a compiler in practice). These results in a sense also indicate how much opportunity we miss due to imprecise data dependence and data access pattern analysis of the compiler. The geometric mean of improvement in this case is about 22.3%. It is interesting that, in some applications, the ideal network generates better results than ideal data analysis, whereas in some others, it is the other way around. This is because, in the first group of applications, the compiler was able to figure out the best static placement for most of the program statements (see Table 1), and as a result, there is not much scope left for ideal data placement.

6.5 Results with Optimized Data Mapping

So far, we did not consider the option of changing the data-to-MC mapping. The second bar in Figure 23 gives the execution time improvement when using the default computation mapping but changing the data-to-MC mapping. Specifically, for each memory page, we recorded the number of accesses from each core and placed it into a bank that is controlled by the MC which is preferred by most of the cores (accessing that page). Note that this is a profile-based strategy and may not be implemented at compile time. The third bar on the other hand is the result from a combined scheme which first applies our computation mapping strategy and then uses the profile-based data-to-MC mapping explained above. We see that geometric means of improvement for our approach, profile based data mapping, and combined scheme are 18.4%, 7.9% and 21.4%, respectively. The data mapping scheme does not perform as good as our computation mapping, mainly because for the pages that are mostly accessed by the cores in the middle of the 2D grid, there is no clearly preferable MC, and data accesses from such cores cover long distances. On the other hand, our scheme can benefit from data mapping (as illustrated by the third bar in Figure 23) if the latter is selectively used to find better places for pages whose accesses cannot be fully optimized by our approach.

6.6 Energy Results

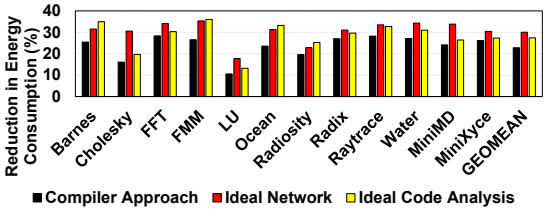


Figure 24: Reduction in energy compared to default computation placement (simulation results).

We now quantify the energy benefits of our proposed approach using our simulation infrastructure. Low level timing and energy parameters for all on-chip components (cores, network, caches, memory controllers) are faithfully modeled using CACTI [62] and McPAT [39]. The first bar for each application in Figure 24 plots the energy savings brought by our approach over the default case. We see that our approach leads an average of 23.1% energy savings over the default case. For comparison purposes, we also report, as the last two bars for each application, in the same figure, the energy savings obtained when using the two ideal schemes explained earlier. Overall, considering both Figure 17 and Figure 24 together clearly shows that our compiler-based approach is beneficial from both the performance and energy perspectives.

6.7 Results with the Other Configurations

As mentioned earlier, the results reported so far are collected using the quadrant cluster mode and flat memory model. In this subsection, we evaluate our approach under other possible configurations. While potentially the hybrid memory mode in KNL can generate better results than the flat and cache modes, it is not trivial to determine what should be the right partitioning of the

available MCDRAM capacity between cache and memory. In our experiments, we used a 50%-50% partitioning Figure 22 presents results with different (cluster mode, memory mode) configurations and with/without our approach. All the results presented are normalized against the default computation partitioning using the flat memory mode and quadrant cluster mode (note that the percentage execution time improvements reported so far in the paper are the improvements brought by (B,X,2) over (B,X,1)). One can make several observations from these results. First, our compiler-based approach improves the performance of the default execution for all application programs, for all (cluster mode, memory mode) combinations tested. Second, the differences between various cluster modes diminish when our approach is used. In particular, even the all-to-all cluster mode performs quite well when used along with our computation partitioning approach. Third, the options with the flat memory mode perform better than those with the cache memory mode. This means that, as long as one carefully chooses the data structures to place into MCDRAM, the flat memory option gives better results than the cache option. Furthermore, compared to the original versions, our approach achieves better savings with the flat memory mode. This is because the flat memory mode tends to increase the communication volume on the 2D mesh, and this in turn increases the opportunities for our approach. Fourth, the best-performing configuration is, as can be expected, (C,X,2), i.e., our optimized version running on flat memory mode and SNC-4 cluster mode, with a geometric mean of execution time improvement of 25% over the default execution. Finally, even (A,X,2) configuration outperforms (C,X,1), indicating that careful computation partitioning and placement, even when used with the all-to-all cluster mode, outperforms the SNC-4 cluster mode with the original applications (the corresponding geometric means are 8.5% and 6.7%).

7 RELATED WORK

Near-data processing related efforts: Prior works explored various concepts related to NDP [2, 6, 24]. Lipovski and Yu[43] presented an architecture where single-bit ALUs are attached to DRAM chips. Ahn et al. [2] proposed a processing-in-memory (PIM) architecture where PIM instructions execute in a data-locality aware fashion. Hsieh et al. [24] explored programmer-transparent schemes to offload select code segments to PIMs. Xu et al. [64] ported different deep learning algorithms to PIM architectures. Jayasena et al. [26] evaluated the energy efficiency of a GPU-based PIM architecture. Eckert et al. [19] have discussed the thermal feasibility of die-stacked PIMs. Azarkhish et al. [5] have designed a high bandwidth interconnect for Hybrid Memory Cube. The remaining related works along similar directions include [21, 34]. Our approach is complementary to most of these prior efforts, as we use compiler's help in exploiting the NDP opportunities. Also, our approach can be integrated with prior schemes that employ programmer annotations to further increase performance benefits.
Compiler related efforts: Kodukula et al. [37] proposed computation scheduling based on data flow instead of control flow. Ding et al. [18] proposed a compiler-based off-chip data access localization strategy. Chu et al. [13] presented a profile-guided scheme for distributing memory accesses across data caches in manycores. Their proposal reduces memory stalls and improves computation parallelism. Liu et al. [45] developed a compiler framework which separates data access and computation to optimize the network latencies experienced by off-chip data accesses. Bondhugula et al. [8] presented a source-to-source translator that optimizes both parallelism and locality for programs. Das et al. [14] proposed an application-to-core mapping strategy with the goal of reducing the interference in the on-chip network as well as memory controllers. Compared to these prior efforts, ours is the first one that implements subcomputation-level scheduling and considers both NDP and data reuse together. More specifically, the main difference between our work and prior compiler based efforts is that our work is finer grain and it takes into account locations of cores, caches, and memory controllers. In addition, our approach breaks computations into subcomputations to further reduce distance-to-data. Also, our approach is complementary to prior compiler-based CC-NUMA works [16, 31, 41, 42, 47, 59]. This is because, compiler-based works that target CC-NUMA systems try to collocate related computation and data in the same node in an attempt to maximize the number of local accesses and minimize the number of remote accesses. But, the smallest granularity they use in mapping is a single loop iteration. In contrast, our work considers finer-grain mapping of computations to cores. In a large shared-memory system, CC-NUMA techniques can be used across nodes, and our technique can be used inside each node (manycore).
OS/hardware/runtime related efforts: Dashti et al. [16] proposed a memory placement algorithm that reduces the on-chip network traffic congestion in NUMA systems. Liu et al. [46] presented an approach that integrates the OS memory management kernels with a mechanism which enables the memory controllers to schedule memory requests in a clustered manner. Augonnet et al. [4] developed a runtime system that generates and schedules parallel tasks for heterogeneous systems. Aji et al. [3] developed an OpenCL runtime which can map command queues to devices. Their scheduler includes a static profiler and a dynamic profiler. Other efforts include [1, 29, 30, 32, 44, 60]. These approaches are orthogonal to ours, and they require either new hardware or modifications to the runtime system.

8 CONCLUSIONS

In this work, we proposed a compiler-directed approach for near data computing (NDP) focusing on data-intensive application programs executing on manycore systems. Our approach partitions a set of computations into subcomputations and schedules the resulting subcomputations on cores with the goal of minimizing the distance between computation and data, and at the same time, takes advantage of L1 locality. Our experiments indicate that the proposed compiler-based approach significantly reduces distance-to-data and leads to an average performance improvement of 18.4%.

ACKNOWLEDGMENT

We thank Prof. Simone Campanoni for his feedback and comments on the paper. We also thank the anonymous reviewers for their feedback. This research is supported in part by NSF grants #1205618, #1213052, #1212962, #1302225, #1302557, #1313560, #1320478, #1320531, #1409095, #1409723, #1439021, #1439057, #1526750, #1629129 and #1629915, and a grant from Intel.

REFERENCES

[1] Vignesh Adhinarayanan, Indrani Paul, Joseph L Greathouse, Wei Huang, Ashutosh Pattnaik, and Wu-chun Feng. 2016. Measuring and modeling on-chip interconnect power on real hardware. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd International Symposium on Computer Architecture*.

[3] Ashwin Mandayam Aji, Antonio J Pena, Pavan Balaji, and Wuchun Feng. 2015. Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL. In *IEEE International Conference on Cluster Computing*.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation* (2011).

[5] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2014. A Logic-base Interconnect for Supporting Near Memory Computation in the Hybrid Memory Cube. In *Proceedings of the 2nd Workshop on Near-Data Processing*.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro, IEEE* (2014).

[7] 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 Compute* (2011).

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[9] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE Computer Architecture Letters* (2016).

[10] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a Smarter Memory Controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*.

[11] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

[12] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*.

[13] Michael Chu, Rajiv Ravindran, and Scott Mahlke. 2007. Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*.

[14] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2012. Application-to-core Mapping Policies to Reduce Memory Interference in Multi-core Systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

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

[16] 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 18th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[17] Wei Ding, Mahmut Kandemir, Praveen Yedlapalli, Yuanrui Zhang, and Jithendra Srinivas. 2013. Locality-aware Mapping and Scheduling for Multicores. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[18] 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*.

[19] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. 2014. Thermal feasibility of die-stacked processing in memory. In *Proceedings of the 2nd Workshop on Near-Data Processing*.

[20] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: the Terasys Massively Parallel PIM Array. *IEEE Computer* (1995).

[21] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 2014. 3D-Stacked Memory-Side Acceleration: Accelerator and system design. In *Proceedings of the 2nd Workshop on Near-Data Processing*.

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

[23] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. Improving Performance via Mini-applications. *Sandia National Laboratories, Tech. Rep.* (2009).

[24] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture*.

[25] Andrew S Huang, Gert Slavenburg, and John Paul Shen. 1994. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *Proceedings of 21 International Symposium on Computer Architecture*.

[26] Nuwan Jayasena, Dong Ping Zhang, Amin Farmahini-Farahani, and Mike Ignatowski. 2015. Realizing the Full Potential of Heterogeneity through Processing in Memory. In *Proceedings of the 3rd Workshop on Near-Data Processing*.

[27] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science.

[28] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck Identification and Scheduling in Multithreaded Applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[29] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. 2015. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*.

[30] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2016. Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. ACM.

[31] 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*.

[32] 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 *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques*.

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

[34] Gwangsun Kim, John Kim, Jung Ho Ahn, and Yongkee Kwon. 2014. Memory Network: Enabling Technology for Scalable Near-Data Computing. In *Proceedings of the 2nd Workshop on Near-Data Processing*.

[35] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*.

[36] 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*.

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

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*.

[39] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[40] Shuangchen Li, Dimin Niu, Krishna T.Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

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

[42] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *Proceedings of the 13th Annual International Conference on Supercomputing*.

[43] G Jack Lipovski and Clement Yu. 1999. The Dynamic Associative Access Memory Chip and Its Application to SIMD Processing and Full-Text Database Retrieval. In *Proceedings of the 1999 IEEE International Workshop on Memory Technology,*

Design, and Testing.

[44] Gu Liu, Hong An, Wenting Han, Xiaoqiang Li, Tao Sun, Wei Zhou, Xuechao Wei, and Xulong Tang. 2012. FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[45] 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.*

[46] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques.*

[47] Henrik Löf and Sverker Holmgren. 2005. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th Annual International Conference on Supercomputing.*

[48] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture.*

[49] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas 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 18th International Conference on Parallel Architectures and Compilation Techniques.*

[50] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. 1991. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.*

[51] S.P Midkiff and D.A. Padua. 1991. A comparison of four synchronization optimization techniques. In *Proceedings 20th International Conference Parallel Processing 1991.*

[52] Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2015. VIP: Virtualizing IP Chains on Handheld Platforms. In *Proceedings of the 42nd International Symposium on Computer Architecture.*

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

[54] Prasanna Venkatesh Rengasamy, Haibo Zhang, Nachiappan Chidhambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2017. Characterizing Diverse Handheld apps for Customized Hardware Acceleration. In *Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC).*

[55] Erik Riedel, Christos Faloutsos, and David Nagle. 2000. *Active disk architecture for databases.* Technical Report. DTIC Document.

[56] 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.*

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

[58] Xulong Tang, Hong An, Gongjin Sun, and Dongrui Fan. 2013. A Video Coding Benchmark Suite for Evaluation of Processor Capability. In *SNPD.*

[59] 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).*

[60] 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.*

[61] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2017. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS).*

[62] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. 2008. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture.*

[63] 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 the 22nd Annual International Symposium on Computer Architecture.*

[64] Lifan Xu, Dong Ping Zhang, and Nuwan Jayasena. 2015. Scaling Deep Learning on Multiple In-Memory Processors. In *Proceedings of the 3rd Workshop on Near-Data Processing.*

[65] Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidhambaram Nachiappan, Anand Sivasubramaniam, Mahmut Kandemir, Ravi Iyer, and Chita R. Das. 2017. Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*